Artix™ ESB

Getting Started with Artix

Version 5.0, July 2007

Making Software Work Together™

Updated: July 26, 2007

Contents

List of Figures

Preface

What is Covered in This Book

Getting Started with Artix provides an introduction to IONA's Artix ESB technology. It gives a brief overview of the architecture and functionality of Artix, and an introduction to Web Services Description Language (WSDL).

This book takes you through the process of creating a WSDL file and generating starting point code in both C++ and Java using the Artix Designer development tool.

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

Who Should Read This Book

Getting Started with Artix is for anyone who needs to understand the concepts and terms used in IONA's Artix product.

Organization of This Book

This book contains conceptual information about Artix and WSDL:

- Chapter 1, "Introduction" on page 9 introduces the Artix product and the types of problems it is designed to solve, and provides an introduction walkthrough of the Artix documentation library.
- Chapter 2, "Artix ESB Concepts" on page 21 explains the main concepts used in Artix.
- Chapter 3, "Artix Designer Introduction and Tutorial" on page 29 explains the basics of using Artix Designer to edit Artix contracts and generate project code from a WSDL contract.
- Appendix A, "Understanding WSDL" on page 85 explains the basics of WSDL.

The Artix Documentation Library

For information on the organization of the Artix library, the document conventions used, and finding additional resources, see Using the Artix Library.

CHAPTER 1

Introduction

This chapter introduces the main features of Artix ESB.

In this chapter

This chapter discusses the following topics:

What is Artix ESB?

Overview

Artix ESB is an extensible enterprise service bus. It provides the tools for rapid application integration that exploits the middleware technologies and the products already present within your enterprise.

The approach taken by Artix ESB relies on existing Web service standards and extends these standards to provide rapid integration solutions that increase operational efficiencies, capitalize on existing infrastructure, and enable the adoption or extension of a service-oriented architecture (SOA).

Web services and SOAs

The information services community generally regards Web services as application-to-application interactions that use SOAP over HTTP.

Web services have the following advantages:

- The data encoding scheme and transport semantics are based on standardized specifications.
- The XML message content is human readable.
- The contract defining the service is XML-based and can be edited by any text editor.
- They promote loosely coupled architectures.

Service-oriented architectures take the Web services concept and extend it to the entire enterprise. Using a service-oriented architecture, your infastructure becomes a collection of loosely coupled services. Each service becomes an endpoint defined by a contract written in Web Services Description Language (WSDL). Clients, or service consumers, can then access the services by reading a service's contract.

Artix and services

Using IONA's proven *Adaptive Runtime Technology* (ART), Artix extends the Web service standards to include more than just SOAP over HTTP. Thus, Artix allows organizations to define their existing applications as services without worrying about the underlying middleware. It also provides the ability to expose those applications across a number of middleware technologies without writing any new code.

Artix also provides developers with the tools to write new applications in C++ or Java that can be exposed as middleware-neutral services. These tools aid in the definition of the new service in WSDL and in the generation of stub and skeleton code.

Just like the WSDL contracts used to define a service, the code that Artix generates adheres to industry standards.

Benefits of Artix

Artix ESB's extensible nature provides a number of benefits compared to other ESB products and older enterprise application integration (EAI) products. Chief among these is its speed and flexibility. In addition, Artix ESB provides enterprise levels of service such as session management, service discovery, security, and cross-middleware transaction propagation.

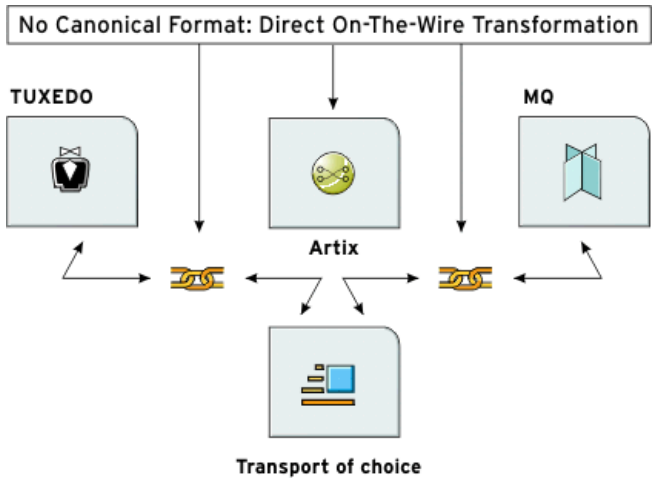
EAI products typically use a proprietary, canonical message format in a centralized EAI hub. When the hub receives a message, it transforms the message to this canonical format and then transforms the message to the format of the target application before sending it to its destination. Each application requires two adapters that are typically proprietary and that translate to and from the canonical format.

By contrast, the Artix ESB bus does not require a hub architecture, nor does it use any intermediate message format. When a message is received by the bus, it is transformed directly into the target application's message format.

Because Artix ESB uses a standardized means of defining its services, the plug-ins used to connect applications to the bus are reusable.

Figure 1 shows an example Artix ESB integration between BEA Tuxedo and IBM WebSphere MQ.

