Artix™

Learning About Artix

Version 2.1, July 2004

Making Software Work Together™

Updated: 04-Jan-2005

M 3 2 0 7

Contents

List of Figures

Preface

What is Covered in this Book

Learning about Artix provides an introduction to IONA's Artix technology. This book gives a brief overview of the architecture and functionality of Artix, and a brief introduction to Web Services Definition Language (WSDL).

It also provides basic tutorial material for learning how to use Artix.

This book also provides guidance for finding your way around the Artix product library.

Who Should Read this Book

Learning about Artix is for anyone who needs to understand the concepts and terms used in IONA's Artix product.

It also provides examples that you can return to and work through at any time to increase your understanding of how Artix works, both from the command line and from the Artix Designer.

Organization of this Book

This book is contains two types of information that you can read separately or together, depending on how much detail you need to know about Artix.

Part I of the book contains conceptual information about Artix and WSDL:

- "Introduction" on page 3 introduces the Artix product, and the types of problem that it is designed to solve.
- "Artix Concepts" on page 17 explains the main concepts used in Artix.

- "WSDL Concepts" on page 27 explains the basics of Web Services Definition Language (WSDL).

Part II contains examples that you can work through to learn more about using Artix:

- "Coding the Web Service" on page 39 details how to use the command-line tools to build an Artix solution.
- "Using the Artix Designer" on page 69 gives an overview of the Artix Designer GUI tool, and explains how to use this tool to perform the same tasks described in the command-line chapter.
- "Faults and Exceptions" on page 131 explains how to declare faults in WSDL files and how to handle the corresponding C++ and Java exceptions in Artix client and server applications.

There is also a "Glossary" on page 147 of this book to help you with any unfamiliar terms.

Online Help

The Artix Designer includes comprehensive online help, providing:

- Detailed step-by-step instructions on how to perform important tasks.
- A contextual description of each screen.
- A full search feature.

There are two ways to access the online help: via the **Help** menu in the Artix Designer, or by clicking the **Help** button on any interface dialog.

In addition, online help is provided for the Artix integration with BMC Enterprise Management Systems. See the *BMC Patrol* ***Help*** menu for details.

Additional Resources for Help

The IONA knowledge base contains helpful articles, written by IONA experts, about Artix and other products.

The IONA update center contains the latest releases and patches for IONA products.

If you need help with this or any other IONA product, contact IONA at: support@iona.com.

Comments on IONA documentation can be sent to:
docs-support@iona.com .

Document Conventions

Typographical conventions

This book uses the following typographical conventions:

`Constant width`	Constant 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 `CORBA::Object` class. Constant width paragraphs represent code examples or information a system displays on the screen. For example: `#include <stdio.h>`
Italic	Italic words in normal text represent *emphasis* and *new terms*. 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: `% cd /users/`*your_name*

Keying Conventions

This book uses the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, a prompt is not used.
%	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 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.
|	A vertical bar separates items in a list of choices enclosed in {} (braces) in format and syntax descriptions.

Part I

Introduction to Artix and WSDL Concepts

In this part

This part contains the following chapters:

CHAPTER 1

Introduction

This chapter introduces the main features of Artix, and describes where to look in the documentation for further information.

In this chapter

This chapter discusses the following topics:

What is Artix?

Overview

Artix is a Web services-based solution for enterprise application integration. It represents a new approach to application integration that exploits the middleware technologies and the products already present within an enterprise. This new approach also allows Artix to rapidly provide integration solutions that increase operational efficiencies, and enable an enterprise to adopt or extend a Service-Oriented Architecture (SOA).

This section contains the following topics:

Web service concepts

The information services community generally regards Web services as application-to-application interactions that use XML data representations and the hypertext transfer protocol (HTTP). The advantages of Web services lie in the fact that the data encoding scheme and transport semantics are based on standardized, non-proprietary specifications. Another advantage of Web Services is that string-based message content is human readable, can be created and manipulated by any programming development tool, and provides a high level of security and data integrity.

Artix technology

Artix allows organizations to define their existing applications as Web services, without worrying about the underlying middleware. It then provides the ability to expose those applications across a number of middleware technologies. Artix also enables developers to write new applications in C++ and Java that can also be exposed in a middleware

neutral manner. In addition, Artix provides enterprise levels of service such as session management, service look-up, security, and transaction propagation.

Artix uses IONA's proven *Adaptive Runtime Technology* (ART) to provide a high-speed, robust, and scalable backbone for Web services deployment. In addition, Artix extends ART and the Web service metaphor by using the *Artix Bus*, IONA's transport and message format switching technology. The Artix Bus enables you to create Web services that communicate using protocols other than SOAP over HTTP. For example, you can develop and deploy Web services using proven enterprise quality communication mechanisms such as TIBCO Rendevous, CORBA, and IBM WebSphere MQ.

Benefits of Artix

Artix differs from the integration approach used by Enterprise Application Integration (EAI) products. The EAI approach uses a canonical format in a centralized EAI hub. All messages are transformed from a source application's native format to this canonical format, and then transformed again to the format of the target application. Each application requires two, typically proprietary, adapters that translate to and from the canonical format.

Requiring two transformations for every message incurs high overheads. Many enterprises would prefer a high-performance solution that directly transforms a small set of message types instead of a more general solution with lower performance.

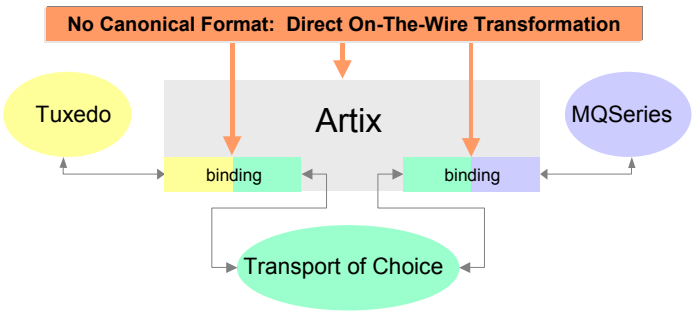


Figure 1: *Artix High-Performance Architecture*

Artix connects applications at the middleware transport level and translates messages only once. It hides the details of the connection and provides high performance. Figure 1 shows an example Artix integration between BEA Tuxedo and IBM MQSeries.

Artix also enables you to obtain maximum value from your IT assets through the reuse of your existing systems. You can lower operating costs by consolidating diverse systems and existing infrastructure.

Lastly, Artix provides a range of easy-to-use tools that enable you to create, manage, and deploy your integration solutions. These include GUI tools, command-line tools, and APIs. For example, Figure 2 shows the main window of the Artix Designer GUI tool. This tool automates and simplifies the creation of Web service integration applications.

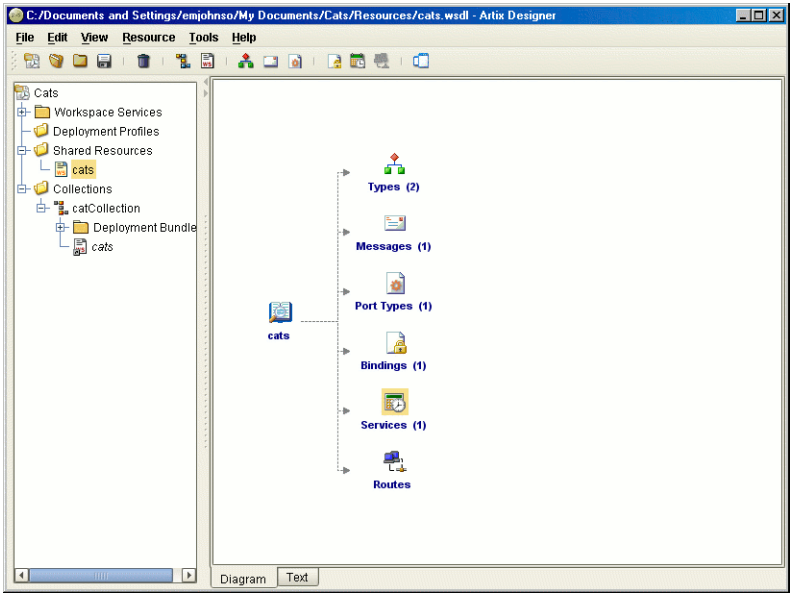


Figure 2: *Artix Designer GUI Tool*

Using Artix

There are three ways to use Artix:

- First, you can write applications using the Artix Application Programming Interface (API). In this situation, you are writing new applications using Artix as your development tool. This is the approach used in the examples in this book.
- Second, you can integrate two existing applications, built on different middleware technologies, into a single application. In this situation, developers work with their current development tools and Artix functions as a broker between the two dissimilar data encoding schemas and transport protocols.

This approach requires the extended functionality of the Artix Advanced or Enterprise products, and is not covered in this book.

- Lastly, you can use Artix as a replacement for other middleware transport protocols. Your application code remains unchanged; the Artix libraries replace the middleware libraries within your executable. This approach is also not covered in this book.

Becoming proficient with Artix

To become an effective Artix developer you need to understand four central concepts, only one of which is related to writing code.

- First, you need to understand the syntax for WSDL files and the Artix extensions to the WSDL specification.
- Second, you need to understand the relationship between Artix WSDL extensions, Adaptive Runtime Technology plug-ins, and setting configuration entries.
- Third, if you are programming in C++ you need an understanding of the Artix API, and the IONA and Artix foundation class libraries, which you can use in your application.
- Fourth, you must gain proficiency with the Artix Designer, a GUI tool through which you can write and edit WSDL files, convert CORBA Interface Definition Language (IDL) files, data files, and COBOL copybooks into WSDL, and generate code.

This book introduces concepts in all four of these categories. The other Artix product documentation covers each of these concepts in greater detail.

Artix features

Artix includes for the following unique features:

- Support for multiple transports and message data formats
- C++ and Java development
- Message routing
- Transaction support
- Asynchronous Web services
- Role-based security, single sign on, and security integration
- Session management and stateful Web services
- Look-up services
- Load-balancing

- Integration with EJBs
- Easy-to-use development tools
- Support for .NET
- Integration with enterprise management tools such as IBM Tivoli and BMC Patrol
- Support for a wide range of codesets
- Support for XSLT-based message transformation

Note: Single sign-on, locator look-up services, and session management are not available in all editions of Artix. Please check the conditions of your Artix license to see if your installation supports these features.

Supported transports and protocols

A *transport* is an on-the-wire format for messages; whereas a *protocol* is a transport that is defined by an open specification. For example, MQ and Tuxedo are transports, while HTTP and IIOP are protocols.

In Artix, both protocols and transports are referred to as transports. Artix supports the following message transports:

- HTTP
- BEA Tuxedo
- IBM WebSphere MQ (formerly MQSeries)
- TIBCO Rendezvous™
- IIOP
- CORBA
- Java Messaging Service

Note: Only HTTP is available in all editions of Artix. Check the conditions of your Artix license to see if your installation supports the remaining transports.

Supported payload formats

A *payload format* controls the layout of a message delivered over a transport. Artix can automatically transform between the following payload formats:

- G2++
- FML (a Tuxedo format)
- FRL (fixed record length)

- Tagged (variable record length)
- SOAP
- TibrvMsg (a TIBCO Rendezvous format)
- Pure XML

Note: SOAP is the only payload format available in all editions of Artix. Check the conditions of your Artix license to see if your installation supports the remaining payloads.

Further information

For more information about supported transports and payload formats, see *Designing Artix Solutions*.

For information about Artix mainframe support, see the Artix Mainframe documentation, available at: /http:/www.iona.com/support/docs/index.xml.

Solving Problems with Artix

Overview

Artix enables you to easily solve problems of how to integrate existing back-end systems using Web services. It also enables you to develop new Web services using C++ or Java, and retain all of the enterprise levels of service that you require.

This section describes the three main phases in an Artix solution:

- "Design phase"
- "Development phase" on page 12
- "Deployment phase" on page 12

Artix integration solutions that use a standalone Artix service do not always require the development of any code, and can involve design and deployment phases only.

Design phase

In the design phase, you map out the architecture of the systems that you want to integrate or develop. You decide what services you want to build, what operations each service needs, and the data that the services needs to exchange.

After making these decisions, you map the information into Artix contracts that describe the services, operations, and data types. As part of this step, you also map out the transports used by each service and any routing rules that will be used.

The Artix Designer GUI and command-line tools automate the mapping of your service descriptions into WSDL-based Artix contracts. These tools enable you to:

- Import existing WSDL documents (for example, those generated by third-party tools).
- Generate a WSDL contact from scratch.
- Generate a WSDL contract from an external metadata source (for example, CORBA IDL).

Development phase

If your solution involves creating new applications, a custom router, or locator or session management features, you need to develop some Artix application code (C++ or Java). This involves generating client stub code and server skeleton code from the Artix contracts that you created in the design phase. You can generate this code using the Artix Designer GUI and command-line tools.

When you have generated the client stub code and server skeleton code, you can then develop the code that implements the business logic you require. Artix takes care of generating the starting point code, and you can then use your favorite development environment to develop and debug the application code.

If necessary, Artix also provides advanced APIs for directly manipulating SOAP messages, and for writing SOAP message handlers. These can be plugged into the Artix runtime for custom-built processing of SOAP messages.

Deployment phase

In the deployment phase, you take the Artix contracts from the design phase, and any applications created in the development phase, and deploy your integrated system. You might need to modify the generated Artix configuration files, or edit the Artix contracts describing your solution to fit the exact circumstances of your deployment environment.

Applications that use Artix can be deployed in two different ways:

Embedded mode

In embedded mode, applications are modified to invoke Artix functions directly and locally, instead of invoking a standalone Artix service. This approach is the most invasive to the application, but also provides the highest performance. Embedded mode requires linking the application with the Artix-generated stub and skeleton code to connect the client and server to the Artix Bus.

Standalone mode

In standalone mode, Artix runs as a separate process that is invoked as a service. Standalone mode provides a zero-touch integration solution that does not involve any coding. However, standalone mode is less efficient than embedded mode.

When designing a system, you simply generate and deploy the Artix contracts that specify each endpoint of the Artix Bus. Because a standalone switch is not linked directly with the applications that use it (in embedded mode), a contract for standalone mode deployment must specify routing information.

Using the Artix Documentation

Overview

The Artix documentation library consists of a number of guides to help you understand and use Artix. The guides are broken down into groups reflecting the three phases of Artix problem solving. This section gives a brief overview of each guide and suggests an order in which to read the library.

This section contains the following topics:

If you are new to Artix

If you are approaching Artix for the first time, it is suggested that you work through the library in the following order:

1. *Learning about Artix (this book)*
2. *Designing Artix Solutions*

   This book describes how to use the Artix GUI or the command line tools to describe your services in an Artix contract. It also provides detailed information about the WSDL extensions used in Artix contracts and explains the mappings between data types and Artix bindings.
3. *Developing Artix Applications in C++*
   Developing Artix Applications in Java

   The development guides discuss the technical aspects of programming applications using the Artix API.
4. *Deploying and Managing Artix Solutions*

   This book describes deploying Artix enabled systems. It provides detailed examples for a number of typical use cases.
5. *Artix Security Guide*

   An introduction to the security features in Artix.
6. *Artix IBM Tivoli Integration Guide*

Describes how to integrate Artix with the IBM Tivoli enterprise application.

7. *Artix BMC Integration Guide*

   Describes how to integrate Artix with the BMC Patrol enterprise application.

8. *Command Line Reference Guide*

   A quick reference to the various command-line tools supplied with Artix

---

Designing with Artix

Designing Artix Solutions is divided into two parts:

- The first part explains how to create and manage Artix contracts using the Artix Designer GUI. It also explains how to generate stub and skeleton code for development, and configuration files for deployment.
- The second part explains how to create and manage Artix contracts using the Artix command line tools. It also explains how to generate stubs, skeletons, and configuration files. This book contains detailed descriptions of the Artix WSDL extensions used to define routes, payload formats, and transports. It also provides an overview of WSDL and how it maps to certain programming concepts.

Developing with Artix

The Artix documentation suite includes two main development guides:

- *Developing Artix Applications in C++*
- *Developing Artix Applications in Java*

Both guides describe how to develop clients and servers using the Artix APIs. They provide examples of using Artix advanced functionality such as transactions, using locator services, session management, and dynamic configuration.

Deploying and Managing Artix Solutions

Deploying and Managing Artix Solutions explains how to configure and deploy all aspects of an Artix solution. It describes the Artix configuration file, where to locate the contracts that control your Artix services, and how to run Artix applications. It also explains how to configure and deploy the Artix locator and session manager.

In addition, the *Artix Tivoli Integration Guide and the Artix BMC Integration Guide* explain Artix integration with third party Enterprise Management Systems IBM Tivoli and BMC Patrol.

Lastly, Artix provides the *Artix Security Guide* for security configuration and management.

Latest updates

The latest updates to the Artix 2.1 documentation can be found at: http://www.iona.com/support/docs/artix/2.1/index.xml.

CHAPTER 2

Artix Concepts

This chapter introduces the key concepts used in the Artix product.

In this chapter

This chapter discusses the following topics:

The Elements of Artix

Overview

This section gives a high-level overview of the main components in the Artix product:

Artix components

Artix' unique features are implemented by a number of plug-ins to IONA's ART platform. These plug-ins form the core of Artix, the *Artix Bus*. Applications that make use of Artix connect to the Artix Bus using Artix *Service Access Points* (SAPs). Service Access Points are described by *Artix Contracts*.

Figure 3 shows an example of how all of these Artix elements fit together.

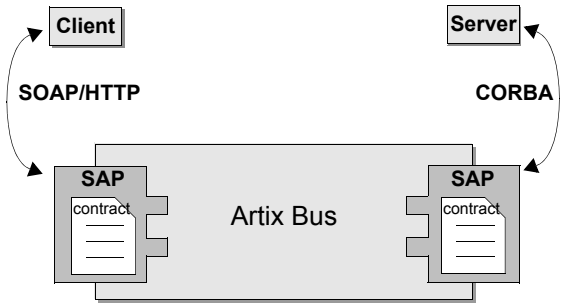


Figure 3: *The Artix Bus*

The rest of this chapter describes each of these components in more detail.

The Artix Bus

Overview

The Artix Bus is a set of plug-ins that work in much the same way as simultaneous translators at the United Nations. Reader plug-ins read data that may be in a number of disparate formats, the Artix Bus directly translates the data into another format, and writer plug-ins write the data back out to the wire in the new format.

In this way, Artix enables all of the applications in your company to communicate over the Web, without needing to understand SOAP or HTTP. It also means that clients can contact Web services without understanding the native language of the server handling requests.

Benefits

While other Web service products provide some ability to expose enterprise applications as Web services, they frequently require a good deal of coding. The Artix Bus eliminates the need to modify your applications or write code by directly translating between the enterprise application's native communication protocol and SOAP over HTTP, which is the prevalent protocol used for Web services.

For example, by deploying an Artix instance with a SOAP over WebSphere MQ Service Access Point and a SOAP over HTTP Service Access Point, you can expose a WebSphere MQ application directly as a Web service. The WebSphere MQ application does not need to be altered or made aware that it was being exposed using SOAP over HTTP.

The Artix Bus translation facility also makes it a powerful integration tool. Unlike traditional EAI products, Artix translates directly between different middlewares, without first translating into a canonical format. This saves processing and increases the speed at which messages are transmitted.

Artix Service Access Points

Overview

An Artix Service Access Point (SAP) is where a service provider or service consumer connects to the Artix Bus. SAPs are described by a contract describing the services offered and the physical representation of the data on the network.

Reconfigurable connection

An Artix SAP provides an abstract connection point between applications, shown in Figure 3 on page 18. The benefit of using this abstract connection is that it allows you to change the underlying communication mechanisms without recoding any of your applications. You simply need to modify the contract describing the SAP.

For example, if one of your back-end service providers is a Tuxedo application and you want to swap it for a CORBA implementation, you simply change the SAP's contract to contain a CORBA connection to the Artix Bus. The clients accessing the back-end service provider do not need to be aware of the change.

Artix Contracts

Overview

Web Services Definition Language (WSDL) is used to describe the characteristics of the Service Access Points (SAPs) of an Artix connection. By defining characteristics such as service operations and messages in an abstract way—independent of the transport or protocol used to implement the SAP—these characteristics can be bound to a variety of protocols and formats. Artix allows an abstract definition to be bound to multiple specific protocols and formats. This means that the same definitions can be reused in multiple implementations of a service.

Artix contracts define the services exposed by a set of systems, the payload formats and transports available to each system, and the rules governing how the systems interact with each other. The most simple Artix contract defines a pair of systems with a shared interface, payload format, and transport. Artix contracts, however, can define very complex integration scenarios.

This section covers the following topics:

WSDL basics

Understanding Artix contracts requires some familiarity with WSDL. The key WSDL terms can be defined as follows:

WSDL types provide data type definitions used to describe messages.

A WSDL message is an abstract definition of the data being communicated and each part of a message is associated with defined types.