Figure 1: *Artix ESB High-Performance Architecture*



Because Artix ESB is built on top of ART, it is modular in nature. This means that it is highly configurable and that it is easily extendable. You can configure Artix ESB to only load the pieces you need for the functionality you require. If Artix ESB does not provide a transport or message format you need, you can easily develop your own plug-in, extend the contract definitions, and configure Artix to load it.

Using Artix ESB

There are two ways to use Artix ESB in your enterprise:

- You can use Artix ESB to develop new applications using the Artix Application Programming Interface (API). In this situation, developers generate Artix stubs and skeletons from an Artix contract, and Artix becomes a part of your development environment.
- You can use the Artix bus to integrate two existing applications, built on different middleware technologies, into a single application. In this situation, developers simply create an Artix contract defining the integration of the systems. In most cases, no new code is needed.

Becoming proficient with Artix ESB

To become an effective Artix ESB developer you need an understanding of the following:

1. The syntax for WSDL files and the Artix ESB extensions to the WSDL specification.
2. The relationship between Artix WSDL extensions, ART plug-ins, and setting configuration entries.
3. The Artix APIs that you can use in your application.
4. Artix Designer, a GUI tool that enables you to write, generate, and edit WSDL files, and to generate, compile, and run code.

This book introduces these four concepts. The other books in the Artix documentation library covers the same technologies in greater detail.

Artix ESB features

Artix ESB includes the following unique features:

- Support for multiple transports and message data formats
- C++ and Java development
- Message routing
- Cross-middleware transaction support
- Asynchronous Web services
- Deployment of services as plug-ins via the Artix container
- Role-based security, single sign-on, and security integration
- Session management and stateful Web services
- Look-up services
- Load-balancing
- High-availability service clustering
- Integration with EJBs
- Easy-to-use development tools
- Support for Microsoft .NET
- Integration with enterprise management tools such as IBM Tivoli and BMC Patrol
- Support for XSLT-based message transformation
- No need to hard-code WSDL references into applications

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, WebSphere MQ and Tuxedo are transports, while HTTP and IIOP are protocols.

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

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

Supported payload formats

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

- CORBA Common Data Representation (CDR)
- G2++
- Fixed record length (FRL)
- SOAP
- Pure XML
- Tagged (variable record length)
- TibrvMsg (a TIBCO Rendezvous format)
- Tuxedo's Field Manipulation Language (FML)

Artix for z/OS

Artix for z/OS allows you to design, create, and deploy a variety of enterprise integration solutions for the mainframe. These solutions include:

- Non-intrusively exposing existing mainframe applications to the network as Web services and CORBA objects, with no need to recode the mainframe applications.
- Developing new mainframe-based Web service applications from WSDL definitions.

An application can be exposed as both a Web service and a CORBA object that can accept client requests via SOAP over HTTP or HTTPS, SOAP over WebSphere MQ, or IIOP over TCP/IP. Thus, Artix for z/OS enables you to transform basic mainframe applications into true multi-protocol applications that are accessible throughout the entire enterprise.

Artix for z/OS is delivered in separate packages, as follows:

- Artix for z/OS is a separate add-on package that provides the on-host mainframe components for development and deployment of Artix services on the mainframe.
- The Artix for z/OS off-host components are included with Artix ESB for the Windows, Linux, and Solaris platforms.

For more information on mainframe support in Artix, see the documentation for Artix for z/OS at http:/www.iona.com/support/docs/index.xml.

Artix Orchestration

Artix Orchestration is an add-on kit that must be installed into an existing Artix ESB installation, as follows:

- The Artix Orchestration add-on for all supported operating systems provides support for the Artix Orchestration BPEL engine.
- The Artix Orchestration add-on for Windows, Linux, and Solaris integrates orchestration development tools into Artix Designer.

The installation requirements are further described in the Artix Orchestration Installation Guide.

Artix Orchestration adds support for designing an orchestrated set of Web services using the standard Business Process Execution Language (BPEL), and for integrating your orchestrated set of services into your Artix environment.

Artix Orchestration adds the following features to Artix:

- BPEL Designer, integrated into Eclipse alongside Artix Designer
- Artix Orchestration and Artix Orchestration Debug perspectives for Eclipse
- An Artix Orchestration BPEL server for hosting and managing deployed BPEL processes
- A Web-based Administration Console for the Artix Orchestration server
- A persistent storage option for the Artix Orchestration server
- Demonstration code

- Documentation embedded as Eclipse Help
- An Artix Orchestration tutorial

Solving Problems with Artix ESB

Overview

Artix ESB allows you to solve problems arising from the integration of existing back-end systems using a service-oriented approach. Artix ESB allows you to develop new services using C++ or Java, and to retain all of the enterprise levels of service that you require.

There are three phases to an Artix ESB project:

1. The design phase, where you define your services and define how they are integrated using Artix contracts.
2. The development phase, where you write the application code required to implement new services.
3. The deployment phase, where you configure and deploy your Artix solution.

Design phase

In the design phase, you define the logical layout of your system in an Artix contract. The logical or abstract definition of a system includes:

- the services that it contains
- the operations each service offers
- the data the services will use to exchange information

Once you have defined the logical aspects of your system, you then add the physical network details to the contracts.

The physical details of your system include the transports and payload formats used by your services, as well as any routing schemes needed to connect services that use different transports or payload formats.

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

- Import existing WSDL documents
- Create Artix contracts from scratch
- Generate Artix contracts from:
  - CORBA IDL
  - A description of tagged data

- A description of fixed record length data
- A COBOL copybook
- A Java class

- Add the following bindings to an Artix contract:
  - CORBA
  - Fixed record length
  - SOAP
  - Tagged data
  - XML

Development phase

You must write Artix application code if your solution involves creating new applications or a custom router, or involves using the Artix session management feature. The first step in writing Artix code is to generate client stub code and server skeleton code from the Artix contracts that you created in the design phase. You can generate this code using Artix Designer or the Artix command-line tools.

After you have generated the client stub code and server skeleton code, you can develop the code that implements the business logic you require. For most applications, Artix-generated code allows you to stick to using standard C++ or Java code for writing business logic.

Artix Designer is integrated with the open-source Eclipse application framework, but you are not required to use Eclipse for the whole project. Once the stub code is generated, you can switch to your favorite development environment to develop and debug the application code.

Artix ESB also provides advanced APIs for directly manipulating messages, for writing message handlers, and for other advanced features your application might require. These can be plugged into the Artix runtime for customized processing of messages.

Deployment phase

In the deployment phase, you configure the Artix runtime to fine-tune the Artix bus for your new Artix system. This involves modifying the Artix configuration files and editing the Artix contracts that describe your solution to fit the exact circumstances of your deployment environment.

This phase also includes the managing of the deployed system. This might involve, for example, using an enterprise management tool such as Tivoli along with the Artix command interface. These tools allow you to further fine-tune your system.

CHAPTER 2

Artix ESB Concepts

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

In this chapter

This chapter discusses the following topics:

The Artix ESB Runtime Components

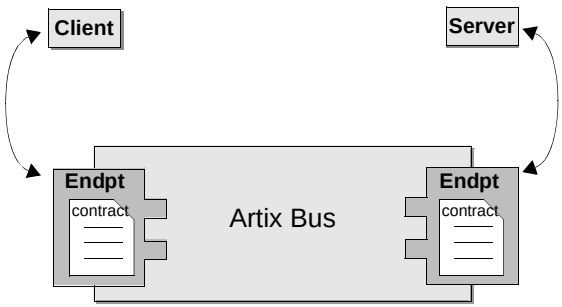
How it fits together

Artix ESB is comprised of a group of components that are built on the Adaptive Runtime Technology (ART) platform:

- The Artix Bus is at the core of Artix, and provides the support for various transports and payload formats.
- Artix Contracts describe your applications in such a way that they become services that can be deployed as Artix Endpoints.
- Artix Services include a number of advanced services, such as the locator and session manager. Each Artix service is defined with an Artix contract and can be deployed as an Artix endpoint.

Figure 2 illustrates how the Artix ESB elements fit together.

Figure 2: *Artix ESB Runtime Components*



Plugability

Because Artix ESB is built on ART, all Artix services are implemented as plug-ins. You can also deploy your own services as plug-ins. This means that you can host any Artix service either as a standalone application or as a plug-in to another Artix application.

Each separate service, regardless of how it is deployed, becomes a separate endpoint.

The Artix Bus

Overview

The Artix bus is at the heart of the Artix ESB architecture. It is the component that hosts the services that you create and connects your applications to those services.

The bus is also responsible for translating data from one format into another. This translation process works as follows:

1. Reader plug-ins accept incoming data in one format.
2. The Artix bus directly translates the data into another format.
3. Writer plug-ins write the data back out to the wire in the new format.

In this way, Artix ESB enables all of the services in your company to communicate, without needing to communicate in the same way. It also means that clients can contact services without understanding the native language of the server handling requests.

Benefits

While other products provide some ability to expose applications as 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 the application's native communication protocol into any of the other supported protocols.

For example, by deploying an Artix instance with a SOAP-over-WebSphere MQ endpoint and a SOAP-over-HTTP endpoint, 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 is 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 overhead and increases the speed at which messages are transmitted.

Artix Endpoints

Overview

An Artix endpoint is the connection point at which a service or a service consumer connects to the Artix bus. Endpoints are described by a contract describing the services offered and the physical representation of the data on the network.

Reconfigurable connection

An Artix endpoint provides an abstract connection point between applications, as shown in Figure 2 on page 22. The benefit of this abstract connection is that it allows you to change the underlying communication mechanism without recoding any of your applications. You only need to modify the contract describing the endpoint.

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 endpoint'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

Artix contracts are written in WSDL. In this way, a standard language is used to describe the characteristics of services and their associated Artix endpoints. By defining characteristics such as service operations and messages in an abstract way—independent of the transport or protocol used to implement the endpoint—these characteristics can be bound to a variety of protocols and formats.

Artix ESB 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 simplest Artix contract defines a single pair of systems with a shared interface, payload format, and transport. Artix contracts can also define very complex integration scenarios.

WSDL elements

Understanding Artix contracts requires some familiarity with WSDL. The key WSDL elements are as follows:

WSDL types provide data type definitions used to describe messages.