A WSDL operation is an abstract definition of the capabilities supported by a service, and is defined in terms of input and output messages.

A WSDL portType is a set of abstract operation descriptions.

A WSDL binding associates a specific data format for operations defined in a `portType`.

A WSDL Port specifies the transport details for a binding, and defines a single communication endpoint.

A WSDL service specifies a set of related ports.

The Artix contract

An Artix contract is specified in WSDL and is conceptually divided into logical and physical components.

The logical contract

The logical contract specifies components that are independent of the underlying transport and wire format. It fully specifies the data structure and the possible operations or interactions with the interface. It enables Artix to generate skeletons and stubs without having to define the physical characteristics of the connection (transport and wire format).

The physical contract

The physical component of an Artix contract defines the format and transport-specific details, for example:

- The wire format, middleware transport, and service groupings.
- The connection between the `portType` operations and wire formats.
- Buffer layout for fixed formats.
- Artix extensions to WSDL.

Table 1: *Artix WSDL Contract Elements*

Logical contract:	
`<schema>`	
`<types>`	(analogous to typedefs)
`<message>`	(analogous to a parameter)
`<portType>`	(analogous to a class or CORBA interface definition)
`<operation>`	(analogous to a method)
Physical contract:	
`<binding>`	(payload format)
`<service>`	(groups of ports)
`<port>`	(transport addressing information)
`<route>`	(rules governing system interaction)

Further information

For a more detailed introduction to the WSDL concepts used in Artix contracts, see "WSDL Concepts" on page 27.

Artix Services

Overview

In addition to the core Artix components, Artix also provides three services:

- "Artix locator"
- "Artix session manager"
- "Artix Transformer" on page 25

These services provide advanced functionality that Artix deployments can use to gain even more flexibility.

Artix locator

The Artix locator provides service look-up and load balancing functionality to an Artix deployment. The locator functionality is provided by two Artix plug-ins:

- The Locator Service plug-in is responsible for collecting service references and passing the references out to client applications.
- The Endpoint Manager plug-in reports service details to the Locator Service plug-in to facilitate load balancing and to ensure that all references are current.

For information about deploying the Artix locator, read *Deploying and Managing Artix Solutions*.

For information about developing client applications that use the locator to look-up services, read *Developing Artix Applications in C++*.

Note: Currently Java clients cannot look up services with the Artix locator.

Artix session manager

The Artix session manager allows you to control the number of concurrent clients allowed to connect to a group of services. It also provides control over the amount of time a client can have access to a service before having to request an extension.

The session manager is split into three plug-ins:

- The Service Manager Service plug-in processes client session requests, passes valid sessions to clients, and processes session renewals.

- The Endpoint Manager plug-in provides service details to the Session Manager Service plug-in and validates client requests based on their session tokens.
- The Simple Policy plug-in defines the duration of each session as well as the number of concurrent sessions permitted in each group.

For information about deploying the Artix session manager, read *Deploying and Managing Artix Solutions*.

For information about developing client applications that use session managed services, read *Developing Artix Applications in C++*.

Note: Currently Java clients cannot use session managed services.

Artix Transformer

The Artix Transformer provides Artix with a way to transform operation parameters on the wire using rules written in Extensible Style Sheet Transformation (XSLT) scripts. The Transformer can be used to provide a simple means of transforming data. For example, it can be used to develop an application that accepts names as a single string and returns them as separate first and last name strings.

The Transformer can also be placed between two applications where it can transform messages as they pass between the applications. This functionality allows you to connect applications that do not use exactly the same interfaces and still realize the benefits of not using a canonical format.

Also, because the transformations are described in XSLT scripts, using the Transformer requires no code changes in either of the applications being integrated. Just as the Artix Bus hides the transport details from the applications, the Artix Transformer hides the interface transformation details from the applications.

For information on deploying the Artix Transformer, read *Deploying and Managing Artix Solutions*.

CHAPTER 3

WSDL Concepts

Artix contracts are WSDL documents that describe logical abstract services and the data they use. This chapter provides a more detailed introduction to WSDL concepts.

In this chapter

This chapter discusses the following topics:

Web Services Description Language Basics

Overview

Web Services Description Language (WSDL) is an XML document format used to describe services offered over the Web. In this respect, it is similar to a CORBA IDL file, an abstract C++ class, or a Java interface definition.

Information within the WSDL file describes the operations offered by the Web service and the location of the Web service. Because the WSDL file is an XML document, it can be validated against an XML schema document to ensure its accuracy.

You do not need to be a WSDL expert to use Artix. However, a basic understanding of WSDL concepts is recommended.

WSDL File Structure

WSDL files include three sections, which collectively define a Web service:

- "Import section"
- "Logical section"
- "Physical section" on page 31

While a complete Web service definition requires content from each of these sections, a specific WSDL file does not need to include all three sections.

Import section

The import section integrates content from other WSDL files into the current WSDL file. Like a CORBA IDL file, you can build a complex WSDL file by simply importing other WSDL files. Inclusion of import elements is optional, but its use greatly facilitates the writing and maintenance of large, or complex, WSDL files.

Logical section

The logical section includes a description of the Web service that is independent of any programming language, marshalling schema or protocol. This section of the WSDL file describes the data types and operations offered by the Web service. It is composed of three subsections:

- Types
- Message
- portTypes

Types

The `types` subsection includes the definitions for specific data types used within an application. It describes how the data is represented within your application's code. Each Web service development tool maps these data type definitions into programming language-specific data types and classes.

This subsection is an XML schema that defines the format of these types. When creating a WSDL file, you can either include the XML schema as part of the file or import an existing schema. By using file imports, you can maintain your type definitions in a separate file that can be used by multiple applications.

The following WSDL file fragment illustrates the contents of the types subsection. There are two data types defined: `InParameter` and `OutParameter`. Both types represent string values. Do not be misled by the names: `InParameter` and `OutParameter`. The types can be used to represent any string value.

```
<types>
 <schema targetNamespace="http://www.iona.com/tutorial"
         xmlns="http://www.w3.org/2001/XMLSchema"
         xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <simpleType name="InParameter">
   <restriction base="xsd:string"/>
  </simpleType>
  <simpleType name="OutParameter">
   <restriction base="xsd:string"/>
  </simpleType>
 </schema>
</types>
```

Messages

The `message` subsection describes how the data are combined to form Web service requests and responses. For example, your messages might specify that a Web service request requires two pieces of data (parts), while the corresponding response includes only a single part.

The following WSDL file fragment illustrates the contents of the message subsection. There are two message definitions. Each message contains a single part. Do not be misled by the names of the messages or parts. Regardless of the assigned name, the messages could be used to represent either a Web service request or response.

```
<message name="RequestMessage">
 <part name="InPart" type="ns1:InParameter"/>
</message>

<message name="ResponseMessage">
 <part name="OutPart" type="ns1:OutParameter"/>
</message>
```

portTypes

The `portType` subsection describes how messages are combined to define the operations available from the Web service, and each `portType` can contain one or more operations. For example, a request/response type operation specifies one input message and one output message. Each Web service development tool maps the `portType` to a class, each operation to a method in the class definition, and each message to either the input or output parameters of a method.

The following WSDL file fragment illustrates the contents of the portType subsection. In Artix, the `portType` name becomes the name of the class that implements the Web service. This class contains the method `sayHi`, whose signature includes an in parameter corresponding to the `input` element and an out parameter corresponding to the `output` element.

```
<portType name="TutorialPortType">
 <operation name="sayHi">
  <input message="ns1:RequestMessage" name="sayHiRequest"/>
  <output message="ns1:ResponseMessage" name="sayHiResponse"/>
 </operation>
</portType>
```

The syntax and format of the logical section is standardized through specifications issued by the World Wide Web Consortium (W3C). All Web service development tools must support these specifications to ensure interoperability between Web services developed with different tools.

Physical section

The physical section includes the data marshalling schema and transport-specific content, and describes the interaction of a Web service application with the runtime environment. The information in this section is specific to your application, and is composed of two subsections:

- binding
- service

The `binding` subsection describes how the data is encoded during transmission, while the `service` subsection provides information specific to the transport protocol.

Binding

For standard SOAP-encoded Web services, there are two formats to the `binding` subsection: rpc/encoded and document/literal. The syntax and contents of both formats are described in W3C specifications. For this reason, the contents of the binding subsection, and its child elements, can be relatively sparse, as each Web service product implements the same specification and the interpretation of the marshalling schema and format can be coded into the product.

Artix supports alternative marshalling schemas and formats for the binding subsection. In WSDL files using these extensions, the contents of the binding subsection is more complex. Artix provides command-line and GUI tools that generate these more complex bindings, so you do not need to hand edit your WSDL files.

The following WSDL file fragment illustrates the contents of the binding and service subsections. In the second and third lines, you can see that the `TutorialPortType_SOAPBinding` describes the marshalling of data for the `TutorialPortType`. Each binding subsection is associated with only one `portType`, although a WSDL file can contain multiple binding elements associated with the same `portType`.

Note that in this example, the binding specifies the rpc/encoded format. Later in this document you will use the Artix Designer to create a WSDL file using the document/literal format.

```
<binding
   name="TutorialPortType_SOAPBinding"
   type="ns1:TutorialPortType">
 <soap:binding
    style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
 <operation name="sayHi">
  <soap:operation soapAction="" style="rpc"/>
  <input name="sayHiRequest">
   <soap:body
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="http://soapinterop.org/" use="encoded"/>
  </input>
  <output name="sayHiResponse">
   <soap:body
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="http://soapinterop.org/" use="encoded"/>
  </output>
 </operation>
</binding>

<service name="HelloWorldService">
 <port
     binding="ns1:TutorialPortType_SOAPBinding"
     name="HelloWorldPort">
  <soap:address location="http://localhost:9000"/>
 </port>
</service>
```

Service

The `service` subsection is associated, through its nested port elements, with the binding `TutorialPortType_SOAPBinding`. Each service element is associated with only one binding.

Although the W3C provides specifications for some binding and service definitions (for example, the Simple Object Access Protocol (SOAP) binding), it is permissible for Web service development tools to define alternative binding and service representations. To support multiple data encoding schemas and transport protocols, Artix extends the W3C specifications.

Namespace Definitions

Overview

Every element in a WSDL file must belong to a namespace. Namespace declarations are scoped. A declaration can exist globally over the entire WSDL document, or locally within an element and its enclosed child elements.

Namespace Prefixes

Namespaces are identified through namespace prefixes, which are generally defined within the opening root element of the WSDL file. Prefixes defined within the root element have global scope and are available throughout the entire WSDL file. However namespaces can be defined, or redefined, within an element. In this case, the scope of the prefix applies only to the element and its child elements.

If an element name is not qualified with a namespace prefix, the element belongs to the default namespace. When writing a WSDL file, you can redefine the default namespace within an element. By redefining the default namespace locally, you can reduce the effort needed to define the child elements contained within this element.

Later in this document you will use the Artix Designer to create a WSDL file. The Artix Designer completely manages the namespace and prefix declarations—you do not need to edit these entries.

The following WSDL file fragment illustrates the contents of the opening root `<definitions>` element. This element contains the global namespace prefixes that are themselves prefixed with `xmlns`, which corresponds to the default namespace.

```
<definitions
  name="HelloWorldTutorial"
  targetNamespace="http://www.iona.com/tutorial"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.iona.com/tutorial"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

Understanding the namespace definition tags

Name

The `name` attribute is an arbitrary, user-defined name assigned to this WSDL file.

TargetNamespace

The `targetNamespace` attribute is an arbitrary, user-defined identifier for the namespace that applies to elements defined within this WSDL file. Although the value of this attribute appears to be an Internet URL, it need not actually represent a Web page. The URL format is used to ensure uniqueness; you can use any unique content for this value. Note that the value of the `targetNamespace` attribute and the value of the `xmlns:tns` namespace prefix are identical. Elements within the WSDL file prefixed with `tns` are also assigned to the target namespace.

XMLNS

The `xmlns` attribute defines the default namespace and corresponds to the schema that defines the structure of a WSDL document. This entry is a valid URL and you can use it to retrieve a copy of the XML schema file that describes the WSDL file contents. Elements within your WSDL file that do not include a namespace prefix become members of this namespace. These elements must be defined in the XML schema file available at http://schemas.xmlsoap.org/wsdl/.

XMLNS:SOAP

The `xmlns:soap` attribute defines the namespace that must be used when adding elements that describe a SOAP binding. Again, this entry is a valid URL and you can use it to retrieve a copy of the XML schema file that describes the SOAP binding. You can see how this namespace prefix is used in the WSDL file fragment described in "Physical section" on page 31.

XMLNS:TNS

The `xmlns:tns` attribute is the namespace prefix for elements defined in this WSDL file document. Its value is assigned by the user and is identical to the value of the `targetNamespace` attribute. You use the `tns` prefix to refer to the original default namespace within elements that have a locally defined target namespace.

XMLNS:WSDL

The `xmlns:wsdl` attribute also defines the default namespace; it is the same value as the `xmlns` attribute. You use this prefix within an element where you have redefined the default namespace.

XMLNS:XSD

Lastly, the `xmlns:xsd` attribute defines the namespace for the basic XML types. This too is a valid URL that you can use to obtain further information about the XML basic types.

A Complete WSDL File

The following WSDL file describes a simple HelloWorld Web service. In the earlier sections of this chapter, you reviewed the contents of this file.

```
<?xml version="1.0" encoding="UTF-8"?>

<definitions
  name="HelloWorldTutorial"
  targetNamespace="http://www.iona.com/tutorial"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.iona.com/tutorial"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<types>
 <schema targetNamespace="http://www.iona.com/tutorial"
         xmlns="http://www.w3.org/2001/XMLSchema"
         xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  <simpleType name="InParameter">
   <restriction base="xsd:string"/>
  </simpleType>
  <simpleType name="OutParameter">
   <restriction base="xsd:string"/>
  </simpleType>
 </schema>
</types>

<message name="RequestMessage">
 <part name="InPart" type="tns:InParameter"/>
</message>

<message name="ResponseMessage">
 <part name="OutPart" type="tns:OutParameter"/>
</message>
```

```
<portType name="TutorialPortType">
 <operation name="sayHi">
  <input message="tns:RequestMessage" name="sayHiRequest"/>
  <output message="tns:ResponseMessage" name="sayHiResponse"/>
 </operation>
</portType>
<binding
   name="TutorialPortType_SOAPBinding"
   type="tns:TutorialPortType">
 <soap:binding
    style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http"/>
 <operation name="sayHi">
  <soap:operation soapAction="" style="rpc"/>
  <input name="sayHiRequest">
   <soap:body
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="http://soapinterop.org/" use="encoded"/>
  </input>
  <output name="sayHiResponse">
   <soap:body
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="http://soapinterop.org/" use="encoded"/>
  </output>
 </operation>
</binding>
<service name="HelloWorldService">
 <port
     binding="tns:TutorialPortType_SOAPBinding"
     name="HelloWorldPort">
  <soap:address location="http://localhost:9000"/>
 </port>
</service>
</definitions>
```

You will use this file in the following chapter to create a Web service application.

CHAPTER 4

Coding the Web Service

You can use a WSDL file to generate code and develop a Web service application. The discussion in this chapter illustrates how to use the Artix wsdltocpp and wsdltojava utilities to generate C++ and Java code from a WSDL file.

In this Chapter

This chapter discusses the following topics:

The wsdltocpp Utility

Overview

Once you have a WSDL file, whether you write it yourself or obtain it from another source, you can write an application. With Artix, you can write a client application against an existing Web service, you can write a server application that implements the Web service, or you can write both the client and server applications. The `wsdltocpp` utility is a command-line tool that you will use to generate C++ code from a WSDL file.

This section discusses the following topics:

Command-line options

You control the output of the `wsdltocpp` command-line utility through command-line options. By specifying the appropriate options, you can generate exactly the code you need. The syntax used to invoke the `wsdltocpp` utility is:

```
wsdltocpp [options] {WSDL-URL}
```

Where `{WSDL-URL}` is the path, or Web location, of the WSDL file, and `options` can be:

```
-e Web-service-name
   The value of the name attribute in the <service> element. If
   the WSDL file includes multiple <service> elements, and the -e
   option is not specified, the value defaults to the name of the
   first <service> element in the WSDL file.
```

```
-t port
   The value of the name attribute in the <port> element. If a
   <service> element contains multiple <port> elements, and the
   -t option is not specified, the value defaults to the name of
   the first <port> element. If neither the -e nor -t option is
   specified, the first <port> element within the first
   <service> element in the WSDL file is used for code
   generation.
-n namespace
   The C++ namespace for the generated code.
   Defaults to the global namespace.
-impl
   Whether to generate starting point code for the C++ class into
   which you will code the Web service implementation.
-m { NMAKE | UNIX }
   Whether to generate a makefile for the selected platform.
   Choose either NMAKE for Windows or UNIX for Unix.
-server
   Whether to generate server stub classes only.
-client
   Whether to generate client proxy classes only.
-sample
   Whether to generate starting point code for client and/or
   server mainline applications. Works in conjunction with the
   -server and -client options.
```

Each of these options is explained in more details in the subsections that follow.

There are other options in addition to those described. These additional options are not, however, commonly used and are not discussed in this book. Refer to *Designing Artix Solutions* for a complete discussion of the command-line options.

Specifying the service/port

If your WSDL file includes multiple `<service>` elements, or multiple `<port>` elements within a single `<service>` element, you need to specify which `<service>` and/or `<port>` should be referenced during the code generation process.

You use the `-e` and `-t` command-line options to specify these values.

If the WSDL file contains multiple service/port definitions and you do not use the `-e` and/or `-t` options, code will be generated for the first `<service>` and `<port>` defined in the WSDL file.

For the simple `HelloWorldTutorial.wsdl` file documented in "WSDL Concepts" on page 27, there is only a single `<service>` element containing a single `<port>` element. Consequently, you will not need to use these options when generating code from this WSDL file.

Specifying the C++ namespace

Generated C++ code should be included within a C++ namespace. You use the `-n` option to provide the name of the namespace. Use of this option is not mandatory, but it is good programming practice to generate code within a namespace.

Generating the implementation class

The `wsdltocpp` utility generates starting point code for the Web service implementation class when you supply the `-impl` option. There is no way to specify names for the generated files; the names of the generated files are derived from either the `portType` name or the name of the WSDL file. Therefore, if you use the `-impl` option multiple times, the starting point code is regenerated, deleting any code you have added to an earlier version of the generated files.

Generating application-specific code

You can control whether code is generated only for client applications, only for server applications, or for both types of application. Use the `-client` option to restrict code generation to client-related files; use the `-server` option to restrict code generation to server-related files.

The `-sample` option indicates whether starting point client and/or server mainline code should be generated. You must be careful not to overwrite the client mainline code once you have begun coding your application. If you need to rerun the `wsdltocpp` utility, be certain not to use the `-sample` option.

Generating the makefile

The `wsdltocpp` utility can create a makefile. Use the `-m {NMAKE | UNIX}` option to create a makefile that is complete for the type of application you are creating. That is, if you are only building a client application, the makefile will not include any references to files and classes specific to server applications.

The wsdltojava Utility

Overview

The `wsdltojava` utility is a command-line tool that you can use to generate Java code from a WSDL file. This section discuss the utility, within these topics:

Command-line options

You control the output of the `wsdltojava` command-line utility through command-line options. By specifying the appropriate options, you can generate exactly the code you need. The syntax used to invoke the `wsdltojava` utility is:

```
wsdltojava [options] {WSDL-URL}
```

Where `{WSDL-URL}` is the path, or Web location, of the WSDL file, and `options` can be:

```
-e Web-service-name
   The value of the name attribute in the <service> element. If
   the WSDL file includes multiple <service> elements, and the -e
   option is not specified, the value defaults to the name of the
   first <service> element in the WSDL file.
-t port
   The value of the name attribute in the <port> element. If a
   <service> element contains multiple <port> elements, and the
   -t option is not specified, the value defaults to the name of
   the first <port> element. If neither the -e nor -t option is
   specified, the first <port> element within the first
   <service> element in the WSDL file is used for code
   generation.
-p package
   The package name for the generated code. If omitted, the
   utility creates a package name from the value of the
   targetNamespace attribute in the WSDL file. If you do not want
   a package, enter -p "".
```