A WSDL message is an abstract definition of the data being communicated. Each part of a message is associated with a defined type.

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 logical contract includes the `types`, `message`, `operation`, and `portType` elements of the WSDL file.

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

The physical contract includes the `binding`, `port`, and `service` elements of the WSDL file.

Artix Services

Overview

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

- Container
- Locator
- Session manager
- Transformer
- Accessing contracts and references

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

Container

The Artix container provides a consistent mechanism for deploying and managing Artix services. It allows you to write Web service implementations as Artix plug-ins and then deploy your services into the Artix container.

Using the container eliminates the need to write your own C++ or Java server mainline. Instead, you can deploy your service by simply passing the location of a generated deployment descriptor to the Artix container's administration client.

IONA strongly recommends that all new client and server Artix implementations be implemented and deployed in an Artix container.

Locator

The Artix locator provides service look-up and load balancing functionality to an Artix deployment. It isolates service consumers from changes in a service's contact information.

The Artix WSDL contract defines how the client contacts the server, and contains the address of the Artix locator. The locator provides the client with a reference to the server.

Servers are automatically registered with the locator when they start, and service endpoints are automatically made available to clients without the need for additional coding.

Session manager

The Artix session manager is a group of plug-ins that work together to manage the number of concurrent clients that access a group of services. This allows you to control how long each client can use the services in the group before having to check back with the session manager.

In addition, the session manager has a pluggable policy callback mechanism that enables you to implement your own session management policies.

Transformer

The Artix transformer provides Artix ESB 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 without rewriting the underlying applications.

Accessing contracts and references

Accessing contracts and references in Artix ESB refers to enabling client and server applications to find WSDL service contracts and references. Using the techniques and conventions of Artix avoids the need to hard code WSDL into your client and server applications.

For more information

For more information on Artix services, see *Configuring and Deploying Artix Solutions*.

CHAPTER 3

Artix Designer Introduction and Tutorial

This chapter introduces 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

Overview

Artix Designer is a GUI development tool that ships as a series of plug-ins to the Eclipse platform. Eclipse is an open source development platform and application framework for building software, as described at eclipse.org.

Artix Designer enables you to write and edit the WSDL files that describe Artix resources and their integration, and to generate starting point code for a Web service. Artix Designer also includes perspectives that enable you to work with Artix for z/OS and Artix database projects, and to manage deployed Artix services.

Generating WSDL

Artix Designer features a number of wizards that enable you to create WSDL files based on:

- CORBA IDL files
- Java classes
- EJB session beans
- XSD schemas
- Fixed record-length data
- Tagged data
- COBOL copybook files

The WSDL editor

Artix Designer's WSDL editor is integrated with its code generation tools, and incorporates a thorough understanding of the Artix extensions to the WSDL standard.

For example, Artix Designer automatically adds the required namespace declarations and prefix definitions when you build Artix applications that involve Artix-extended data marshalling schemas, transport protocols, or routing.

The Artix Designer WSDL editor provides a number of wizards that take you through the process of creating and editing `type`, `message`, `portType`, `binding`, `service`, and `route` elements in your WSDL files.

Generating code

Artix Designer's code generation tool incorporates the same technology as the Artix command-line tools. This allows Artix Designer to generate starting point code from your WSDL files in C++, JAX-RPC Java, and JAX-WS Java.

Integration with the Eclipse Java Development Tools (JDT) and C/C++ Development Tools (CDT) means that any code you create is compiled automatically after you generate it, and is recompiled when you make any changes to your source.

Note: The **Build Automatically** option must be enabled in the Eclipse **Project** menu for code to be compiled automatically.

The Artix code generator allows you to create a variety of code generation configurations, which you can save and reuse. For example, you can create configurations for:

- Client and server applications
- Artix router applications
- CORBA IDL
- Artix service plug-ins
- Container applications for hosting service plug-ins

See "Tutorial: Generating Code" on page 66 for an example of the Artix code generator at work.

Artix for z/OS off-host components

The off-host components of the Artix for z/OS product are included with the base configuration of Artix Designer. These off-host components are designed to be used in conjunction with the mainframe components of Artix for z/OS, which are separately licensed.

Launching Artix Designer

To launch Artix Designer in Windows, select **Start|(All) Programs|IONA|Artix *version*|Artix Designer**.

To launch Artix Designer in Linux or Solaris, run:

```
InstallDir/tools/eclipse/eclipse
```

The Eclipse platform launches with the Artix Designer plug-ins loaded.

Note: You can install Artix Designer into an existing Eclipse 3.2 installation, as described in the Artix Installation Guide.

Artix-related perspectives

In the Eclipse development framework, a perspective is a predefined layout of the windows, views, menus, and tools in the Eclipse window. The following Artix-related perspectives are shipped with Artix Designer:

- The **Artix** perspective is associated with basic Web services projects, as well as CORBA and EJB projects.
- The **Artix Database** perspective is associated with Artix database projects.
- The **Artix Mainframe** perspective is associated with Artix for z/OS projects.

The optional Artix Orchestration add-on package installs two more perspectives: **Artix Orchestration** and **Artix Orchestration Debug**.

Artix Designer also includes three perspectives from the client side of Artix Registry/Repository: **Artix Repository Explorer**, **Artix Repository Governance**, and **Artix Repository Infrastructure**. If you have an Artix Repository database at your site, you can log in to it from one of the Registry/Repository perspectives.

The Artix perspective

When you create a new Artix Designer Web services project, Eclipse automatically switches to the Artix perspective.

The Artix perspective provides you with the tools that you need to develop an Artix project in Eclipse. It includes the following features:

- The Artix toolbar
- The Navigator view
- The Outline view

Artix Designer project types

In Eclipse, all development is performed within a project. When you create a new project in the Artix perspective, Artix Designer offers a choice between the following project creation wizards:

- C++/JAX-RPC:
  - Basic Web services project.

- CORBA Web services project.
- Database Web services project.
- Empty Web Services Project.
- Web services projects from EJB.

- Java JAX-WS:
  - Database Web services project.
  - Empty Web Services Project.
  - Java first project.
  - WSDL first project.
- Mainframe.
  - CORBA Web services project from IDL.
  - Web services project from application.
  - Web services project from DB2.
  - Web services project from deployment descriptors.
  - Web services projects from WSDL.

A CORBA Web services project creates a WSDL file and a router configuration based on a CORBA IDL data source.

Artix Designer project templates

The project creation wizards other than **Basic** have preselected project templates. When you select one of these wizards, you also select its associated project template.

When you create a new project starting with the **Basic Web Services Project** wizard, Artix Designer prompts you to specify a template. The template sets up files and a directory structure for you.

The **Empty Project** template creates only a project folder and an Eclipse `.project` file. You must import or link to an existing WSDL file, or create a new one, in order to have a starting point for generating code.

The other project templates create all the starting point code and configuration information needed for your Web services application.

Note: When using a template other than **Empty**, you must have a valid WSDL file prepared in advance.

Artix Designer provides the following project templates:

- Empty project
- Artix router
- C++ client
- C++ client and plug-in
- C++ client and server
- C++ plug-in
- C++ server
- Java client
- Java client and plug-in
- Java client and server
- Java plug-in
- Java server

Using the Artix toolbar

The Artix toolbar gives you quick access to the primary Artix Designer functionality. It contains the following buttons:

Table 1: *Artix Designer Toolbar Buttons*

Button		Description
	Standard	Re-run the last-run Artix Tools configuration.[a]
		Import Artix demos into Artix Designer.
	Artix for z/OS	Export Artix for z/OS project (active in the Artix for z/OS perspective).
		Create a BIM file from the current WSDL file (active in the Artix for z/OS perspective).
		Validate selected WSDL for Artix for z/OS (active in the Artix for z/OS perspective).

Table 1: *Artix Designer Toolbar Buttons (Continued)*

Button		Description
	Standard	Add `import` element to currently selected WSDL file.
		Add `type` element to currently selected WSDL file.
		Add `message` element to currently selected WSDL file.
		Add `portType` element to currently selected WSDL file.
	Standard	Add `binding` element to currently selected WSDL file.
		Add `service` element to currently selected WSDL file.
		Add `route` element to currently selected WSDL file.
	Standard	Define an access control list (ACL) to apply to a port type or an operation.
		CORBA-enable the current WSDL file after it has a fully defined interface.
		SOAP-enable the current WSDL file after it has a fully defined interface.

a. If a code generation configuration already exists, clicking this button launches the last-used configuration. Click the down arrow next to this button to run other configurations, or to open the Artix Tools dialog.

Cheat sheets

The Eclipse environment provides an online documentation type that it calls cheat sheets. Cheat sheets are interactive tutorials that guide you step by step through common tasks.

Artix Designer ships with several Artix-related cheat sheets to help you:

- Create an Artix Designer project
- Generate a client-server application
- Create a WSDL file's logical and physical elements
- Generate code for a services plug-in and deploy it in an Artix container

Each cheat sheet lists the steps required to complete a particular task. As you progress from one step to the next, the cheat sheet automatically launches the required tools for you.

Artix Designer also provides cheat sheets to help you learn to use the Artix Container Admin perspective, the Artix for z/OS off-host components, and Artix database services.

To view the available Artix Designer cheat sheets, select **Help|Cheat Sheets**.

Online help

Help on Artix Designer is available from within the Eclipse online help system.

Select **Help|Help Contents** to view the Eclipse Help. The Artix Designer Help section is listed in the table of contents frame on the left.

In addition, you can access context-sensitive Help from within the Artix Designer wizards and the Artix Tools window by pressing **F1**.

Artix Designer Tutorial

Overview

This section provides an overview of the tutorial on using Artix Designer to create, build, and run Web services client and server applications: for JAX-WS Java, for JAX-RPC Java, and for C++.

Tutorial stages

The stages of the tutorial, and of most Artix Web services projects, are the following:

1. Create a new Eclipse project to contain the files for each application.
   "Tutorial: Creating New Projects" on page 38.
2. Import or create a skeleton WSDL file.
   "Tutorial: Creating a Blank WSDL File" on page 40.
3. If you are creating a WSDL file, populate it with the necessary parts to define your intended Web service. This tutorial creates a hello world service that responds with a greeting to messages sent.
   "Tutorial: Defining the WSDL Elements" on page 43.
4. Generate skeleton code for your client and server applications.
   "Tutorial: Generating Code" on page 66.
5. Run and test your applications.
   "Tutorial: Running the Applications" on page 75.