```
-impl
   Whether to generate starting point code for the Java class
   into which you will code the Web service implementation.
-server
   Whether to only generate code for the server stub classes and
   a server mainline.
-client
   Whether to generate client proxy classes only.
-sample
   Whether to generate starting point code for the client
   mainline application.
-ant
   New
```

Each of these options is explained in more detail in the subsections that follow.

There are other options in addition to those described. These additional options are not, however, commonly used and are not be discussed in this book. Refer to *Designing Artix Solutions* for a complete discussion of the command-line options.

Specifying the service/port

If your WSDL file includes multiple `<service>` elements, or multiple `<port>` elements within a single `<service>` element, you need to specify what `<service>` and/or `<port>` should be referenced during the code generation process. You use the `-e` and `-t` command-line options to specify these values. If the WSDL file contains multiple service/port definitions and you do not use the `-e` and/or `-t` options, code is generated for the first service and port defined in the WSDL file.

For the simple `HelloWorldTutorial.wsdl` file developed in "WSDL Concepts" on page 27, there is only a single `<service>` element containing a single `<port>` element. Consequently, you do not need to use these options when generating code from this WSDL file.

Specifying the package

Generated Java code should be included within a package. You use the `-p` option to provide the package name.

Generating the implementation class

The `wsdltojava` utility generates starting point code for the Web service implementation class when you supply the `-impl` option. There is no way to specify names for the generated files; the names of the generated files are derived from either the `portType` name or the name of the WSDL file.

Therefore, if you use the `-impl` option multiple times, the starting point code is regenerated, deleting any code you have added to an earlier version of the generated files.

Generating application-specific code

You can control whether code is generated only for client applications, only for server applications, or for both types of application. Use the `-client` option to restrict code generation to client-related files; use the `-server` option to restrict code generation to server-related files. When you supply the `-server` option, the `wsdltojava` utility generates both the server stubs and mainline code.

The `-sample` option indicates whether starting point client mainline code should be generated. You must be careful not to overwrite mainline code once you have begun coding your application. If you need to rerun the `wsdltojava` utility, do not use the `-sample` option.

Generating Code

Overview

This section discusses how to generate the code for your applications, and contains the following topics:

Configuring Artix

During the installation process, Artix creates two configuration files. The file `<installationDirectory>\artix\2.1\etc\domains\artix.cfg` is the main configuration file. During start-up, every Artix process reads this file. For the purposes of the exercises in this book you do not need to change the contents of this file. As you develop more complex applications, you might need to extend and/or edit this file.

The file `<installationDirectory>\artix\2.1\bin\artix_env.bat` is used to set the Artix development and runtime environments. You must run this file in every command window before building or running an Artix application.

You must also be certain that your C++ compiler and libraries are available to your applications. With Windows, you might need to run the `vcvars32.bat` file to properly set your environment.

Note: You might find it convenient to place a call to the `vcvars32.bat` file into the `artix_env` script file. To do this, open the `artix_env` script file in a text editor and place the following entry at the beginning of the file:

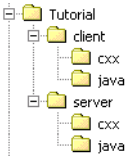
```
call "C:\Program Files\Microsoft Visual
   Studio\VC98\bin\cvars32.bat";
```

This syntax assumes that your Visual Studio installation is in the default location. The quotation marks and semicolon are required.

Creating the directory structure

A number of demo applications are available in the `<installationDirectory>\artix\2.1\demos` directory. After you complete this the examples in this book you should review each of these demos.

For the examples in this book, create a subdirectory called `Tutorial` under the `Demos` directory. Under the `Tutorial` directory, create the subdirectories `client` and `server`, and within the `client` and `server` subdirectories, create the subdirectories `cxx` and `java`.



Copying the WSDL file

Copy the contents of the `HelloWorldTutorial.wsdl`, from "A Complete WSDL File" on page 36, into a text document.

Note: In the PDF version of this tutorial, the file content spans two pages. You need to copy and paste the contents from the first page, and then copy and paste the contents from the second page.

Save the file as `HelloWorldTutorial.wsdl` into the `Tutorial` directory. To view the contents with formatting, open the file in your Internet Explorer browser.

Generating the Client Application Code

Overview

This subsection discusses how to generate the client application code, and contains the following topics:

Generating the C++ client application

To generate the C++ client application, complete the following steps:

1. Open a command window to the `<installationDirectory>\artix\2.1\bin` directory and run the `artix_env.bat` file.
2. Move to the following directory:
   `<installationDirectory>\artix\2.1\demos\Tutorial\client`
3. Generate the C++ client application with the command:

```
wsdltocpp -n ArtixDemo -client -sample -m NMAKE
   HelloWorldTutorial.wsdl
```

Generated files

The following files are generated within the `client\cxx` subdirectory:

- `Tutorial.h`: A header file that defines the method signatures for the Web service.
- `TutorialClient.h`, `TutorialClient.cxx`: Header and implementation files that define the client proxy class. This client proxy class implements the virtual `sayHi` method. You do not need to edit the code in these files, but you should review the contents of the header file. Note that there are multiple constructors defined. In this example, your code uses the first constructor, which does not require any input from your code. As you develop more complex applications you will learn the value of the alternative constructors. The alternative constructors are not discussed in this book.
- `TutorialClientSample.cxx`: The starting point code for your client application. In this example, you need to add code to this file.

- `HelloWorldTutorial_wsdlTypes.h`, `HelloWorldTutorial_wsdlTypes.cxx`: Header and implementation files that include the definitions for the classes that represent the data types defined in the WSDL file `<types>` section.
- You must review the contents of the header file, from which you learn the APIs needed to work with the generated type definitions.
- `HelloWorldTutorial_wsdlTypesFactory.h`, `HelloWorldTutorial_wsdlTypesFactory.cxx`: Header and implementation files for factory classes required if you have defined an `anyType` in your WSDL file.
- You do not need to review the contents of these files.
- `makefile`: A makefile that you can use to build the client application.

Generating the Java client application

To generate the Java client application, complete the following steps:

1. Open a command window to the `<installationDirectory>\artix\2.1\bin` directory and run the `artix_env.bat` file.
2. Move to the following directory: `<installationDirectory>\artix\2.1\demos\Tutorial\client\java`
3. Generate the Java client application with the command:

```
wsdltojava -p ArtixDemo -client -sample -ant
   ../../HelloWorldTutorial.wsdl
```

Generated files

The `wsdltojava` utility creates the subdirectory `ArtixDemo`, into which the following files are generated:

- `Tutorial.java`: An interface that defines the method signatures for the Web service.
- `TutorialTypeFactory.java`: Definition of a class that creates and manages `anyTypes` defined in your WSDL file.
- `TutorialDemo.java`: Starting point code for a client mainline application. For this simple example, the generated code in this file represents a fully functional application. With a more complex application, you can use this code as a template.

The `wsdltojava` utility also creates the subdirectory `client\java`, into which the following file is generated:

- `build.xml`: an Apache ant-based make file containing the information needed to build the application

Generating the Server Application Code

Overview

This subsection discusses how to generate the server application code, and contains the following topics:

Generate the C++ server application

To generate code for the C++ server application, complete the following steps:

1. Move to the `Tutorial\server\cxx` directory.
2. Generate the C++ server application with the command:

```
wsdltocpp -n ArtixDemo -server -sample -m NMAKE -impl
   ../HelloWorldTutorial.wsdl
```

Generated files

The following files are generated into the `server\cxx` subdirectory:

- `Tutorial.h`: A header file that defines the method signatures for the Web service.
- `TutorialServer.h`, `TutorialServer.cxx`: Header and implementation files that define the server stub class.
- You do not need to review the contents of these files.
- `TutorialServerSample.cxx`: The starting point code for your server mainline application. In this example, you do not need to add code to this file. In more complex applications, you might need to extend the generated code, for example, by using the servant management classes.
- `TutorialImpl.h`, `TutorialImpl.cxx`: Header and implementation files that contain starting point code for your Web service implementation class. For this example you need to add coding to the method bodies corresponding to the Web service operations. In more complex

applications, you might need to edit the header file as well as add code to the implementation file, for example, by overriding the activated method inherited from the implementation class' superclass.

- `HelloWorldTutorial_wsdlTypes.h`, `HelloWorldTutorial_wsdlTypes.cxx`: Header and implementation files that include definitions for the classes representing the data types defined in the WSDL `<types>` section.

  You must review the contents of the header file, from which you learn the APIs needed to work with the generated type definitions.
- `HelloWorldTutorial_wsdlTypesFactory.h`, `HelloWorldTutorial_wsdlTypesFactory.cxx`: Header and implementation files for factory classes that create instances of your application-specific data types.

  You do not need to review the contents of these files.
- `makefile`: A makefile that you can use the build the server application.

Generating the Java server application

To generate code for the Java server application, complete the following steps:

1. Move to the `Tutorial\server` directory.
2. Generate the Java server application with the command:

```
wsdltojava -p ArtixDemo  -server -impl -ant
   ../../HelloWorldTutorial.wsdl
```

Generated files

The `wsdltojava` utility creates the subdirectory `ArtixDemo`, into which the following files are generated:

- `Tutorial.java`: An interface that defines the method signatures for the Web service.
- `TutorialTypeFactory.java`: Definition of a class that creates and manages `anyTypes` defined in your WSDL file.
- `TutorialImpl.java`: Starting point code for your Web service implementation class. In this example you need to add code to the methods corresponding to the Web service operations.

- `TutorialServer.java`: Starting point code for a server mainline application. For this simple example, the generated code in this file represents a fully functional application. With a more complex application, you might extend the generated code.

The `wsdltojava` utility also creates the subdirectory `client\java`, into which the following file is generated:

- `build.xml`: an Apache ant-based make file containing the information needed to build the application.

Adding Processing Logic to the Coding

Overview

The files describing the C++ or Java implementation classes contain compilable code, but there is no processing logic in the method bodies. In this demo application, you need to add processing logic to the implementation class' `sayHi` method.

This section discusses how to add the processing logic, and contains the following topics:

The C++ implementation class

In a text editor, open the `TutorialImpl.cxx` file and note the signature for the `sayHi` method:

```
void
TutorialImpl::sayHi(
    const ArtixDemo::InParameter & InPart,
    ArtixDemo::OutParameter & OutPart
) IT_THROW_DECL((IT_Bus::Exception))
{
}
```

The method includes two parameters: the first representing the part within the input message, and the second representing the part within the output message. The return type is void.

All C++ methods in Artix have void return types and output is always represented by out parameters. At first this may seem to be an inconvenience, but when you consider that input and output messages could include multiple parts and that WSDL has no concept of a return value, this approach makes sense. InParameter correspond to the parts of the input message and OutParameters correspond to the parts of the output message. Since an output message does not assign greater importance to one of its possible multiple parts, it would be impossible for the code generating logic to select which part should correspond to a return value.

To add processing logic to the `sayHi` method, return a message that includes the input. For example, Hello Artix User, where Artix User corresponds to the value of InPart.

You can assume that as the `InParameter` and the `OutParameter` both correspond to an `xsd:string` you can concatenate `"Hello "` with `InPart` and assign the new string to `OutPart`. This is, however, incorrect.

If you examine the definitions for the `InParameter` and `OutParameter` in the `HelloWorldTutorial_wsldTypes.h` file (shown below), you will notice that they are not strings, but classes that encapsulate a member variable that is a string. To get and set the value of this member variable, you must use the accessor methods.

```
namespace ArtixDemo
{
    . . .

    class InParameter : public IT_Bus::AnySimpleType
    {
      public:
        InParameter();
        InParameter(const InParameter & value);
        . . .

        void setvalue(const IT_Bus::String & value);
        const IT_Bus::String & getvalue() const;

      private:
        IT_Bus::String m_val;

    };
    . . .

    class OutParameter : public IT_Bus::AnySimpleType
    {
      public:
        OutParameter();
        OutParameter(const OutParameter & value);
        . . .

        void setvalue(const IT_Bus::String & value);
        const IT_Bus::String & getvalue() const;

      private:
        IT_Bus::String m_val;

    };
    . . .
};
```

Consequently, the code you add to the `sayHi` method body is:

```
OutPart.setvalue("Hello " + InPart.getvalue());
```

The Java implementation class

In a text editor, open the `TutorialImpl.java` file. Add the following code to the `sayHi` method body:

```
return "Hello " + inPart;
```

Building the Application

Overview

You are now ready to build your applications.

This section walks you through this process, within the following topics:

Building the Server Application

Overview

This subsection contains the following topics:

Building the C++ server application

To build the C++ Server application, complete the following steps:

1. Run the `artix_env.bat` file at the following directory:
   `<installationDirectory>\artix\2.1\bin`
2. Move to the `<installationDirectory>\artix\2.1\demos\Tutorial\server\cxx` directory.
3. Build the server application with the command:

```
nmake all
```

This creates the server executable file `server.exe`.

Building the Java server application

To build the Java Server application, complete the following steps:

1. Move to the `<installationDirectory>\artix\2.1\demos\Tutorial\server\java` directory.
2. Build the server application with the command:

```
ant (or ant build)
```

This creates the server file `TutorialServer.class`.

The C++ client application

For this example, you need to work with the `TutorialClientSample.cxx` file. Although this file is compilable, it does not actually invoke the Web service operations. Open the file and examine the generated code:

```
#include <it_bus/bus.h>
#include <it_bus/Exception.h>
#include <it_cal/iostream.h>

#include "TutorialClient.h"

IT_USING_NAMESPACE_STD
using namespace ArtixDemo
using namespace IT_Bus;

int
main(int argc, char* argv[])
{
    cout << "Tutorial Client" << endl;

    try
    {
        IT_Bus::init(argc, argv);
        TutorialClient client;

        // Sample invocation calls are shown in
        // commented lines below.
        /*
         ArtixDemo::InParameter   InPart;
         ArtixDemo::OutParameter   OutPart;
         client.sayHi ( InPart,  OutPart);
         */
    }
    catch(IT_Bus::Exception& e)
    {
        cout << endl << "Error : Unexpected error occured!"
             << endl << e.Message()
             << endl;
        return -1;
    }
    return 0;
}
```

Note that the code generation process produced a simple invocation of the `sayHi` method, but the code is commented out and there is no value assigned to the in parameter, and no output statement to display the value returned in the out parameter.

You need to remove the comment delimiters and edit the code as follows:

```
ArtixDemo::InParameter    InPart;
ArtixDemo::OutParameter   OutPart;

InPart.setvalue("Artix User");

client.sayHi ( InPart,  OutPart);

cout << "sayHi returned: " + OutPart.getvalue() << endl;
```

The Java client application

For this example, the generated code is a complete running application. To run the demo, there is no need to modify the generated code.

Building the Client Application

Overview

Now that you have completed coding the client mainline, you can build the application. This subsection contains the following topics:

- "Building the C++ client application"
- "Building the Java client application"

Building the C++ client application

To build the C++ Client application. complete the following steps:

1. Run the `artix_env.bat` file at the following directory: `<installationDirectory>\artix\2.1\bin`.
2. Move to the `<installationDirectory>\artix\2.1\demos\Tutorial\client\cxx` directory.
3. Build the client application with the command:

```
nmake all
```

This creates the client executable file `client.exe`.

Building the Java client application

To build the Java Client application, complete the following steps:

1. Move to the `<installationDirectory>\artix\2.1\demos\Tutorial\java` directory.
2. Build the client application with the command:

```
ant (or ant build)
```

This creates the client file `TutorialDemo.class`.

Running the Application

In this section

This section contains the following topics:

Set the runtime environment

Before you can run the application, you must set the environment. To set the Artix environment, complete the following steps:

1. Run the `artix_env.bat` file at the following directory:
   `<installationDirectory>\artix\2.1\bin`
2. Alter the CLASSPATH to include the current directory, by running the following command:

```
set CLASSPATH=.;%CLASSPATH%
```

The C++ Application

Overview

You can now start the C++ server application and then run the C++ client application.

Start the C++ server application

To start the C++ server application:

1. Move to the `<installationDirectory>\artix\2.1\demos\Tutorial\server\cxx` directory.
2. Start the server application with the command:

```
start tutorialserver.exe
```

A new command window opens and the server application starts.

Run the C++ client application

To run the C++ client application:

1. Move to the `<installationDirectory>\artix\2.1\demos\Tutorial\client\cxx` directory.
2. Run the client application with the command:

```
tutorialclient.exe
```

The client application invokes on the Web service and displays the return.

Stop the C++ server application

To stop the C++ Server application, issue `Ctrl-C` in the command window running the server application.

The Java Application

Overview

You can now start the Java server application and then run the Java client application.

Start the Java server application

To start the Java server application:

1. Move to the `<installationDirectory>\artix\2.1\demos\Tutorial\server\java` directory.
2. Start the server application with the command:

```
start ant run.TutorialServer
```

A new command window opens and the server application starts.

Run the Java client application

To run the Java client application:

1. Move to the `<installationDirectory>\artix\2.1\demos\Tutorial\client\java` directory.
2. Run the client application with the command:

```
java ArtixDemo.TutorialDemo sayHi <your name>
```

The client application invokes on the Web service and displays the return.

Stop the Java server application

To stop the Java Server application, issue `Ctrl-C` in the command window running the server application.

Interoperability Between the C++ and Java Applications

To demonstrate that the C++ and Java Web service applications and clients are interoperable, you can run the Java client application against the C++ server application, and vice versa.

Part II

Using Artix

In this part

This part contains the following chapters:

CHAPTER 5

Using the Artix Designer

This chapter introduces the Artix Designer, and outlines how you can use it to build a WSDL file and to generate starting point code.

In This Chapter

This chapter discusses the following topics:

Introduction

In this section

This section discusses the following topics:

Overview

The Artix Designer is a graphical user interface (GUI) application through which you can write and edit WSDL files. Although there are other XML editing tools that you can use to write a WSDL file, the Artix Designer has an understanding of the Artix WSDL extensions and is a much easier way to write the WSDL files used in an Artix application. For example, the Artix Designer automatically adds the required namespace declarations and prefix definitions when you build Artix applications that involve other data marshalling schemas, transport protocols, or routing.

The Artix Designer is also integrated with the Artix command-line tools, such as the `wsdltocpp` utility, so that you can also use it to generate starting point code. Integration with other command-line utilities allows the Artix Designer to import IDL files and convert their contents into WSDL, generate starting point code for Java Web service applications, or convert WSDL files into IDL.

Starting the Artix Designer

In Windows you can start the Artix Designer in any of the following ways.

- Select **Start | Programs | IONA Artix 2.1 | Designer**.
- Double-click on either an `.iad` file or a `.wsdl` file
- From the `<installationDirectory>\artix\2.1\bin` directory, run the batch file `start_designer.bat`.
  Selecting the menu entry simply runs the `start_designer.bat` file.

Welcome dialog

The first time you start the Artix Designer, the Welcome dialog is displayed, which offers the following options:

- Create a workspace (the entity that structures your solution)
- Open an existing workspace or resource file
- Go straight to the Designer
- Run an interactive demo (an overview of the Designer's features)

To stop this dialog displaying every time you start Artix, check the box provided. Once you have done this, Artix will open with your most recently used workspace or WSDL file loaded.

The Artix workspace

An Artix workspace defines the structure of your Artix solution, and includes all your WSDL contracts. The Artix Designer provides a range of wizard templates to help you get started.

For example, you can use templates to create a C++ Web services client, or a C++ Web services client and server, or you can create a new workspace using the New Workspace wizard. Figure 4 shows the new workspace wizard selected in the New Workspace dialog.

It is not necessary to create all of the application files using the Artix Designer. For example, one approach is to import an existing WSDL file into the Designer and then edit the file as required. Alternatively, you can import a CORBA IDL file into the Designer, and the Designer transforms the contents of the IDL file into the equivalent WSDL file.

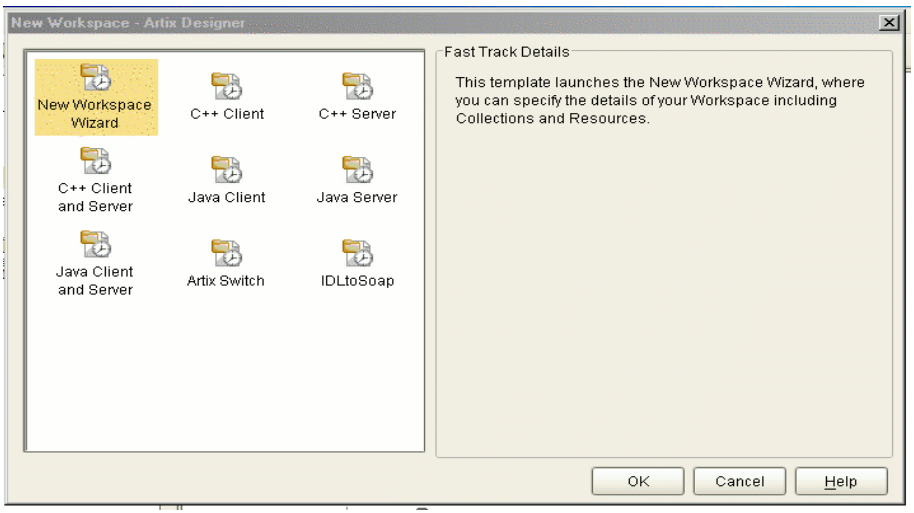


Figure 4: *New Workspace dialog*

Designer Tree

The Designer Tree is a navigation tree displayed on the left side of the Artix Designer. The Designer Tree displays the following information, in the order shown:

Workspace

An Artix Workspace includes all the WSDL contracts in your Artix solution.

The Designer Tree displays the workspace name (for example, `HelloWorld`), and contains its profiles, collections, and shared resources.

Workspace Services	Lists the details of the Artix services, and their status for this workspace. The services available are: - Locator - Session Manager - Security - Management
Shared Resources	These are all the WSDL contracts that you want to work with. Shared resources are also displayed within collections, by italicized text and a dimmed icon. If you click on a shared resource, the pane on the right of the screen displays the WSDL view of that resource.
Deployment Profiles	Deployment Profiles define machine-level information such as the Artix save location, the compiler location, and the operating system being used. If you click on a Deployment Profile, the pane on the right of the screen displays the details for that profile.
Collections	These are groups of WSDL contracts that are organized into logical collections for deployment purposes. A collection maps to an executable or process that implements the WSDL defined in it. You can drag and drop resources between collections and also from the shared resource folder to a collection. If you click on a collection, the pane on the right of the screen displays the details of that collection.

Deployment Bundles The Deployment Bundle defines the deployment characteristics for a collection, such as the deployment type (client, server, or switch), code generation options, and configuration details. You can also modify the service WSDL for each deployment bundle, if necessary.

If you click on a Deployment Bundle, the pane on the right of the screen displays the details for that bundle.

Figure 5 shows a simple example of a workspace named `GoogleSearch` displayed in the Designer Tree.

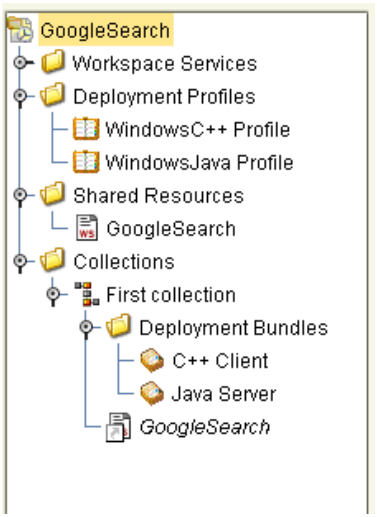


Figure 5: *Designer Tree*

Artix Designer main window

Figure 6 displays the Artix Designer main window. The right-hand pane displays summary information for items displayed in the Designer Tree. For example, clicking on your workspace folder in the tree displays the **Workspace Details**, shown in Figure 6.

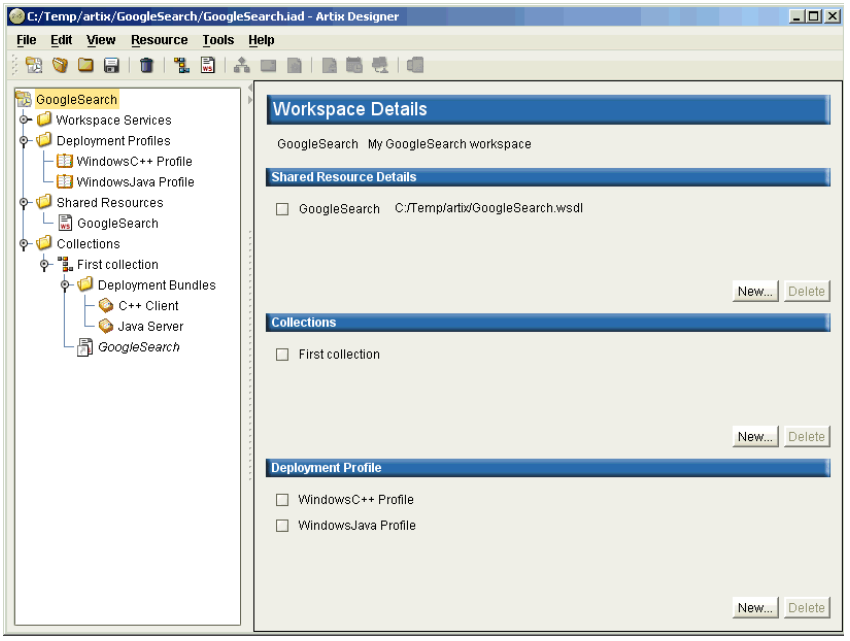


Figure 6: *Artix Designer Main Window*

Similarly, clicking on the **Shared Resources, Collections, Deployment Profiles, or Deployment Bundles** folder displays summary information for these items in right-hand pane.

The Resource Navigator

The Resource Navigator is the engine room of the Artix Designer. It has two main purposes—firstly, it provides a way for you to navigate around the various components of your resource. Secondly, it provides you with editing tools to add or update components in your resources.

Clicking on resource in the Designer Tree displays a graphical view of the resource in the right-hand pane. This is the Resource Navigator **diagram** view, shown in Figure 7.

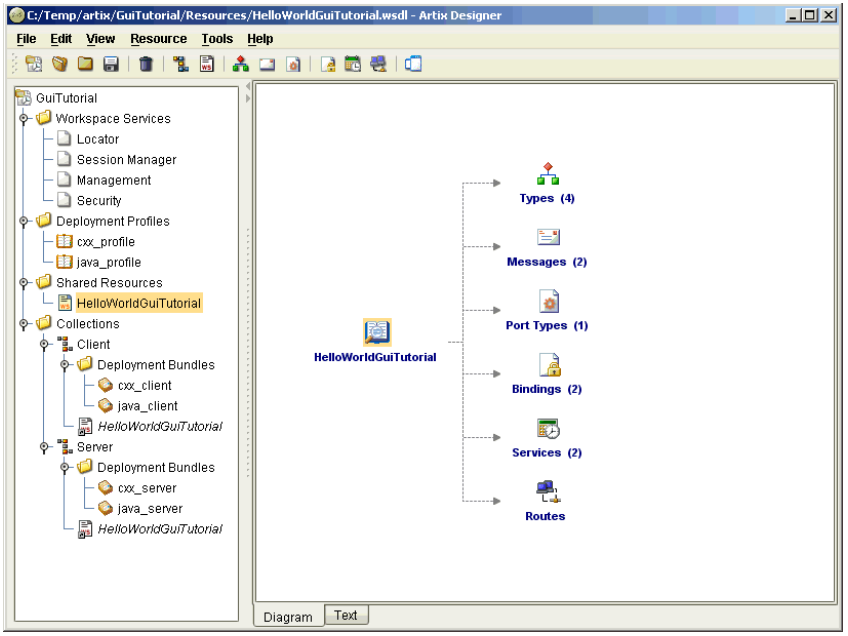


Figure 7: *Resource Navigator—Diagram View*

The icons representing the resource elements (types, messages, services, and so on) in the graphical view can have a small plus sign attached. This indicates that the element has children.

You can view children (types, messages, and so on) by double clicking on the element icon. You can then view or edit the individual items directly from the Resource Navigator. Each child component has an associated dialog to enable viewing or editing. For example, double clicking on a type launches the **Edit Type Attributes** dialog, shown in Figure 8.

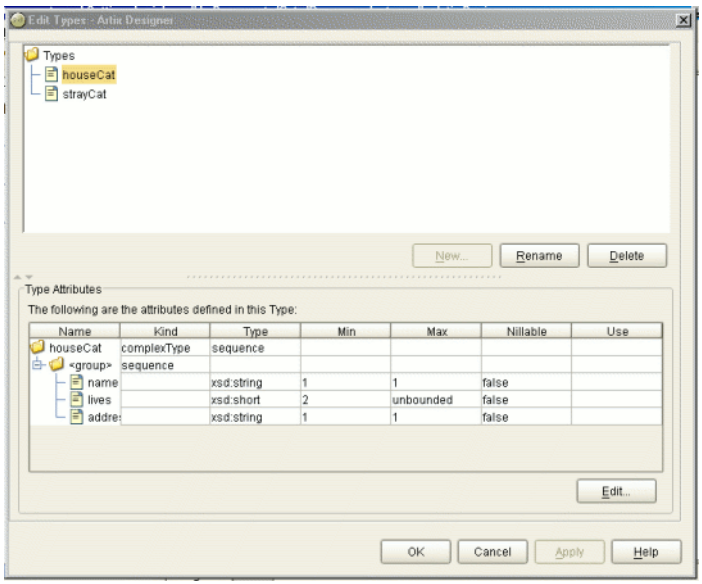


Figure 8: *Edit Type Attributes Dialog*

Working with WSDL

The Resource Navigator also enables you to view and edit the resource's XML directly instead of working through the graphical representation as previously described.

To access the text view of the resource, shown in Figure 9, click on the **Text** tab at the bottom of the Resource Navigator pane.

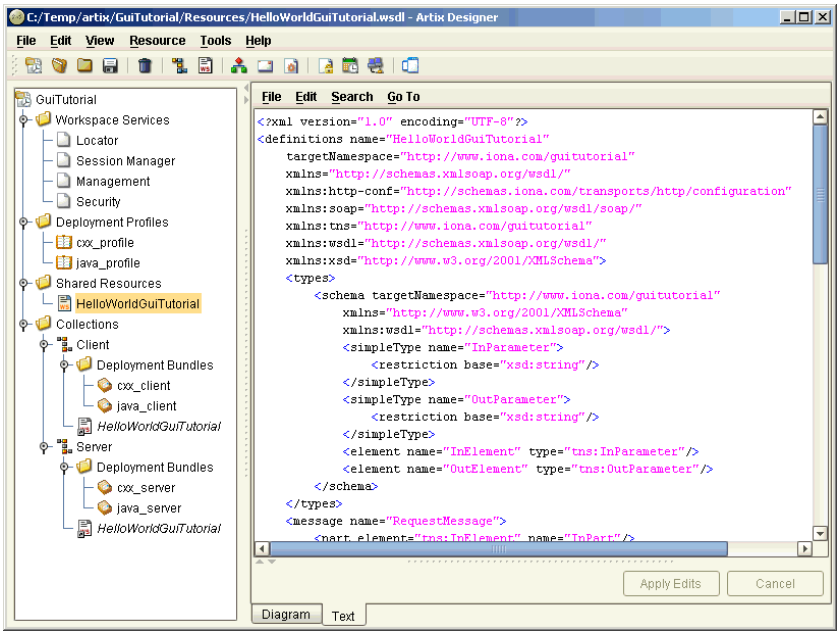


Figure 9: *Resource Navigator—Text View*

Working in the Text view of the resource requires a sound knowledge of WSDL and or XSD. Be aware that if you make changes to the text, it could easily invalidate your resource.

If you do make a change to the text that causes a problem, errors are identified in a separate **ERRORS** panel directly under the text. This enables you to easily identify the exact position of the problem within the file.

Creating a New Workspace

To create a new workspace, complete the following steps:

1. After starting the Artix Designer, you are presented with the Welcome window. Select **Create a new workspace** and click **OK** to display the New Workspace window, as shown in Figure 10.

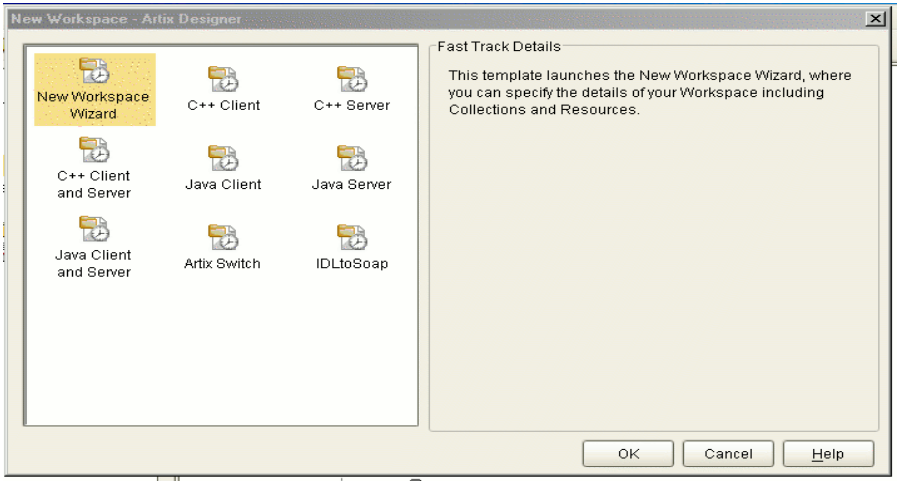


Figure 10: *New Workspace dialog*

2. Select **New Workspace Wizard** icon and click **OK** to display the New Workspace wizard.
3. In the Define Workspace panel, name your project **`GuiTutorial`**, and enter, or browse to, the directory that will contain your project.

   Leave the Add Shared Resources check box unchecked—you are using the Artix Designer to write a new WSDL file.

4. Click **Next** to display the Show Summary panel, as shown in Figure 11.

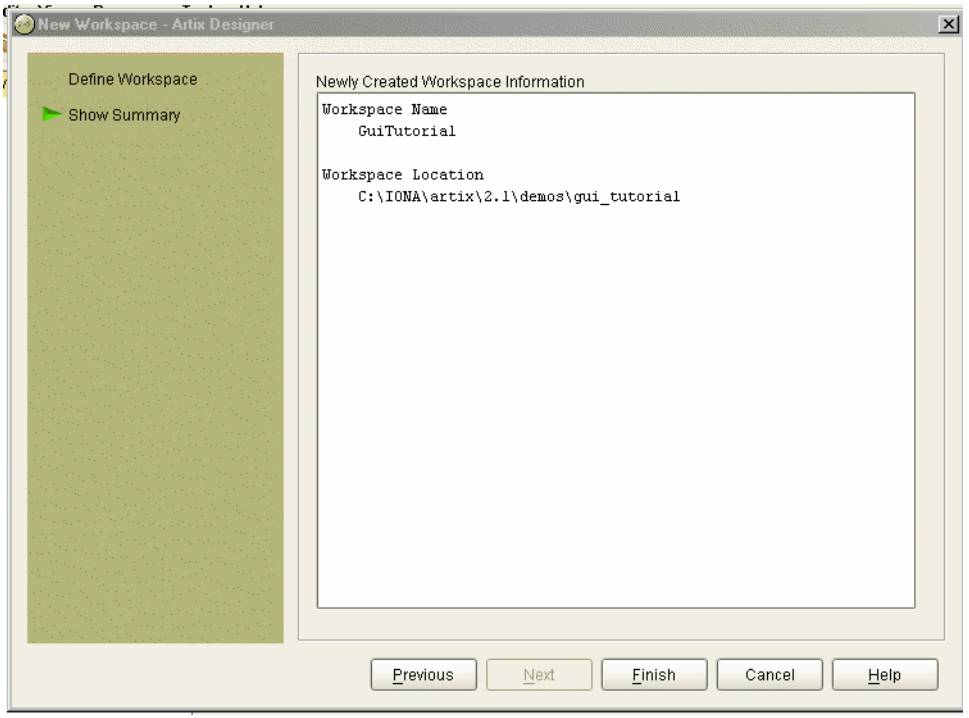


Figure 11: *New Workspace wizard—Summary panel*

5. Click **Finish** to close this wizard and return to the Artix Designer.

   The Artix Designer has created the workspace, and displays it in the Designer Tree with a folder for shared resources, deployment profiles, and collections (processes). Save the workspace to create the necessary directories on your drive.

Creating the WSDL File

Overview

The next task is to create a WSDL file, or Artix contract. This is stored on the tree as an entry under the Shared Resources icon.

Procedure

To create the WSDL file, complete the following steps:

1. Right click on the Shared Resources folder and select **New Resource** (or select **File | New Resource**), to display the New Resource dialog as shown in Figure 12.

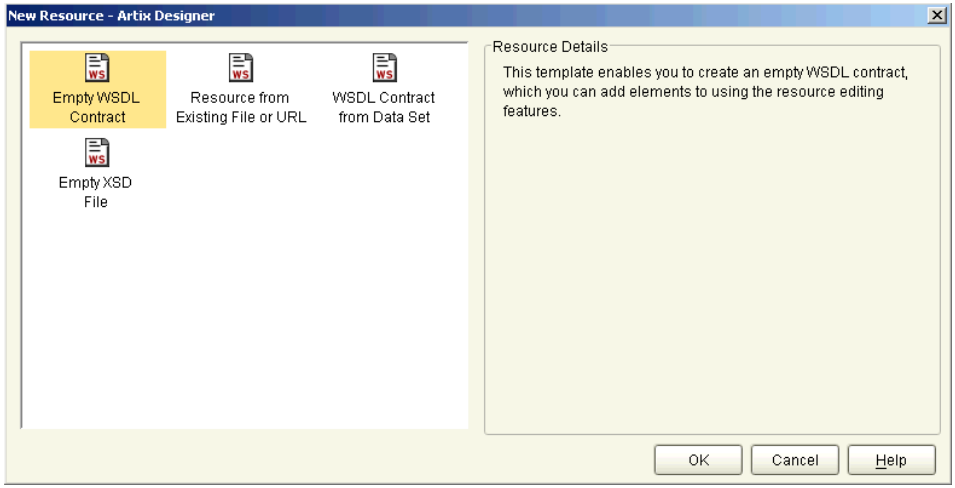


Figure 12: *New Resource dialog*

2. Select Empty WSDL Contract and click **OK** to display the New Contract dialog.

Note: The Resource from Existing File or URL icon lets you to create a resource from an existing WSDL or IDL file, and the Contract from Data Set icon let you create a resource from a description of an existing data record format.

3. In the New Contract dialog, enter **`HelloWorldGuiTutorial`** into the Name text box, and enter **`http://www.iona.com/guitutorial`** into the TargetNamespace text box.

   These entries become the values of the `name`, `targetNamespace`, and `xmlns:tns` attributes in the opening `<definitions>` element in the WSDL file.

4. Click the **OK** button to close this dialog and return to the Artix Designer.

   Artix Designer displays the structure of the new contract in the Resource Navigator.

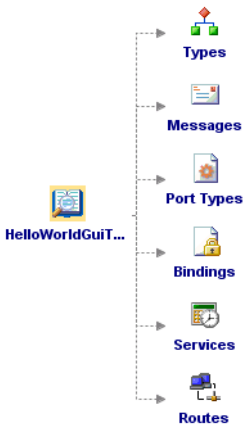


Figure 13: *Resource Navigator containing new contract*

5. To view the contents of the WSDL file click on the **Text** tab.

The WSDL appears as shown in Figure 14:

Figure 14: *New Contract—Text view*

Currently the WSDL file only includes the opening `<definitions>` element.

Defining the Contract Elements

Overview

The Artix contract mirrors the structure of a WSDL file. As such, you need to create these elements in your contract to make it a valid WSDL file for use by Artix.

The Artix Designer makes it very easy to create these elements by providing a series of wizards—one for each of the WSDL elements:

- Types
- Messages
  Port Types
- Bindings
- Services

Figure 15 displays the first panel of the New Type wizard, as an example.

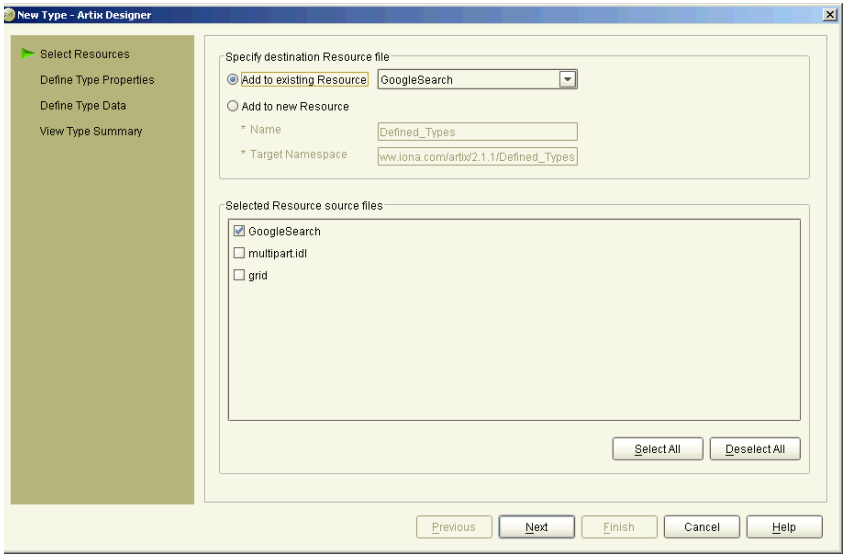


Figure 15: *New Type wizard*

This section guides you through creating the contract elements, in the following topics:

Defining the Types

Overview

The `<types>` section of the WSDL file contains your data type definitions. For this simple application, you have several choices:

- You could choose not to define unique types for the application and use a basic type instead, for example, `xsd:string`, as message parts.
- Alternatively, you could define simple types; that is, new types that are derived from an existing `xsd` type.
- Lastly, you could define element types, which are wrappers around other defined types. This approach is especially useful if your types are complex or highly structured; for example, a structure or array. Additionally, using elements as message parts allows you to select the document/literal encoding format for your binding.

  This is the approach outlined in this subsection.

Defining the simple types

To create a type, complete the following steps:

1. Select the **HelloWorldGuiTutorial** icon under Shared Resources and select **Resource | New | Type** to display the New Type wizard, as shown in Figure 15.
2. In the Select WSDL panel, select the **Add to existing WSDL "HelloWorldGuiTutorial"** radio button and click **Next** to display the Define Type Properties panel, as shown in Figure 16.

Note: Throughout the entire WSDL file creation process you add your new content to the current WSDL file. Selecting the **Add to new WSDL** radio button creates another WSDL file that includes selected content. This is the approach you follow if you want to create WSDL file fragments for reuse.

3. In the Define Type Properties panel, enter **`InParameter`** into the Name text box and select the **simpleType** radio button.

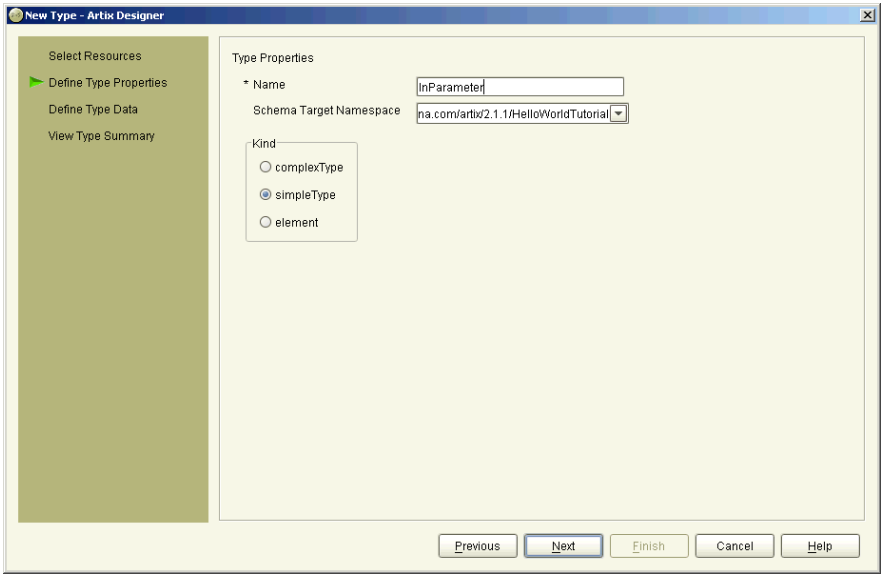


Figure 16: *New Type wizard—Define Type Properties panel*

4. Accept the default namespace provided.
5. Click **Next** to display the Define Type Attributes panel.
6. Select **xsd:string** from the **Base Type** drop down list.

   The remaining controls are used to further restrict the simple type, for example, limiting the length of the string to a specific number of characters or to one of a restricted number of entries. For this example, you do not need these additional restrictions.
7. Click **Next** to display the View Summary panel, as shown in Figure 17.

8. In the View Summary panel, you can review the content that will be added to the WSDL file.

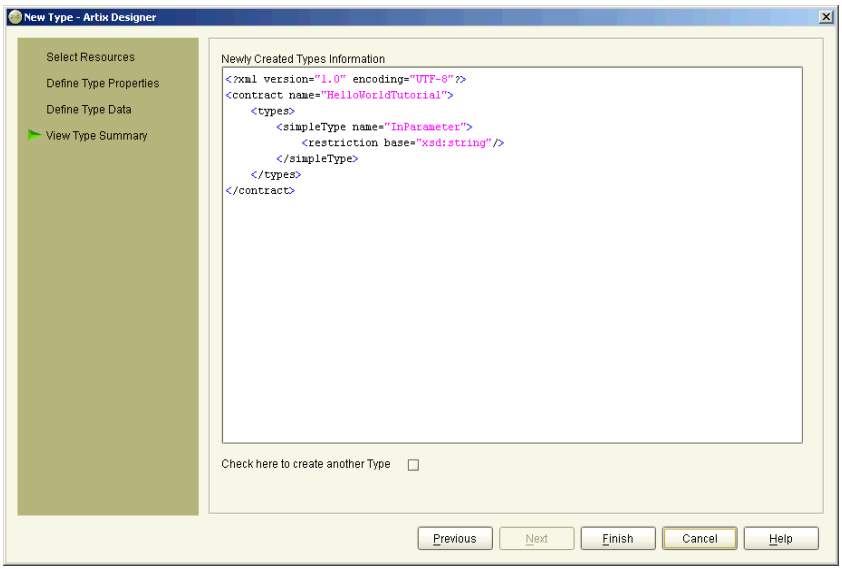


Figure 17: *New Type wizard—Summary panel*

9. Select the check box at the bottom of the panel to create a new type and click **Next** to return you to the first panel in the wizard, as shown in Figure 15.
10. Create a second simple type named **`OutParameter`**.

    Note that the **Base Type** drop-down list now includes `ns1:InParameter` as a valid type. Be careful, as newly defined types are added to the top of the list; you need to scroll down the list to find the `xsd:string` entry.
11. Click **Finish** to close the wizard and return to the Artix Designer.
12. Click on the **Save** icon in the toolbar, or select **File | Save** to save the changes to your workspace.

13. Double-click on the Types icon in the right-hand pane to see the two new parameters, as shown in Figure 18:

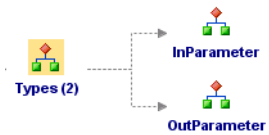


Figure 18: *Resource Navigator showing the new Types*

14. Select the Text tab to review the current contents of the WSDL file. Note that the `<types>` section has been added to the file.

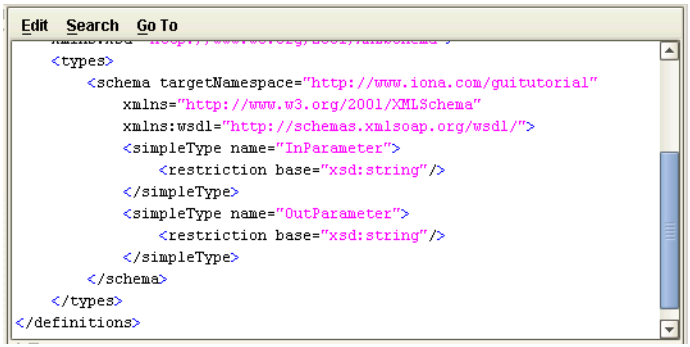


Figure 19: *WSDL with Types added*

Defining the Element Types

You now need to define element types that wrap each of your simple types. The procedure for doing this is identical to that of defining a simple type except you select the **element** radio button in the Define Type Properties panel of the New Type wizard.

To create the element types, complete the following steps:

1. Select the **HelloWorldGuiTutorial** icon in the Designer Tree and select **Resource | New | Type**.
2. In the Select WSDL panel, select the **Add to existing WSDL "HelloWorldGuiTutorial"** radio button and click **Next** to display the Define Type Properties panel.
3. In the Define Type Properties panel, enter **`InElement`** into the Name text box, and accept the default provided for the namespace.
4. Select the **element** radio button, as shown in Figure 20.

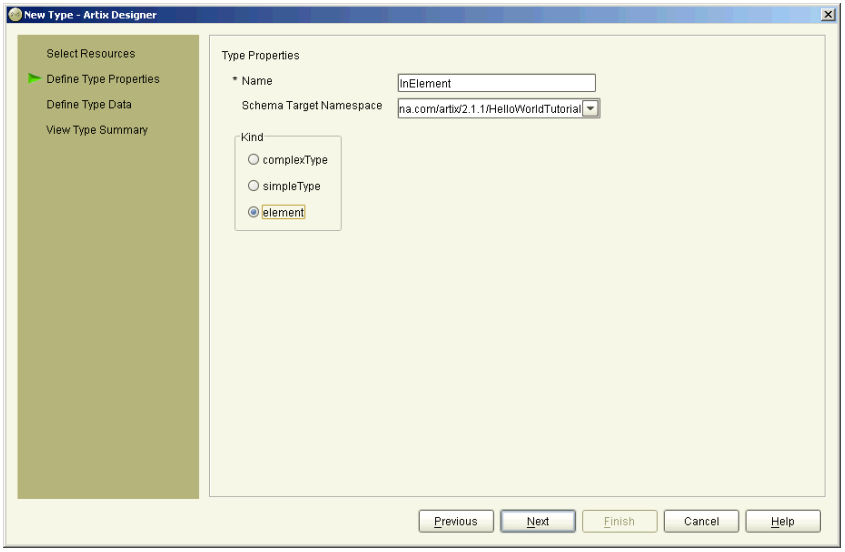


Figure 20: *New Type wizard—Define Type Properties panel*

5. Click **Next** to display the Define Type Attributes panel.
6. Select **ns1:InParameter** from the **Type** drop-down list.
7. Click **Next** to display the View Summary panel.
8. Select the check box at the bottom of the panel to create another new type and click **Next** to return you to the first panel in the wizard.
9. Create a second type named **`OutElement`**. The type should be **ns1:OutParameter**.

10. Click **Finish** to close the wizard and return to the Artix Designer.
11. Click on the **Save** icon in the toolbar or select **File | Save**.
12. Select the **HelloWorldGuiTutorial** icon and click the Text tab to review the contents of the WSDL file, as shown in Figure 21.

Figure 21: *New Types in WSDL*

Note that the `<types>` section now includes four definitions:

- Simple type `InParameter`, of type `xsd:string`.
- Simple type `OutParameter`, of type `xsd:string`.
- Element type `InElement`, of type `ns1:InParameter`.
- Element type `OutElement`, of type `ns1:OutParameter`.

For a more thorough explanation of creating types, including all of the screen shots from the New Type wizard, see *Designing Artix Solutions*.

Defining the Messages

Overview

Now that you have defined the required types, you can begin to define the messages. Your types are used as the message parts.

Procedure

To define a message, complete the following steps:

1. Select the **HelloWorldGuiTutorial** icon and select **Resource | New | Message** to display the New Message wizard, as shown in Figure 22.

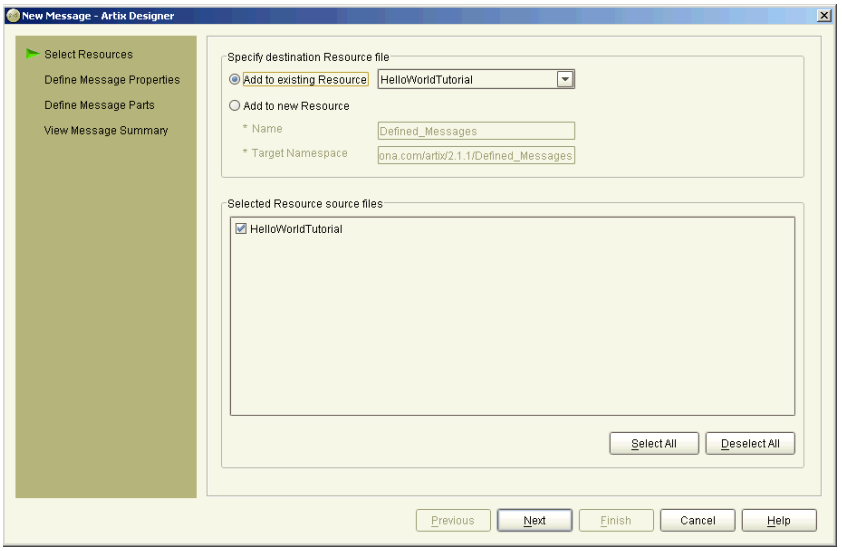


Figure 22: *New Message wizard—Select WSDL panel*

2. Select the **Add to existing WSDL "HelloWorldGuiTutorial"** radio button and click **Next** to display the Define Message Properties panel.
3. Type **`RequestMessage`** in the **Name** field and click **Next** to display the Define Parts panel, as shown in Figure 23.

4. Type **`InPart`** into the Name text box and select **ns1:InElement** from the Type drop-down list.

   Note: Be careful — Do not select **ns1:InParameter** from the Type drop-down list

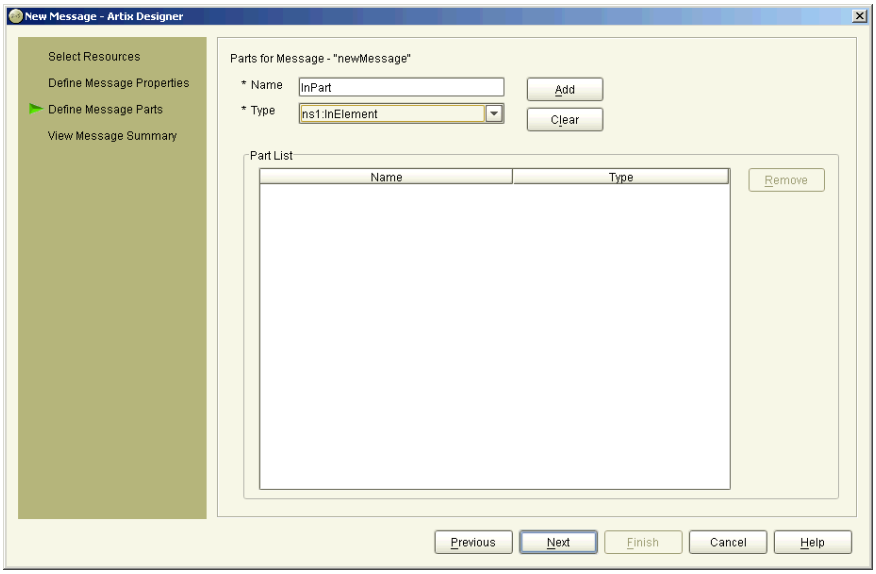


Figure 23: *New Message wizard—Define Parts panel*

5. Click **Add** to add your part to the Part List control.

   If your message requires multiple parts (which is not the situation in this case), you can define another part and add it to the Part List.

6. Click **Next** to display the View Summary panel as shown in Figure 24.

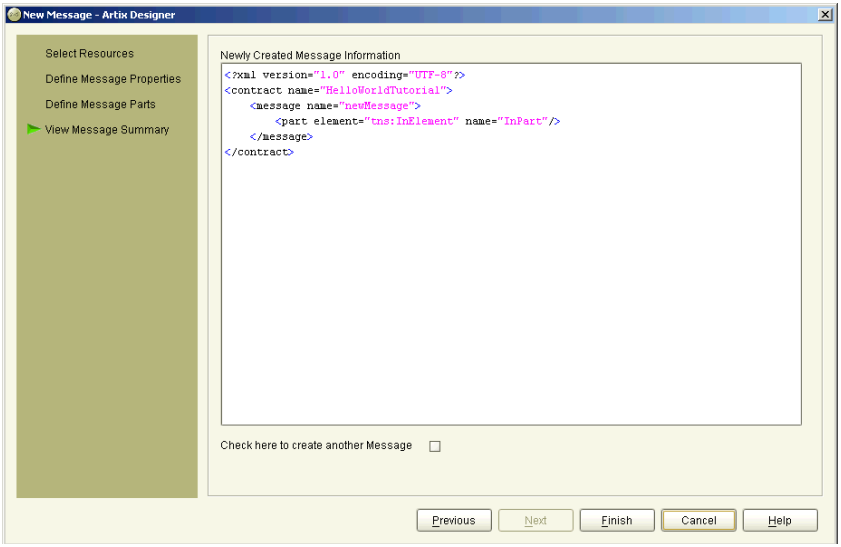


Figure 24: *New Message wizard—View Summary panel*

7. Select the check box at the bottom of the panel to create another new message and click **Next** to return you to the first panel in the wizard.
8. Repeat the process to create a second message – **`ResponseMessage`** – with a part named **`OutPart`** of type **ns1:OutElement**.
9. This time at the View Summary panel, click **Finish** to close the wizard and return to the Artix Designer.
10. Click on the **Save** icon in the toolbar or select **File | Save**.

11. Select the **HelloWorldGuiTutorial** icon and click the **Text** tab to review the contents of the WSDL file, as shown in Figure 25.

Figure 25: *New messages definition in WSDL*

For a more thorough explanation of creating messages, including all of the screen shots from the New Message wizard, see *Designing Artix Solutions*.

Defining the Port Type

Overview

A port type contains operations, which are composed of one or more messages:

- A one-way operation includes only an input message; the client application does not receive a response from the Web service.
- A request-response operation includes an input message, an output message, and zero, or more, fault messages. Defining and coding fault messages will be discussed in the following chapter.

In this example, you will define a port type that includes one request-response operation called `sayHi` which uses `RequestMessage` as its input and `ResponseMessage` as its output.

There is nothing significant about the names assigned to the messages or parts; name assignments are to assist the developer. Artix does not care what names are used. An identical application could be created by naming the messages One and Two, and the parts X and Y.

Procedure

To create a new port type, complete the following steps:

1. Select the **HelloWorldGuiTutorial** icon and select **Resource | New | Port Type** to display the New Port Type wizard.
2. In the Select WSDL panel, select the **Add to existing WSDL "HelloWorldGuiTutorial"** radio button and click **Next** to display the Define Port Properties panel.

3. Type **`GuiTutorialPT`** in the **Name** field and click **Next** to display the Define Port Type Operations panel, as shown in Figure 26.

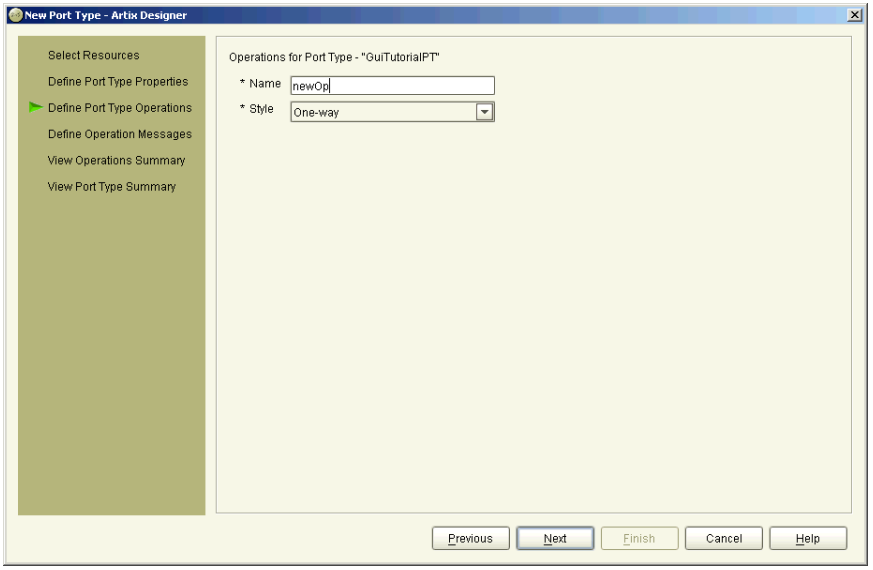


Figure 26: *New Port Type wizard—Define Port Type Operations panel*

4. Type **`sayHi`** in **Name** field.

5. Select Request-response from the **Style** drop-down list and click **Next** to display the Define Operations Messages panel.

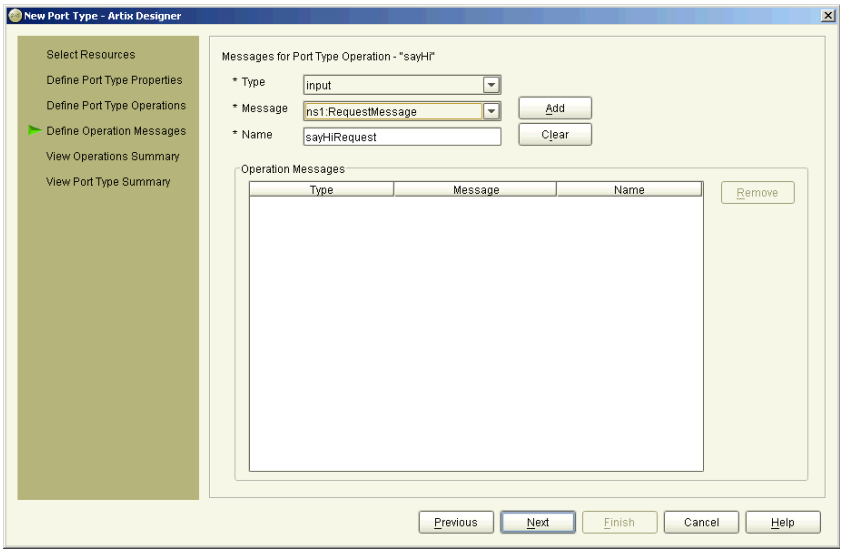


Figure 27: *New Port Type wizard—Define Operation Messages panel*

6. Select input from the **Type** drop-down list and **ns1:RequestMessage** from the **Message** drop-down list.

   The Name **sayHiRequest** appears in the Name text box. If desired, you can change this entry to something more meaningful to your application. In this example, leave the suggested content.

7. Click **Add**, which transfers the input message to the Operation Messages control.

8. Now, click on the **Type** drop-down list. Note that input no longer appears in the listing; an operation can have only one input message. Select output from the **Type** list and **ns1:ResponseMessage** from the **Message** drop-down list.

   The Name **sayHiResponse** appears in the Name text box; leave this suggested content.

9. Click **Add** to transfer the ouptut message to the Operation Messages control.