Tutorial: Creating New Projects

Overview

This section begins a tutorial on creating a simple Web service client and server using Artix Designer. This section walks you through the steps to create three empty, basic Web services projects, one for JAX-WS Java, one for JAX-RPC Java and one for C++.

The Web service that the tutorial creates is a simple *hello world* service that responds to a particular message with a response greeting. The client application sends the message to the service.

Enable the Navigator view

If Eclipse does not already show the Artix Navigator view, enable it now by selecting **Window|Show View|Navigator**.

Create the JAX-WS Web services project

Create a project to contain the JAX-WS version of the hello world application:

1. In Eclipse, select **File|New|Project**.
2. In the **New Project** dialog box, select **Artix|Java JAX-WS|Empty Web Services Project** and click **Next**.
3. In the **New Basic Web Service Project** panel:
   i. Type `JaxWsHello` in the **Project name** text box.
   ii. Leave the **Use default location** checkbox checked (unless you want to store the project somewhere other than your current Eclipse workspace).
   iii. Click **Finish**.

If is not currently open, Eclipse automatically switches to the Artix perspective. An empty project named **JaxWsHello** is added to the Navigator view in Eclipse.

Create the JAX-RPC Web services project

Create a project to contain the JAX-RPC version of the hello world application:

1. In Eclipse, select **File|New|Project**.
2. In the **New Project** dialog box, select **Artix|C++/JAX-RPC|Empty Web Services Project** and click **Next**.
3. In the **New Basic Web Service Project** panel:
   i. Type `JaxRpcHello` in the **Project name** text box.
   ii. Leave the **Use default location** checkbox checked (unless you want to store the project somewhere other than your current Eclipse workspace).
   iii. Click **Finish**.

An empty project named **JaxWsHello** is added to the Navigator view in Eclipse.

Create the C++ Web services project

Now create another project, by selecting **Artix|C++/JAX-RPC|Empty Web Services Project**, and name it **CppHello**.

Summary

Your Eclipse workspace now displays the following Artix projects in the Navigator view:

- **JaxWsHello**
- **JaxRpcHello**
- **CppHello**

Tutorial: Creating a Blank WSDL File

Overview

This section shows how to use Artix Designer to create a skeleton WSDL file that forms the basis of your description of the Web service portion of your application. The same WSDL file is used to generate the JAX-WS, JAX-RPC, and C++ versions of the tutorial's application.

Start with WSDL

Each Artix Designer project starts with a WSDL definition of the Web service that will accept connections for your application. For example, if you are developing only the client application for an existing Web service, you can import an existing service's WSDL file into your new project.

This tutorial aims to create both client and server components of its Web service, so we will create a new WSDL description of the service.

The first step in creating WSDL from scratch is to create an empty skeleton WSDL file. In the next step, we will populate the WSDL elements with Artix Designer wizards.

Creating an empty WSDL file

To create an empty WSDL file:

1. Make sure the Artix perspective is currently active in the Eclipse workspace.
2. Select **File|New|WSDL File**.
3. In the **WSDL File** panel, select the **JaxWsHello** project folder. This specifies where the WSDL file is to be stored.
4. In the **File name** text box, type `HelloWorld`.
5. Click **Finish**.

The `HelloWorld.wsdl` file opens in the WSDL Editor.

Linking to the WSDL file from JaxRpcHello

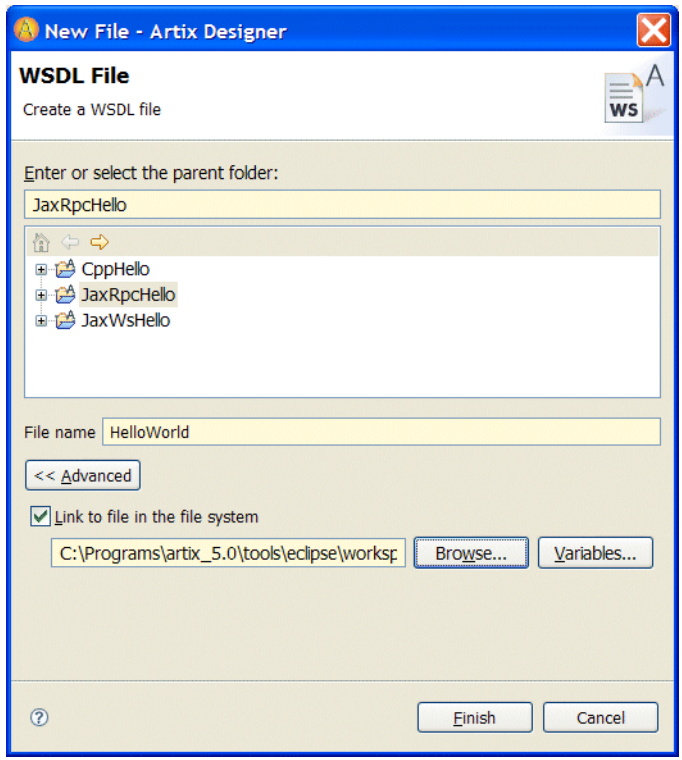
To generate the JAX-RPC client and server code from the same `HelloWorld.wsdl` WSDL file, link to the JaxWsHello project's WSDL file from the JaxRpcHello project:

1. In the Eclipse workspace, select **File|New|WSDL File**.
2. In the **WSDL File** panel, select the **JaxRpcHello** project folder.