   If you click on the **Type** drop-down list you will see that the **output** entry no longer appears in the listing; an operation can have only one output message. Although in this example you are not adding fault messages to this operation, multiple fault message can be added to an operation. To do this, you repeat the process outlined above.

10. Lastly, click **Next** to display the View Port Operations Summary panel. Since this example only requires one `portType`, click **Next** to display the Port Type Summary panel, and then **Finish** to close this wizard and return to the Artix Designer.

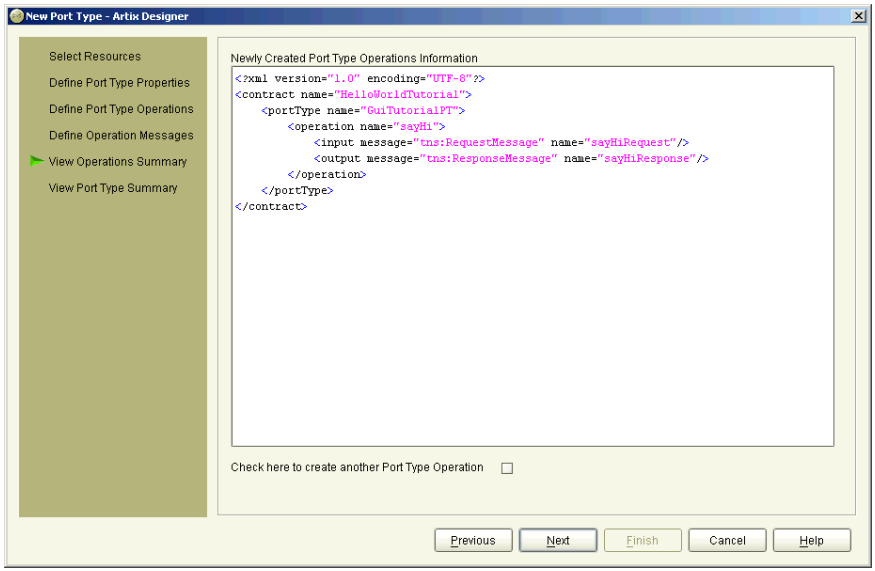


Figure 28: *New Port Type wizard—Define Operation Messages panel*

11. Click on the **Save** icon in the toolbar or select **File | Save**.
12. Select the **HelloWorldGuiTutorial** icon and click the **Text** tab, and review the contents of the WSDL file.

Defining the Binding

Overview

A binding describes how the messages are marshalled. Each binding is associated with a single `portType`, although the same `portType` can be associated with multiple bindings.

In "Coding the Web Service" on page 39, the binding used the rpc/encoded style. In this example, you specify the document/literal style, which is required when message parts are element types.

Procedure

To define a binding, complete the following steps:

1. Select the **HelloWorldGuiTutorial** icon and select **Resource | New | Binding to** display the New Binding wizard.
2. In the Select WSDL panel, select the **Add to existing WSDL "HelloWorldGuiTutorial"** radio button and click **Next** to display the Select Binding Type panel.

3. Select **SOAP** and click **Next** to display the Set Binding Defaults panel, as shown in Figure 29.

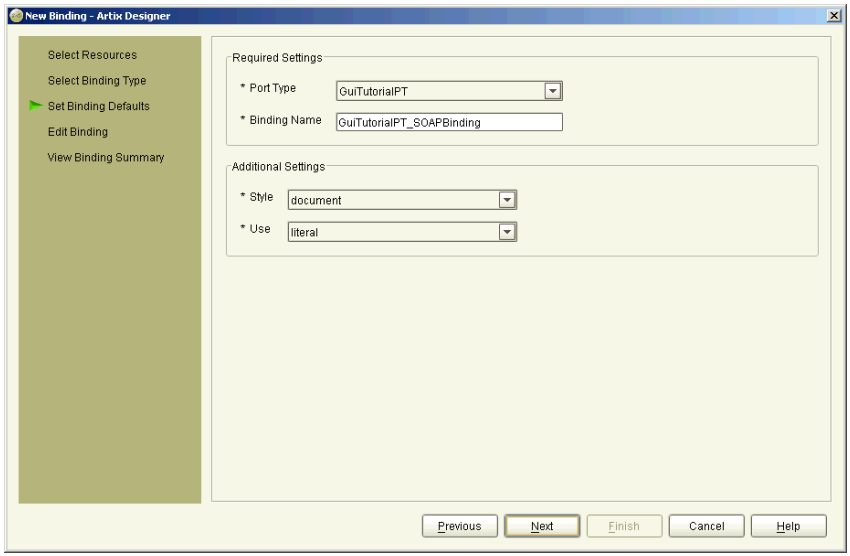


Figure 29: *New Binding wizard—Set Binding Defaults panel*

4. Select GuiTutorialPT from the **Port Type** drop-down list.

   Because your WSDL file only contains one `portType` definition, this is the only one in the list. If there were multiple port types defined, you would need to select the desired `portType` from the list.

   Note that a name is already entered into the **Binding** field. You can change this entry. The only requirement is that each binding in the WSDL file be given a unique name.

5. In the Additional Settings group there are two drop-down lists. From the **Style** list, select document, and from the **Use** list, select literal. If you select an invalid combination, for example **rpc/encoded** or **document/encoded**, you will not be able to move to the next window. Click **Next** to display the Edit Binding panel, as shown in Figure 30.

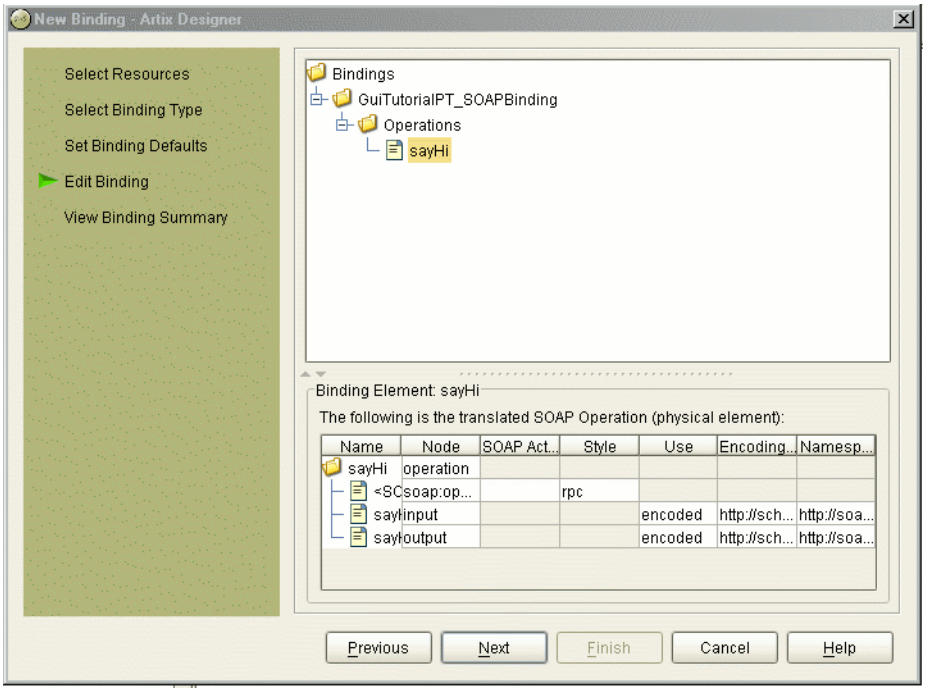


Figure 30: *New Binding wizard—Edit Binding panel*

6. Select the **sayHi** icon representing your operation and review the binding details. Click **Next** to view the View WSDL Contract panel, where you can review the content that will be added to the WSDL file.
7. Click **Finish** to close this wizard and return to the Artix Designer.
8. Click on the **Save** icon in the toolbar or select **File | Save**.
9. Select the **HelloWorldGuiTutorial** icon and click on the WSDL tab to review the contents of the WSDL file.

Defining the Service

Overview

A service provides transport-specific information. Each service element can include one, or more, port elements. The port elements must be uniquely identified through the value of the name attribute. Each port element is associated with a single binding element, although the same binding element can be associated with one or more port elements. In addition, a WSDL file may contain multiple service elements.

In this example, the WSDL file contains one service element, which contains a single port element.

Procedure

To create a service, complete the following steps:

1. Select the **HelloWorldGuiTutorial** icon and select **Resource | New | Service** to display the New Service wizard.
2. Select the **Add to existing WSDL "HelloWorldGuiTutorial"** radio button and click **Next** to display the Define Service panel.

3. Type **`HelloWorldService`** in the **Name** field and click **Next** to display the Define Port panel, as shown in Figure 31.

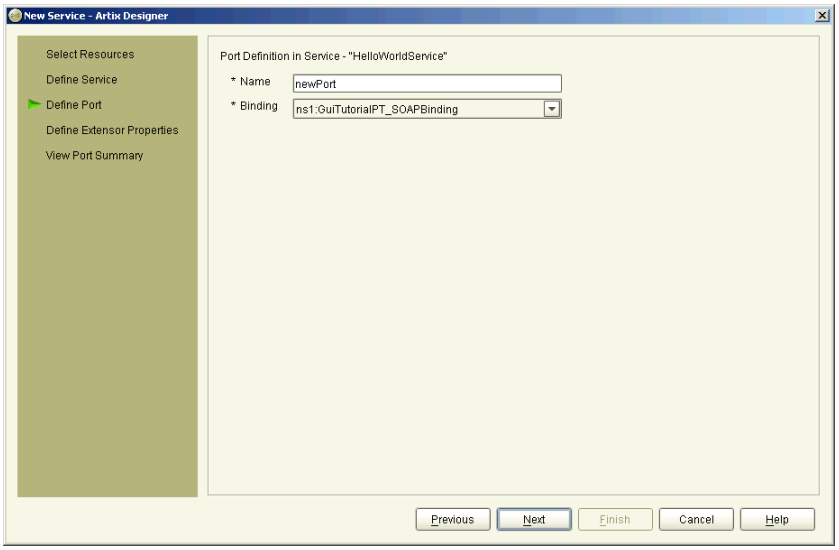


Figure 31: *New Service wizard—Define Port panel*

4. Type **`HelloWorldPort`** in the **Name** field and select `ns1:GuiTutorialPT_SOAPBinding` from the **Binding** drop-down list.

   Because your WSDL file only contains one binding definition, this is the only entry in the list. If there were multiple bindings defined, you would need to select the desired binding from the list.

Click **Next** to display the Define Extensor Properties panel, as shown in Figure 32.

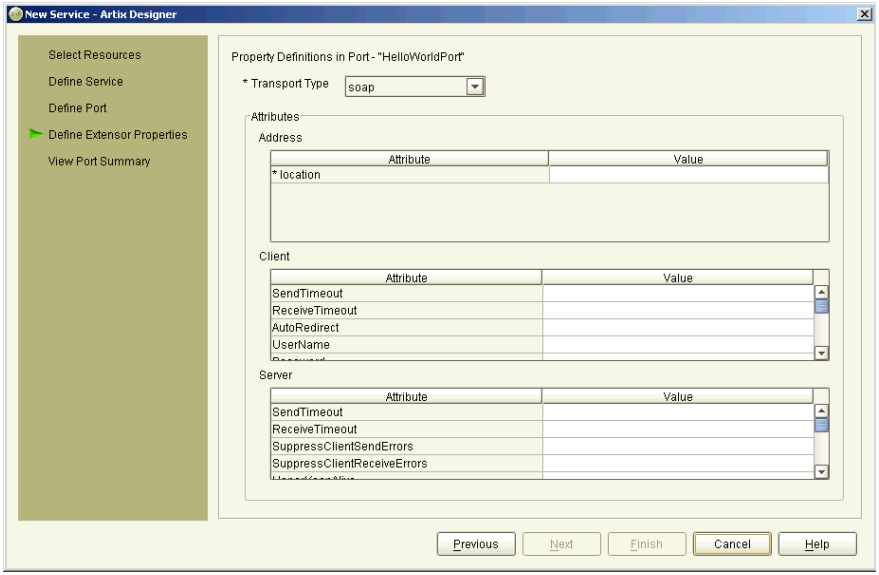


Figure 32: *New Service wizard—Define Extensor Properties panel*

5. Select **soap** from the **Transport Type** drop-down list and enter `http://localhost:9000` as the value for the **Location**. This is the only required field, and you can specify any port number you choose.

   Click **Next** to display the Port Summary panel where you can review the new content that will be added to the WSDL file.
6. Lastly, click **Finish** to close this wizard and return to the Artix Designer.
7. Click on the **Save** icon in the toolbar or select **File | Save**.
8. Select the **HelloWorldGuiTutorial** icon, click on the Text tab, and review the contents of the WSDL file.

For a more thorough explanation of adding services, including all of the screen shots from the New Service wizard, see *Designing Artix Solutions*.

You've now completed your WSDL file, and you are ready to use it to develop an application.

Developing an Application

Overview

Currently your WSDL contract is located under the Shared Resources icon within the Artix Designer and in the `GuiTutorial\Resources` directory on your drive. Other than the fact that this WSDL file uses element types and document/literal encoding, this WSDL file is functionally equivalent to the file used in "Coding the Web Service" on page 39. You could repeat that example using this WSDL file instead of the file provided in this document.

In this section, use the same WSDL file for both the client and server applications. With the Artix Designer you accomplish this goal by creating Collections, which are the resources that are used by a Web service component.

This section contains the following topics to help you develop an application:

- "Creating a client application"
- "Creating a server application" on page 108

Creating a client application

When you create a new collection, Artix places a corresponding icon under the Collections icon in the Designer Tree.

To create a collection, complete the following steps:

1. Select **File | New | Collection**. to display the New Collection dialog.
2. Select the **New Collection** icon and click **OK** to display the New Collection wizard, as shown in Figure 33.

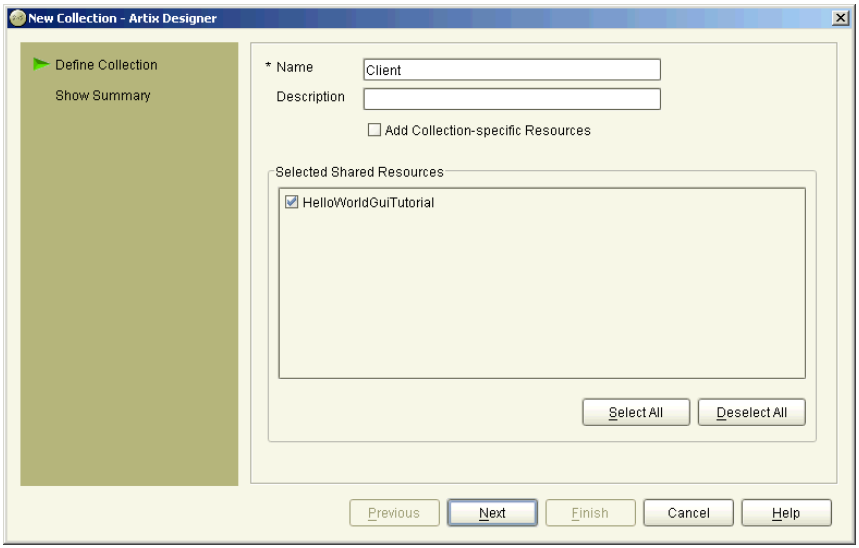


Figure 33: *New Collection wizard*

3. Type **`Client`** in the **Name** field.

   If you wanted to add any extra resources (other than the Shared ones listed on this panel) to this collection, you would select the Add Collection Resources check box. For the purpose of this example, however, you are not adding any extra resources to this collection.

   Note that the Artix Designer automatically adds the `HelloWorldGuiTutorial` resource to the collection. If you had a workspace with multiple entries under the Shared Resources icon, you might not want to include each resource in every collection. In such cases, uncheck any entry you want excluded.

   Click **Next** and then **Finish** to close this wizard and return to the Artix Designer.

4. Click on the **Save** icon in the toolbar or select **File | Save**.

The Designer Tree now includes an icon representing the client application. Note the nested icon and italic font representing the included (shared) resource file. This format indicates that the collection's resource is a link to the resource file listed under the Shared Resources icon and not a resource uniquely associated with this application.

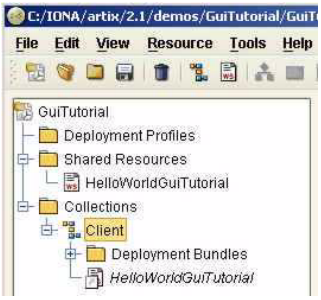


Figure 34: *New Client item*

In a more complex application, your collection might include both shared resources and resources specific to the application. In this situation, you could create a new resource within the collection rather than as a shared resource as described in step 3 in this section.

Creating a server application

Following the same procedure, create a second collection named **`Server`** that also includes the **HelloWorldGuiTutorial** resource.

You have now created your client and server applications. The next step is to generate the starting point code.

Generating Starting Point Code

Overview

The Artix Designer is capable of generating starting point (stub and skeleton) code plus configuration scripts for multiple platforms and operating systems and for multiple implementation languages. Depending on your objectives, you might want to generate only client code, only server code, or only some of the files produced by the `wsdltocpp` or `wsdltojava` utilities. You provide this information to the Artix Designer by defining a Deployment Profile and Deployment Bundle, and then running the Code Generator.

This section outlines how to do this. The following topics are covered:

Defining Deployment Profiles

Overview

A Deployment Profile contains platform, operating system, and implementation language specifications that apply to all entities in the workspace.

A deployment profile can be used with multiple deployment bundles in generating code for the same platform, operating system, and implementation language.

This section contains the following topics:

- "Creating the C++ profile"
- "Creating the Java profile"

Creating the C++ profile

To create the C++ deployment profile, complete the following steps:

1. Select the **GuiTutorial** icon in the Designer Tree and select **File | New | Deployment Profile** to display the New Deployment Profile wizard.
2. Type **`cxx_profile`** in the **Name** field and select **Windows** from the **Operating System** drop-down list.
3. Click **Next** to display the Artix Location panel.
4. Confirm that the paths to your Artix Installation Directory and `artix_env` script files are correct.
5. Select the **C++** radio button and click **Next** and then **Finish** to close this wizard and return to the Artix Designer. The C++ Profile is listed in the Designer Tree under the Deployment Profiles folder.
6. Click on the **Save** icon in the toolbar or select **File | Save**.

You use this profile to generate C++ code for the Windows operating system.

Creating the Java profile

To create the Java deployment profile, complete the following steps:

1. Select the **GuiTutorial** icon in the Designer Tree and select **File | New | Deployment Profile** to display the New Deployment Profile wizard.
2. Type **`java_profile`** in the **Name** field and select **Windows** from the **Operating System** drop-down list.
3. Click **Next** to display the Artix Location panel.

4. Confirm that the paths to your Artix Installation Directory and `artix_env` script files are correct.
5. Select the **Java** radio button and click **Next** and **Finish** to close this wizard and return to the Artix Designer. The Java Profile is listed in the Designer Tree under the Deployment Profiles folder.
6. Click on the **Save** icon in the toolbar or select **File | Save**.

You use this profile to generate Java code for the Windows operating system.

Defining Deployment Bundles

Overview

A Deployment Bundle defines a collection's deployment-specific details, such as the deployment type and the code to be used (C++ or Java).

Note: You can create multiple deployment bundles for a collection, but you must create at least one deployment profile before creating a bundle.

Because you have separate collections for the client and server applications, and you want to generate both C++ and Java starting point code, you need to define four deployment bundles:

- C++ client
- C++ server
- Java client
- Java server

Creating the two bundles is almost exactly the same procedure; the only difference occurs when it comes to selecting the deployment profile and setting the Code Generation options.

This section contains the following topics:

- "Creating the C++ client bundle"
- "Creating the Java client bundle" on page 113
- "Creating the C++ server bundles" on page 114
- "Creating the Java server bundle" on page 114

Creating the C++ client bundle

To create the C++ client bundle, complete the following steps:

1. Select the Client collection in the Designer Tree.
2. Select **File | New | Deployment Bundle** to display the New Deployment Bundle wizard.
3. Type **`cxx_client`** in the Name field.
4. Type the path to the directory into which you want to generate the starting point code in the Location field. For this example, accept the default: `C:\IONA\artix\2.1\demos\GuiTutorial\Client\cxx_client`.
5. Select **cxx_profile** from the Deployment Profile drop-down list.

6. Select the **Client** radio button and click **Next** to display the Code Generation panel.
7. Check the Generate Code and Sample check boxes for the **HelloWorldService** and **HelloWorldPort**.
8. Check the **Configure Language Options** check box and click Next to display the Language Options panel.
9. Enter **`GUI`** as the value for the Namespace. This entry becomes the C++ namespace within the generated code.
10. Click **Next** twice, and then **Finish** to close this wizard and return to the Artix Designer.
11. Click the **Save** icon in the toolbar or select **File | Save**.

Creating the Java client bundle

To create the Java client bundle, complete the following steps:

1. Select the Client collection in the Designer Tree.
2. Select **File | New | Deployment Bundle** to display the New Deployment Bundle wizard.
3. Type **`java_client`** in the **Name** field.
4. Enter the path to directory into which you want to save the starting point code in the Location field. For this example, accept the default: `C:\IONA\artix\2.1\demos\GuiTutorial\Client\java_client`.
5. Select **java_profile** from the Deployment Profile drop-down list.
6. Select the **Client** radio button and click **Next** to display the Code Generation panel.
7. Check the Generate Code and Sample check boxes for the **HelloWorldService** and **HelloWorldPort**.
8. Check the **Configure Language Options** check box and click Next to display the Language Options panel.
9. Click the **Override Namespace as packaging name** check box, and enter **`com.iona`** as the value for the Java Package within the Code Generation Options grouping. This entry becomes the Java package hierarchy within the generated code.
10. Click **Next** twice, and then **Finish** to close this wizard and return to the Designer.
11. Click the **Save** icon in the toolbar or select **File | Save**.

Creating the C++ server bundles

To create the C++ server bundle, complete the following steps:

1. Select the Server collection in the Designer Tree.
2. Select **File | New | Deployment Bundle** to display the New Deployment Bundle wizard.
3. Type **`cxx_server`** in the Name field.
4. Enter the path to directory into which you want to save the starting point code in the Location field. For this example, accept the default: `C:\IONA\artix\2.1\demos\GuiTutorial\Server\cxx_server`.
5. Select **cxx_profile** from the Deployment Profile drop-down list.
6. Select the **Server** radio button, and click **Next** to display the Code Generation panel.
7. Check the Generate Code and Sample check boxes for the **HelloWorldService** and **HelloWorldPort**.
8. Check the **Configure Language Options** check box and click Next to display the Language Options panel.
9. Enter **`GUI`** as the value for the Namespace within the Code Generation Options grouping. This entry becomes the C++ namespace within the generated code.
10. Click **Next** twice, and then **Finish** to close this wizard and return to the Designer.
11. Click the **Save** icon in the toolbar or select **File | Save**.

Creating the Java server bundle

To create the Java server bundle, complete the following steps:

1. Select the Server collection in the Designer Tree.
2. Select **File | New | Deployment Bundle** to display the Deployment Bundle wizard.
3. Type **java_server** in the Name field.
4. Enter the path to directory into which you want to save the starting point code in the Location field. For this example, accept the default: `C:\IONA\artix\2.1\demos\GuiTutorial\Server\java_server`.
5. Select **java_profile** from the Deployment Profile drop-down list.
6. Select the **Server** radio button, and click **Next** to display the Code Generation panel.

7. Check the Generate Code and Sample check boxes for the **HelloWorldService** and **HelloWorldPort**.
8. Check the **Configure Language Options** check box and click Next to display the Language Options panel.
9. Click the **Override Namespace as package name** check box, and enter `com.iona` as the value for the Java Package within the Code Generation Options grouping. This entry becomes the Java package hierarchy within the generated code.
10. Click **Next** twice, and then **Finish** to close this wizard and return to the Artix Designer. (The skipped windows are used when specifying more advanced code generation processes.)
11. Click the **Save** icon in the toolbar or select **File | Save**.

You have now created all the required deployment artifacts and are ready to generate the code for your applications.

Generating the C++ and Java Code

Overview

When you have created your deployment profile and bundle, you can use the **Code Generator** to create the code for your collection. This generates the code, environment scripts, and configuration files in the locations that you specify.

This section contains the following topics:

Generating the C++ client code

To generate the code for the C++ client, complete the following steps:

1. Select the Client icon in the Designer Tree and select **Tools | Generate Code** to display the Generate Code dialog, as shown in Figure 35.

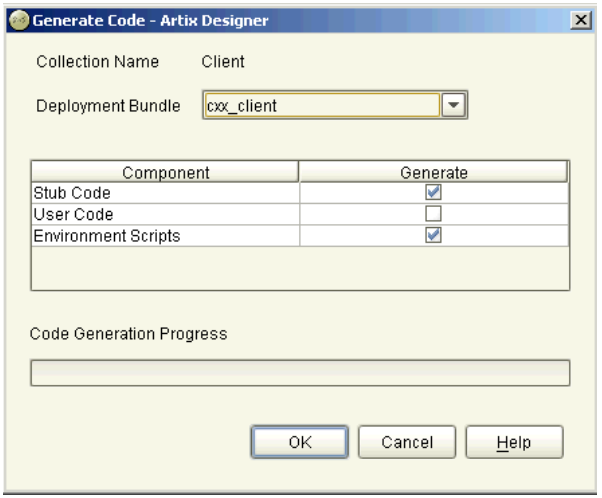


Figure 35: *Generate Code dialog*

2. Select **cxx_client** from the Deployment Bundle drop-down list. Note that the list only displays bundles that were defined for the client application.
3. The deployer is preconfigured to generate stub code and environment scripts for the client application. To generate all of the client starting point code, check the box under the Generate heading for the User Code component
4. Click **OK**.

   The code generation process runs to completion.
5. Click **Close** to close this dialog and return to the Artix Designer.

Generating the Java client code

To generate the code for the Java Client, complete the following steps:

1. Select the Client icon in the Designer Tree and select **Tools | Generate Code** to display the Generate Code dialog.
2. Select **java_client** from the Deployment Bundle drop-down list. Note that the list only displays bundles that were defined for the client application.
3. The deployer is preconfigured to generate stub code and environment scripts for the client application. Since you want to generate all of the client starting point code, check the box under the Generate heading for the User Code component.
4. Click **OK**.

   The code generation process runs to completion.
5. Click **Close** to close this dialog and return to the Artix Designer.

Generating the C++ server code

To generate the code for the C++ server, complete the following steps:

1. Select the Server icon from the Designer Tree and select **Tools | Generate Code** to display the Generate Code dialog.
2. Select **cxx_server** from the Deployment Bundle drop-down list.
3. The deployer is preconfigured to generate stub code and environment scripts and start/stop scripts for the server application. To generate all of the server starting point code, check the box under the Generate heading for the User Code component.
4. Click **OK**.

The code generation process runs to completion.

5. Click **Close** to close this dialog and return to the Artix Designer.

Generating the Java server code

To generate the code for the Java server, complete the following steps:

1. Select Server icon from the Designer Tree and select **Tools | Generate Code** to display the Generate Code dialog.
2. Select **java_server** from the Deployment Bundle drop-down list.
3. The deployer is preconfigured to generate stub code and environment scripts and start/stop scripts for the server application. To generate all of the server starting point code, check the box under the Generate heading for the User Code component.
4. Click **OK**.

   The code generation process runs to completion.
5. Click Close to close this dialog and return to the Artix Designer.

You have now generated all the required code and are finished with the Artix Designer for now. Close the application by selecting **File | Exit**.

Adding Logic to the Code

Overview

Through the code generation process you created four applications:

- The C++ client
- The Java client
- The C++ server
- The Java server.

All of these applications compile and run. However, because there is no business logic in the `sayHi` method body, and because the C++ client code does not actually make a request against the Web service, running the applications does not produce output.

You need to complete the coding in the files representing the C++ and Java implementation objects and in the C++ client mainline file. The Java client mainline file is a complete, albeit very basic, application and so needs no modification.

This section contains the following topics:

The C++ Client Code

Overview

The code generation produces several files. This subsection explains what each of these files is for. The files are:

- "GuiTutorialPT.h"
- "GuiTutorialPTClient.h/.cxx"
- "HelloWorldGuiTutorial_ wsdlTypes.h/.cxx"
- "HelloWorldGuiTutorial_ wsdlTypesFactory.h/.cxx" on page 121
- "GuiTutorialPTClientSample.cxx" on page 121

GuiTutorialPT.h

This header file is common to both the client and server applications. It contains the signatures for each of the Web service operations. Open this file in a text editor and review the signature for the `sayHi` method.