3. In the **File name** text box, type **`HelloWorld`**.
4. Click the **Advanced** button.
5. Select the **Link to file in the file system** checkbox and click **Browse**.
6. Browse to the *`EclipseWorkspace`*`/JaxWsHello` directory, select the **HelloWorld.wsdl** file, and click **Open**.

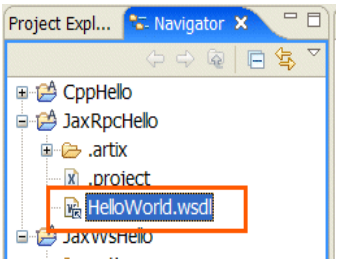
Note: You specified the location of your *`EclipseWorkspace`* directory when you first started Artix Designer. For Windows users, IONA recommends using a non-default location whose path does not contain a space, such as `C:\EclipseWS`. The default workspace location for UNIX and Linux users is `~/workspace`.

7. Click **Finish**.



The `HelloWorld.wsdl` file now appears as a link in the **JaxRpcHello** project.

Figure 3: *The JaxRpcHello Project with Link to the HelloWorld.wsdl File*



Note: When you use a link to a file (instead of a copy of the file), the same file is used by both the JaxRpcHello and JaxWsHello projects.

It is also possible to import the WSDL file into the JaxRpcHello project by selecting **File|Import**. However, this would create a separate physical file, and any changes you made to one WSDL file would not be replicated in the other.

Linking to the WSDL file from CppHello

To generate the C++ client and server code from the same `HelloWorld.wsdl` WSDL file, link to the JaxWsHello project's WSDL file from the CppHello project.

Follow the analogous instructions from "Linking to the WSDL file from JaxRpcHello" on page 40, except that this time the file link is inserted into the CppHello project.

Tutorial: Defining the WSDL Elements

Overview

Next, populate the empty WSDL file skeleton with the elements to make it a valid Artix contract. Artix Designer provides a series of wizards that allow you to create each of these elements.

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

Defining Types

Overview

The `types` element of the WSDL file contains all the data types used between the client and server.

In this simple example, we will create two `element` types of type `string`:

- `InElement`, which maps to the *in* part of the request message that you will create later.
- `OutElement`, which maps to the *out* part of the response message.

Defining element types

To define the `InElement` type:

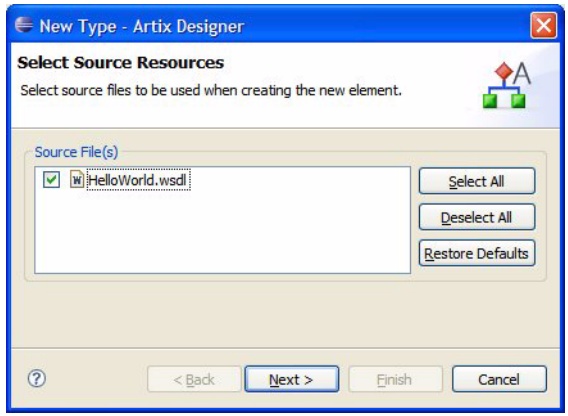
1. Open the `HelloWorld.wsdl` file from the JaxWsHello project.
2. Click the **Diagram** tab at bottom of the WSDL Editor view.
3. In the Diagram view, right-click the **Types** node.

   Note: You can also add elements to a WSDL file from the **Artix Designer** menu, or by clicking the appropriate icon in the Artix toolbar. See Table 1 on page 34 for more on the available icons.

4. Select **New Type** from the pop-up menu. The New Type wizard opens.

5. In the Select Source Resources panel, make sure **HelloWorld.wsdl** is selected in the **Source File(s)** section.

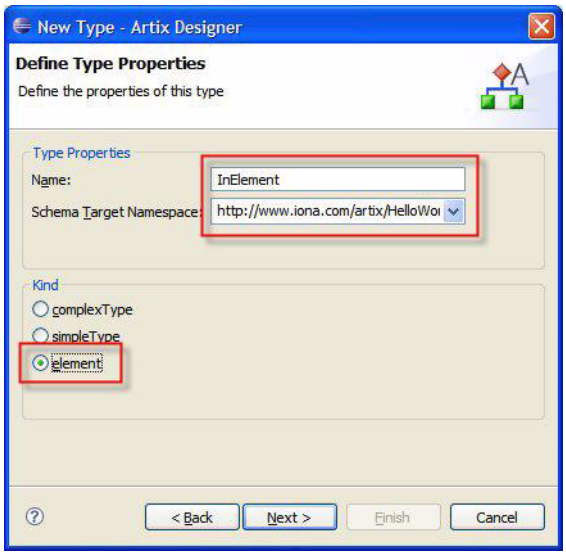
Figure 4: *The Select Source Resources Panel*