```
virtual void
  sayHi(
    const InParameter & InPart,
          OutParameter & OutPart
     ) IT_THROW_DECL((IT_Bus::Exception)) = 0;
```

Note that although the message parts were defined as the element types `InElement` and `OutElement`, the method signature uses C++ classes derived from the simple types `InParameter` and `OutParameter`.

GuiTutorialPTClient.h/.cxx

These files represent the client proxy class. Your client mainline code must instantiate an instance of this class to invoke on the Web service. The proxy class includes multiple constructors, a destructor, and a method for each of the Web service's operations.

In this simple application your client code uses the no argument constructor. Alternative constructors allow you to change the WSDL file, service name, or port name initialization values. One constructor allows initialization from an Artix reference.

HelloWorldGuiTutorial_ wsdlTypes.h/.cxx

These files are common to both the client and server applications and include the definitions and implementations for the classes that represent your application-specific types.

You must review the contents of these files to understand how to use the APIs of these classes.

HelloWorldGuiTutorial_wsdlTypesFactory.h/.cxx

These files are common to both the client and server applications and include definitions and implementations for the factory methods required if your application-specific types includes the `anyType`.

For this example, you do not need to be concerned with the contents of these files.

GuiTutorialPTClientSample.cxx

For the client application, you only need to work with the `GuiTutorialPTClientSample.cxx` file.

Note that the code generation process produced a simple invocation of the `sayHi` method, but the code is commented out and there is no value assigned to the in parameter and no output statement to display the value returned in the out parameter.

To do this, complete the following steps:

1. In a text editor, open the `GuiTutorialPTClientSample.cxx` file and add the following code:

```
InParameter    InPart;
OutParameter   OutPart;

InPart.setvalue("Artix User");

client.sayHi ( InPart,  OutPart);

cout << "sayHi returned: " + OutPart.getvalue() << endl;
```

2. Save and exit the file.

The C++ Server Code

Overview

The code generation produces several files. This subsection explains what each of these files is for. The files are:

- "GuiTutorialPT.h"
- "GuiTutorialPTServer.h/.cxx"
- "HelloWorldGuiTutorial_ wsdlTypes.h/.cxx"
- "HelloWorldGuiTutorial_ wsdlTypesFactory.h/.cxx"
- "GuiTutorialPTServerSample.cxx"
- "GuiTutorialPTImpl.h/.cxx" on page 123

GuiTutorialPT.h

The header file that is common to both the client and server applications.

GuiTutorialPTServer.h/.cxx

These files represent the server stub class. Your code does not directly use this class. Rather, the implementation class is a subclass of the `GuiTutorialPTServer` class.

HelloWorldGuiTutorial_ wsdlTypes.h/.cxx

These files are common to both the client and server applications and include the definitions and implementations for the classes that represent your application-specific types.

You must review the contents of these files to understand how to use the APIs of these classes.

HelloWorldGuiTutorial_ wsdlTypesFactory.h/.cxx

These files are common to both the client and server application, and include definitions and implementations for the factory methods required if your application-specific types includes the `anyType`.

For this application, you do not need to be concerned with the contents of these files.

GuiTutorialPTServerSample.cxx

This file represents the server mainline application. For this application you do not need to edit the contents of this file. The server mainline instantiates an instance of the implementation class and registers it with the Artix runtime. The process then enters an event loop to process incoming requests.

GuiTutorialPTImpl.h/.cxx

These files represent your Web service's implementation class. The `GuiTutorialPTImpl.cxx` file contains compilable code, but there is no processing logic in the method bodies.

For the server application, you need to add processing logic to the implementation class' `sayHi` method. To do this, complete the following steps:

1. In a text editor, open the `GuiTutorialPTImpl.cxx` file and note the signature for the `sayHi` method.

   The method includes two parameters, the first representing the part within the input message and the second representing the part within the output message. The return type is void.

2. Add the following code to the `sayHi` method body:

```
OutPart.setvalue("Hello " + InPart.getvalue());
```

3. Save and exit the file.

The Java Client Code

Overview

The code generation produced several files. This subsection explains what each of these files is for. The files are:

- "GuiTutorialPT.java"
- "GuiTutorialPTTypes.java and GuiTutorialPTTypesFactory.java"
- "GuiTutorialPTDemo.java"

GuiTutorialPT.java

This file represents the interface definition common to both the client and server applications. This interface defines the operation offered by the Web service.

```
public String sayHi(String inPart) throws RemoteException
```

GuiTutorialPTTypes.java and GuiTutorialPTTypesFactory.java

Definition of the classes that create and manage `anyTypes` defined in your WSDL file.

GuiTutorialPTDemo.java

This file represents the client mainline application. For this simple example, the generated code in this file represents a fully functional application. With a more involved application, you would use this code as a template for writing a more complex client application.

The Java Server Code

The code generation produced several files. This subsection explains what each of these files is for. The files are:

- "GuiTutorialPT.java"
- "GuiTutorialPTTypes.java and GuiTutorialPTTypesFactory.java"
- "GuiTutorialPTServer.java"
- "GuiTutorialPTImpl.java"

GuiTutorialPT.java

The interface definition common to both the client and server applications.

GuiTutorialPTTypes.java and GuiTutorialPTTypesFactory.java

Definition of the classes that create and manage anyTypes defined in your WSDL file.

GuiTutorialPTServer.java

Starting point code for a server mainline application. For this simple example, the generated code in this file represents a fully functional application. With a more involved application, you might extend the generated code.

GuiTutorialPTImpl.java

Starting point code for your Web service's implementation class.

For this example, you need to add coding to the method bodies corresponding to the Web service operations. To do this, complete the following steps:

1. In a text editor, open the file `GuiTutorialPTImpl.java`.
2. Add the following code to the `sayHi` method body:

```
return "Hello " + inPart;
```

3. Save and edit the file.

Compiling the Applications

Overview

You are now ready to compile the applications. The following topics in this section will help you with the process:

- "The C++ applications"
- "The Java applications"

The C++ applications

To build the C++ client and server applications, complete the following steps:

1. Run the `artix_env.bat` file from the following directory: `<installationDirectory>\artix\2.1\bin`.
2. Move to the following directory: `<installationDirectory>\artix\2.1\demos\GuiTutorial\Client\cxx_client\src\cxx`.
3. Build the client application with the command:

```
nmake all
```

4. Move to the following directory: `<installationDirectory>\artix\2.1\demos\GuiTutorial\Server\cxx_server\src\cxx`.
5. Build the server application with the command:

```
nmake all
```

The Java applications

To build the Java client and server applications, complete the following steps:

1. Run the `artix_env.bat` file from the following directory: `<installationDirectory>\artix\2.1\bin`.
2. In the open command window, place the current directory onto the CLASSPATH with the command:

```
set CLASSPATH=.;%CLASSPATH%
```

3. Move to the following directory:
   `<installationDirectory>\artix\2.1\demos\GuiTutorial\Client\java_client\src\java`.
4. Build the client application with the command:

```
javac com\iona\*.java
```

5. Move to the following directory:
   `<installationDirectory>\artix\2.1\demos\GuiTutorial\Server\java_server\src\java`.
6. Build the server application with the command:

```
javac com\iona\*.java
```

7. Close the command window.

Running the Application

Overview

You are now ready to run the applications. The following topics in this section will help you with the process:

- "The C++ applications"
- "The Java applications"
- "Interoperability" on page 129

The C++ applications

To run the C++ client against the C++ server, complete the following steps:

1. Run the `artix_env.bat` file from the following directory: `<installationDirectory>\artix\2.1\bin`.
2. Move to the following directory: `<installationDirectory>\artix\2.1\demos\GuiTutorial\Server\cxx_server\src\cxx`.
3. Start the server process with the command:

```
start guitutorialptserver.exe
```

   The server process starts in a new command window.

4. Move to the following directory: `<installationDirectory>\artix\2.1\demos\GuiTutorial\Client\cxx_client\src\cxx`.
5. Run the client process with the command:

```
guitutorialptclient.exe
```

   Observe the message in the client process window.

6. Stop the server process by issuing the command `Ctrl-C` in its command window.

The Java applications

To run the Java client against the Java server, complete the following steps:

1. Run the `artix_env.bat` file from the following directory: `<installationDirectory>\artix\2.1\bin`.
2. In the open command window, place the current directory onto the CLASSPATH with the command:

```
set CLASSPATH=.;%CLASSPATH%
```

3. Move to the following directory: `<installationDirectory>\artix\2.1\demos\GuiTutorial\Server\java_server\src\java`.
4. Start the server process with the command:

```
start java com.iona.GuiTutorialPTServer
```

   The server process starts in a new command window.

5. Move to the following directory: `<installationDirectory>\artix\2.1\demos\GuiTutorial\Client\java_client\src\java`.
6. Run the client process with the command:

```
java com.iona.GuiTutorialPTDemo sayHi <a_Name>
```

   Observe the message in the client process window.

7. Stop the server process by issuing the command `Ctrl-C` in its command window.

Interoperability

You can also run the C++ client against the Java server or the Java client against the C++ server. Use the steps in the previous sections as a guide.

CHAPTER 6

Faults and Exceptions

This chapter explains how to declare faults in WSDL files and how to handle the corresponding C++ and Java exceptions in Artix client and server applications.

In this chapter

This chapter discusses the following topics:

Raising Exceptions

Overview

Exceptions can originate from three different sources:

- "Artix runtime libraries"
- "Artix runtime services", for example, the locator service
- "Web service business logic"

In each case, the exception is returned to the client application.

Artix runtime libraries

The Artix runtime libraries can throw a C++ exception. The WSDL file provides no information about exceptions originating from the Artix runtime libraries because these exceptions are not directly related to your Web service contract. In C++ applications, these exceptions are returned as subclasses of the Artix class `IT_Bus::Exception`. Consequently, your client code must use `try{}` and `catch (IT_Bus::Exception){}` blocks to gracefully handle possible exceptions. In Java applications, these exceptions are returned as `java.lang.Exception`.

Artix runtime services

The Artix runtime services can throw a C++ exception. Many of the Artix runtime services are described in WSDL files, and a service's operations can include fault messages. If your application uses these services, your application must also include the client-side classes generated from this WSDL file. In this case, you can use the runtime service's WSDL file, and the contents of the generated code, to understand how the WSDL faults map to C++ and Java classes. Your application code uses these classes to handle the service's exceptions.

Web service business logic

The business logic within a Web service can throw a C++ or Java exception. When you write the WSDL file that describes your Web service, you can include zero or more fault messages in each request:response operation. When you run the code generation utilities, these fault messages become C++ or Java classes that your application code uses to handle your application's exceptions.

Handling exceptions raised by either an Artix runtime service or your application's business logic is similar. You enclose your application code within a `try{}` block and use one, or more, `catch{}` blocks to handle the possible exceptions.

Handling Runtime Exceptions

Overview

This section discusses runtime exceptions, and contains the following topics:

- "Types of runtime exceptions"
- "IT_Bus::Exception Class API"
- "Handling IT_Bus::Exception" on page 135

Types of runtime exceptions

Artix includes an extensive collection of runtime exceptions, which primarily represent errors that occur during marshalling and transport:

- IT_Bus::ConnectException
- IT_Bus::DeserializationException
- IT_Bus::IOException
- IT_Bus::NoDataException
- IT_Bus::SecurityException
- IT_Bus::SerializationException
- IT_Bus::ServiceException
- IT_Bus::TransportException
- IT_Bus::UserFaultException

These exceptions are defined in corresponding header files, which are located in the `<installationDirectory>\artix\2.1\include\it_bus` directory.

IT_Bus::Exception Class API

The `IT_Bus::Exception` is the superclass for all of these exception classes. You can use the `IT_Bus::Exception` class' API to extract details about what caused the exception.

The `IT_Bus::Exception` class is actually just a typedef of the `IT_Bus::FWException` class. The header file that includes this `typedef` entry is `<installationDirectory>\artix\2.1\include\it_bus\types.h`.

Your application code will never create an instance of a runtime exception. Consequently, the only API methods you need are used to obtain a description, and optionally a message code describing the processing error.

The `IT_Bus::Exception::message()` method returns an informative description of the error that caused the runtime exception.

The `IT_Bus::Exception::error()` method returns an exception code.

Handling IT_Bus::Exception

The code generated for your C++ client application includes a `try{}` block around all of the application logic and a `catch(IT_Bus::Exception){}` block.

```
int
main(int argc, char* argv[])
{
    . . .

    try
    {
        . . .
    }
    catch(IT_Bus::Exception& e)
    {
        cout << endl << "Error : Unexpected error occured!"
             << endl << e.message()
             << endl;
        return -1;
    }
    return 0;
}
```

The Java client application's main method includes a throws Exception clause. To handle the runtime exceptions, you could place a `try{}` block around the remote method invocation followed by a corresponding `catch{}` block.

This is generally all that is needed, although your code could catch each of the runtime exceptions separately.

Working with WSDL Faults

Overview

This section discusses WSDL faults, and contains the following topics:

Defining WSDL faults

A WSDL fault is simply a message that, when using the document literal paradigm, can contain zero or one part. A message corresponding to a WSDL fault is referenced by the `<fault>` child element under the `<operation>` element. A request-response operation can include zero, or more, child fault elements. If appropriate to the Web service, the same fault message can be associated with multiple operations.

The `wsdltocpp` utility creates a C++ class corresponding to the message; a message part becomes an instance variable, and accessor methods are provided to manipulate the value of this variable. If a fault message needs to contain multiple parts, you need to define a complex type, which then becomes the type of the message part.

You need to study the generated code to understand how to create and manipulate the exception class.

As with messages representing a request or response, fault messages can contain either encoded or literal element parts. The following WSDL file fragment illustrates the WSDL file definition of a fault message.

```
<types>
 <schema targetNamespace="http://www.iona.com/guitutorial"
         xmlns="http://www.w3.org/2001/XMLSchema"
         xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
  . . .

  <complexType name="FaultDetails">
   <sequence>
    <element name="FaultMsg" type="xsd:string"/>
    <element name="FaultID"   type="xsd:int"/>
   </sequence>
  </complexType>
  <element name="DoIKnowYou" type="tns:FaultDetails"/>

 </schema>
</types>

<message name="UnknownUser">
 <part element="tns:DoIKnowYou" name="theFault"/>
</message>
```

Note that the message, `UnknownUser`, only contains one part, `theFault`, which is an instance of the element `DoIKnowYou`. `DoIKnowYou` is a wrapper around the complex type `FaultDetails`, which contains two pieces of information, a string message and a numeric code.

The operation that uses this fault must include a fault child element within the operation element, as illustrated in the following WSDL file fragment.

```
<portType name="GuiTutorialPT">
 <operation name="sayHi">
  <input message="tns:RequestMessage" name="sayHiRequest"/>
  <output message="tns:ResponseMessage" name="sayHiResponse"/>
  <fault message="tns:UnknownUser" name="sayHiFault"/>
 </operation>
</portType>
```

If you are using the Artix Designer to create your WSDL file, you do not need to worry about how to include the fault message in the binding; the Designer handles this task.

When you run the `wsdltocpp` utility, two C++ classes are generated. The class `FaultDetails` corresponds to the complex type. This class includes variables corresponding to the `FaultMsg` and `FaultID` elements, and accessor methods to manipulate these values. The class `UnknownUserException` corresponds to the `UnknownUser` message. This class includes a variable of type `FaultDetails` and accessor methods to manipulate this value.

When you run the `wsdltojava` utility, two Java classes are generated. The class `FaultDetails` corresponds to the complex type. This class includes variables corresponding to the `FaultMsg` and `FaultID` elements and accessor methods to manipulate these values. The class `FaultDetailsException` corresponds to the `UnknownUser` message. This class includes instance variables corresponding to the `FaultMsg` and `FaultID` elements and accessor methods to manipulate these values. Your code uses the `FaultDetailsException` directly; there is no need to use the `FaultDetails` class.

Throwing the exception

Although both C++ class definitions include a copy constructor, neither class includes a constructor that allows you to set the instance variables. Consequently, to throw the exception from your Web service's code, you must first instantiate and initialize an instance of the `FaultDetails` class and use this instance to initialize an instance of the `UnknownUserException` class. Finally, your code throws the `UnknownUserException` instance.

```
FaultDetails faultData;
faultData.setFaultMsg("User unknown to me");
faultData.setFaultID(200);

UnknownUserException ex;
ex.settheFault(faultData);

throw ex;
```

In the Java `FaultDetailsException` class, there is a constructor that allows you to set values for the `FaultMsg` and `FaultID`. Consequently, your code can instantiate, initialize, and throw the exception in a single line of code:

```
throw new FaultDetailsException("User unknown to me", 200);
```

Handling the exception

In C++, the exception `UnknownUserException` is derived from the Artix class `IT_Bus::UserFaultException`, which is derived from `IT_Bus::Exception`.

Consequently, you must include code to catch this exception before your code that handles `IT_Bus::Exception`. Since the catch block receives a reference to the `UnknownUserException` object, your code needs to use the accessor method to obtain the `FaultDetails` object and then extract the `FaultMsg` and `FaultID`.

```
catch(UnknownUserException& ex)
{
  FaultDetails& fd = ex.gettheFault();
  cout << "Error Message: " << fd.getFaultMsg() << endl;
  cout << "Error ID: " << fd.getFaultID() << endl;
  return -1;
}
```

In Java, the client mainline code generated by the `wsdltojava` utility includes a `catch{}` block to process the `FaultDetailsException`. Do not be concerned with the exception stack trace; the generated code prints this information:

```
catch ( com.iona.FaultDetailsException ex )
  {
    System.out.println
     ("Exception: com.iona.FaultDetailsException has Occurred.");
    ex.printStackTrace();
  }
```

Developing an Application

Overview

The GuiTutorial application developed in "Using the Artix Designer" on page 69 can be easily modified to demonstrate fault usage. To do this you must define new types representing the fault details, define a new message, and modify the `sayHi` operation details. These changes have an impact on the binding definition. It is easiest to delete the existing binding and service elements from the WSDL file and recreate these entries once the other modifications are complete.

This section contains the following topics:

Modifying the WSDL file

Start the Artix Designer and open the `GuiTutorial` project.

1. Select the `HelloWorldGuiTutorial` icon under the Shared Resources icon and click on the Text tab to display the WSDL view of the contract.
2. Select Resource | **Edit** | **Services** to display the Edit Service dialog.
3. Select the **HelloWorldService** icon in the top panel and click on **Delete**. Confirm your action by clicking **Yes** when prompted.
4. Click **Apply** and then **OK** to close this dialog and return to the WSDL view of the contract.
5. Check that the `<service>...</service>` section has been removed.

6. Select **Resource | Edit | Bindings** to display the Edit Binding dialog.
7. Select the **GuiTutorialPT_SOAPBinding** icon in the top panel and click **Delete**. Confirm you action by clicking **Yes** when prompted.
8. Click **Apply** and then **OK** to close this dialog and return to the WSDL view of the contract.
9. Check that the `<binding>...</binding>` section has been removed.

Note: If you select the Client or Server version of the **HelloWorldGuiTutorial** icon under the Collections folder and view the WSDL file contents, you will see the edited content. These icons represent links to the WSDL file in the Shared Resources directory, so edits are applied to the file associated with the Client and Server collections.

Creating the data types

To create the data types that represent the exception details, complete the following steps:

1. Select the **HelloWorldGuiTutorial** icon in the Designer Tree and select **Resource | New | Type** to display the New Type wizard.
2. Select the **Add to existing WSDL "HelloWorldGuiTutorial"** radio button and click **Next** to display the Define Properties panel.
3. Type **`FaultDetails`** in the Name field and select the **complexType** radio button.
4. Click **Next** to display the Define Type Attributes panel.
5. Select **sequence** from the Group Type drop-down list.
6. Select the **xsd:string** from the Type drop-down list.
7. Type **`FaultMsg`** in the Name field.
8. Click **Add** to transfer this element to the Element List table.
9. To add the second member of the `FaultDetails` sequence, select **xsd:int** from the Type drop-down list and then enter **`FaultID`** into the Name text box. Click the **Add** button. Your sequence now includes two members.
10. Click **Next** and then **Finish** to complete the type definition entry.

Defining the element type

To define an element type that wraps your complex types, repeat the procedure described in "Creating the data types" with the following exceptions:

1. Select the **element** radio button in the Define Type Properties window.
2. Create one element type called **DoIKnowYou** of type **ns1:FaultDetails**.

   In the Define Type Attributes window, you are only presented with a Type drop-down list. Because an element type is simply a wrapper around another type, there are no additional options.
3. When you have finished, click **Finish** to close this wizard and return to the Artix Designer.

Defining the fault message

To define the fault message, complete the following steps:

1. Select the **HelloWorldGuiTutorial** icon in the Designer Tree and select **Resource | New | Message** menu entry to display the New Message wizard.
2. Select the **Add to existing WSDL "HelloWorldGuiTutorial"** radio button and click **Next** to display the Message Properties panel.
3. Type **`UnknownUser`** in the Name field and click **Next** to display the Define Parts panel.
4. Type **`theFault`** in the Name field and select **ns1:DoIKnowYou** from the Type drop-down list.
5. Click **Add** to add the part to the Part List table.
6. Click **Next** to display the View Summary panel, where you can review the content that will be added to the WSDL file.
7. Click **Finish** to close this wizard and return to the Artix Designer.

Editing the portType definition

To edit the `portType` definition, complete the following steps:

1. Select the **HelloWorldGuiTutorial** icon from the Designer Tree and select **Resource | Edit | Port Types** to display the Edit Port Type dialog.
2. Select the **sayHi** operation in the top panel and then click **Edit** to display the Edit Operation Messages dialog.
3. Select **fault** from the Type drop-down list.

4. Select **ns1:UnknownUser** from the Message drop-down list.
5. Click **Add** to add the fault message to the Operation Messages table.
6. Click **Apply** and then **OK** to close this dialog.
7. Click **OK** to close the Edit Port Type dialog and return to the Artix Designer where you can review the contents of the WSDL file and confirm that the `sayHi` operation now includes a fault element.

Recreating the SOAP binding

Now you need to recreate the SOAP binding and service and port definitions. To create the new binding, complete the following steps:

1. Select the **HelloWorldGuiTutorial** icon from the Designer Tree and select **Resource | New | Binding** to display the New Binding wizard.
2. Select the **Add to existing WSDL "HelloWorldGuiTutorial"** radio button and click **Next** to display the Select Binding Type panel.
3. Select the **SOAP** radio button and click **Next** to display the Select Port Type panel.
4. Select **GuiTutorialPT** from the Port Type drop-down list. Note that a suggested name is already entered in the Binding Name field. You can change this entry; the only requirement is that each binding in the WSDL file, if you create multiple bindings, have a unique Binding Name.
5. Select **document** from the Style list, and select **literal** from the Use list.
6. Click **Next** to display the Edit Binding panel.
7. Select the **sayHi** icon representing your operation and review the binding details. Click **Next** to display the View Summary panel where you can review the new content that will be added to the WSDL file.
8. Click **Finish** to close this wizard and return to the Artix Designer.

Creating the new Service

To create a new service, complete the following steps:

1. Select the **HelloWorldGuiTutorial** icon form the Designer Tree and click the WSDL tab, where you can review the contents of the WSDL file.
2. Once again, select the **HelloWorldGuiTutorial** icon and select **Resource | New | Service** to display the New Service wizard.
3. Select the **Add to existing WSDL "HelloWorldGuiTutorial"** radio button and click **Next** to display the Define Service panel.

4. Type **`HelloWorldService`** in the Name field and click **Next** to display the Define Port panel.
5. Type **`HelloWorldPort`** in the Name field and select **GuiTutorialPT_SOAPBinding** from the Binding drop-down list.
6. Click **Next** to display the Define Extensor Properties panel.
7. Select **soap** from the Transport Type drop-down list and enter **http://localhost:9000** as the Location value. This is the only required entry, and you can select any port screen from the list.
8. Click **Next** to display the Port Summary panel where you can review the new content that will be added to the WSDL file.
9. Click **Finish** to close this wizard and return to the Artix Designer.

Generating the application code

You will use the Artix Designer to generate starting point code for the application. Since the Deployment Profiles and Deployment Bundles have already been created, you only need to regenerate the starting point code and then re-implement and recompile the applications.

Repeat the steps described in "Generating the C++ and Java Code" on page 116.

Completing the code

In the C++ implementation class, you need to complete the `sayHi` method. You modify the previous coding so that the `UnknownUserException` is thrown unless the value of `InPart` is `Artix User`.

```
if (InPart.getvalue() != "Artix User")
{
  FaultDetails faultData;
  faultData.setFaultMsg("User unknown to me");
  faultData.setFaultID(200);

  UnknownUserException ex;
  ex.settheFault(faultData);

  throw ex;
}
OutPart.setvalue("Hello " + InPart.getvalue());
```

In the client application `GuiTutorialPTClientSample.cxx` file, remove the comment delimiters and replace with the following code:

```
GuiTutorial::InParameter    InPart;
GuiTutorial::OutParameter   OutPart;

// Set user name to command line parameter
InPart.set_value("Artix User");
if (argc > 1)
{
  InPart.setvalue(argv[1]);
}
// Alternative code to set user name
/*
argc > 1 ? InPart.set_value(argv[1]) : \
             InPart.setvalue("Artix User");
*/
client.sayHi ( InPart,  OutPart);
cout << "sayHi returned: " + OutPart.getvalue() << endl;
```

Also add a new `catch{}` block before the existing `catch{}` block.

```
catch(UnknownUserException& ex)
{
  FaultDetails& fd = ex.gettheFault();
  cout << "Error Message: " << fd.getFaultMsg() << endl;
  cout << "Error ID: " << fd.getFaultID() << endl;
  return -1;
}
```

In the Java application, the client mainline produced by the `wsdltojava` utility already includes a `catch{}` block to handle the `FaultDetailsException`.

```
try {
    if ("sayHi".equals(args[0]))
    {
        . . .
    }
catch ( com.iona.FaultDetailsException ex )
    {
        System.out.println
          ("Exception: com.iona.FaultDetailsException
            has Occurred.");
        ex.printStackTrace();
    }
```

You need to add code to the `GuiTutorialPTImpl.java file`, providing an implementation for the `sayHi` method that throws the `FaultDetailsException`.

```
public String sayHi(String inPart) throws FaultDetailsException
{
    String _return = null;
    if (inPart.equals("Artix User"))
    {
        _return = "Hello " + inPart;
    }
    else
    {
        FaultDetailsException fd = new FaultDetailsException
            ("User unknown to me", 200);
        throw fd;
    }
    return _return;
}
```

Building the application

Now that you have completed coding, you can build the application. Repeat the steps described in "Compiling the Applications" on page 126.

Running the application

Run the applications as described in "Running the Application" on page 128.

When running the C++ client, Artix User is supplied to the Web service when you do not provide a name on the command line. If you provide a name on the command line, that name is passed to the Web service.

When running the Java client supply "Artix User" or another name as the required parameter (quotation marks around Artix User are important).

Note that the server simply throws the exception, which the client applications catch and display if the name is not Artix User.

Glossary

B

Binding

A binding associates a specific transport/protocol and data format with the operations defined in a `<portType>`.

Bus

See Service Bus

Bridge

A usage mode in which Artix is used to integrate applications using different payload formats.

C

Collection

A group of related WSDL contracts that can be deployed as one or more physical entities such as Java, C++, or CORBA-based applications. It can also be deployed as a switch process.

Connection

An established communication link between any two Artix endpoints.

Contract

An Artix contract is a WSDL file that defines the interface and all connection-related information for that interface. A contract contains two components: logical and physical.

The logical contract defines things that are independent of the underlying transport and wire format, and is specified in the `<portType>`, `<operation>`, `<message>`, `<type>`, and `<schema>` WSDL elements.

The physical contract defines the payload format, middleware transport, and service groupings, and the mappings between these things and `portType` 'operations.' The physical contract is specified in the `<port>`, `<binding>` and `<service>` WSDL elements.

CORBA

CORBA (Common Object Request Broker Architecture) defines standards for interoperability and portability among distributed objects, independently of the language in which those objects are written. It is a robust, industry-accepted standard from the OMG (Object Management Group), deployed in thousands of mission-critical systems.

CORBA also specifies an extensive set of services for creating and managing distributed objects, accessing them by name, storing them in persistent stores, externalizing their state, and defining ad hoc relationships between them. An ORB is the core element of the wider OMG framework for developing and deploying distributed components.

D

Deployment Mode

One of two ways in which an Artix application can be deployed: Embedded and Standalone. An embedded-mode Artix application is linked with Artix-generated stubs and skeletons to connect client and server to the service bus. A standalone application runs as a separate process in the form of a daemon.

E

Embedded Mode

Operational mode in which an application creates a Service Access Point, either by invoking Artix APIs directly, or by compiling and linking Artix-generated stubs and skeletons to connect client and server to the service bus.

Endpoint

The runtime deployment of one or more contracts, where one or more transports and its marshalling is defined, and at least one contract results in a generated stub or skeleton (thus an endpoint can be compiled into an application). Contrast with Service.

Extensible Style Sheet Transformation

A set of extensions to the XML style sheet language that describes transformations between XML documents. For more information see the XSLT specification.

H

Host

The network node on which a particular service resides.

M

Marshalling Format

A marshalling format controls the layout of a message to be delivered over a transport. A marshalling format is bound to a transport in the WSDL definition of a port and its binding. A binding can also be specified in a logical contract port type, which allows for a logical contract to have multiple bindings and thus multiple wire message formats for the same contract.

Message

A WSDL message is an abstract definition of the data being communicated. Each part of a message is associated with defined types. A WSDL message is analogous to a parameter in object-oriented programming.

O

Operation

A WSDL operation is an abstract definition of the action supported by the service. It is defined in terms of input and output messages. An operation is loosely analogous to a function or method in object-oriented programming, or a message queue or business process.

P

Payload Format

The on-the-wire structure of a message over a given transport. A payload format is associated with a port (transport) in the WSDL file using the binding definition.

Port Type

A WSDL port type is a collection of abstract operations, supported by one or more endpoints. A port type is loosely analogous to a class in object-oriented programming. A port type can be mapped to multiple transports using multiple bindings.

Protocol

A protocol is a transport whose format is defined by an open standard.

R

Resource

A resource can be one of two things:

- A WSDL file that defines the interface of your Artix solution
- A Schema that defines one or more types. This schema can be a stand alone resource or it can define the types within a WSDL contract.

Resources are contained within collections. There can be one or more resources in a collection, and the resources can either be specific to that collection, or shared across several collections (shared resources).

Resources are created either from scratch using the Resource Editor wizards and dialogs to define them, or are based on an existing files. For example, you can use the Designer to convert an IDL file into WSDL.

Resource Editor

A GUI tool used for editing Artix contracts. It provides several wizards for adding services, transports, and bindings to an Artix contract.

Routing

The redirection of a message from one WSDL binding to another. Routing rules are specified in a WSDL contract and apply to both endpoints and standalone services. Artix supports port-based routing and operation-based routing defined in WSDL contracts. Content-based routing is supported at the application level.

Router

A usage mode in which Artix redirects messages based on rules defined in an Artix contract.

S

Service

An Artix service is an instance of an Artix runtime deployed with one or more contracts, but with no generated language bindings. The service has no compile-time dependencies. A service is dynamically configured by deploying one or more contracts on it.

Service Access Point

The mechanism and the points at which individual service providers and consumers connect to the service bus.

Service Bus

The set of service providers and consumers that communicate via Artix. Also known as an Enterprise Service Bus.

SOAP

SOAP is an XML-based messaging framework specifically designed for exchanging formatted data across the Internet. It can be used for sending request and reply messages or for sending entire XML documents. As a protocol, SOAP is simple, easy to use, and completely neutral with respect to operating system, programming language, or distributed computing platform.

Standalone Mode

An Artix instance running independently of either of the applications it is integrating. This provides a minimally invasive integration solution and is fully described by an Artix contract.

Switch

A usage mode in which Artix connects applications using two different transport mechanisms.

System

A collection of services and transports.

T

Transport

An on-the-wire format for messages.

Transport Plug-in

A plug-in module that provides wire-level interoperation with a specific type of middleware. When configured with a given transport plug-in, Artix will interoperate with the specified middleware at a remote location or in another process. The transport is specified in the `<port>` element of an Artix contract.

Type

A WSDL data type is a container for data type definitions that is used to describe messages (for example an XML schema).

W

Web Services Description Language

An XML-based specification for defining Web services. For more information see the WSDL specification.

Workspace

The Artix Workspace defines the structure of your Artix solution. It is the first thing you need to create when using the Artix Designer, and all of the solution's components are included within it.

A workspace typically has one or more collections, which in turn contain resources that define your solution's interface. A workspace also contains shared resources that are common across one or more collections.

WSDL

WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information.

A WSDL document defines services as collections of network endpoints, or ports. In WSDL, the abstract definition of endpoints and messages is separated from their concrete network deployment or data binding formats. This allows the reuse of abstract definitions: messages, which are abstract descriptions of the data being exchanged, and port types which are abstract collections of operations. The concrete protocol and data format specifications for a particular port type constitutes a reusable binding. A port is defined by associating a network address with a reusable binding, and a collection of ports define a service. Hence, a WSDL document uses the following elements in the definition of network services:

- **Types**—a container for data type definitions using some type system (such as XSD).
- **Message**—an abstract, typed definition of the data being communicated.
- **Operation**—an abstract definition of an action supported by the service.
- **Port Type—**an abstract set of operations supported by one or more endpoints.
- **Binding—**a concrete protocol and data format specification for a particular port type.

- **Port**—a single endpoint defined as a combination of a binding and a network address.
- **Service**—a collection of related endpoints.

Source: Web Services Description Language (WSDL) 1.1. W3C Note 15 March 2001. (http://www.w3.org/TR/wsdl)

X

XML

XML is a simpler but restricted form of SGML (Standard General Markup Language). The markup describes the meaning of the text. XML enables the separation of content from data. XML was created so that richly structured documents could be used over the web.

XSD

XML Schema Definition (XSD) is the language used to define an XML Schema. The XML Schema defines the structure of an XML document.

In Artix, a schema can be a standalone resource within a collection, or it can be used as an import do define the types within a WSDL contract.

Index

T

V

W

X