6. Click **Next** to display the Define Type Properties panel.

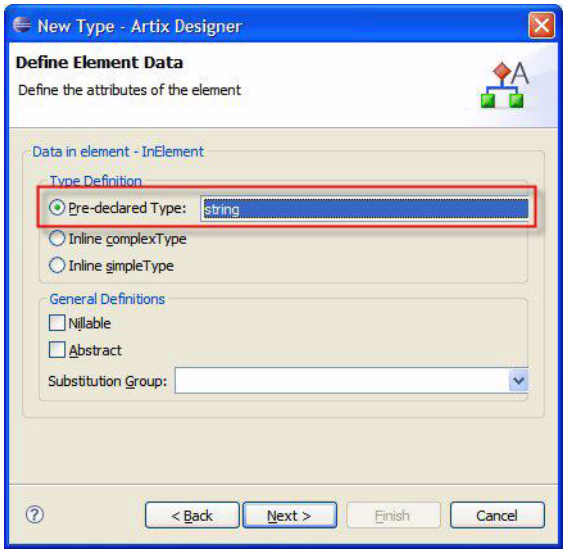
7. In the Define Type Properties panel:
   i. Type `InElement` in the **Name** text box.
   ii. Accept the default target namespace provided.
   iii. Select the **element** control.
   iv. Click **Next**.

Figure 5: *The Define Type Properties Panel*



8. In the Define Element Data panel:
   i. Select the **Pre-declared Type** button.
   ii. Select **string** from the drop-down list.
   iii. Leave the other controls blank.

Figure 6: *The Define Element Data Panel*



9. Click **Next** to display the View Type Summary panel, then click **Finish**.

To define the `OutElement` type:

1. Repeat steps 2 to 6 above.
2. In the Define Type Properties panel:
   i. Enter **`OutElement`** in the **Name** field.
   ii. Select the **element** radio button
   iii. Click **Next**.
3. In the Define Element Data panel:
   i. Select **Pre-declared Type**
   ii. Select **string** from the drop-down list.
   iii. Click **Next**.
4. In the View Type Summary panel, click **Finish**.

Save your WSDL file by selecting **File|Save** from the menu bar or right-click in the Source view and select **Save**.

Review

Click the **Source** tab at the bottom of the WSDL Editor view to look over the WSDL file created so far.

In the Outline view in the lower left of the Eclipse window, open the **Types** node. Click the name of a `types` element to jump to that element in the WSDL Editor view.

```
<types>
  <schema targetNamespace="http://www.iona.com/artix/HelloWorld"
    xmlns="http://www.w3.org/2001/XMLSchema">
    <element name="InElement" type="string"/>
    <element name="OutElement" type="string"/>
  </schema>
</types>
```

Defining Messages

Overview

Now that you have created the WSDL types, you can define the request and response messages for your Web service.

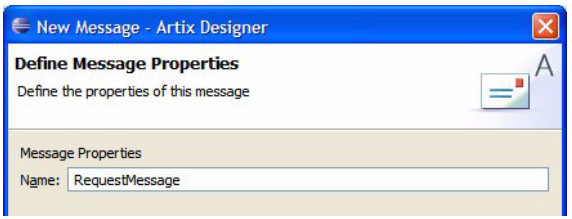
You will use your types as the message parts.

Defining messages

To define the request message:

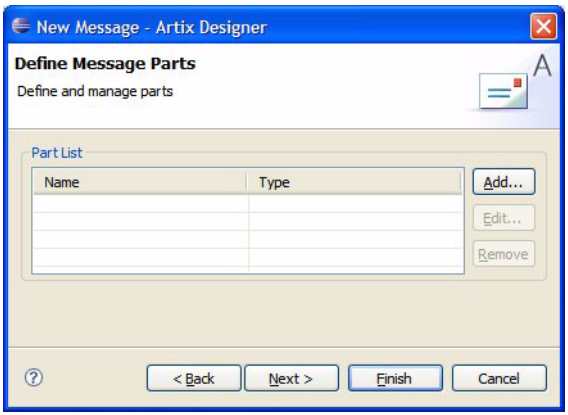
1. With the `HelloWorld.wsdl` file open and the Diagram view displayed, right-click the **Messages** node.
2. Select **New Message** from the pop-up menu. The New Message wizard opens.
3. In the Select Source Resources panel, make sure **HelloWorld.wsdl** is selected in the **Source File(s)** section and click **Next**.
4. In the Define Message Properties panel:
   i. Type **`RequestMessage`** in the **Name** text box.
   ii. Click **Next**.

Figure 7: *Define Message Properties panel*



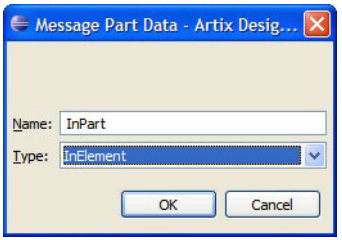
5. In the Define Message Parts panel, click **Add**.

Figure 8: *The Define Message Parts Panel*



6. In the Message Part Data dialog box:
   i. Type `InPart` in the **Name** text box.
   ii. Select **InElement** from the **Type** drop-down list.

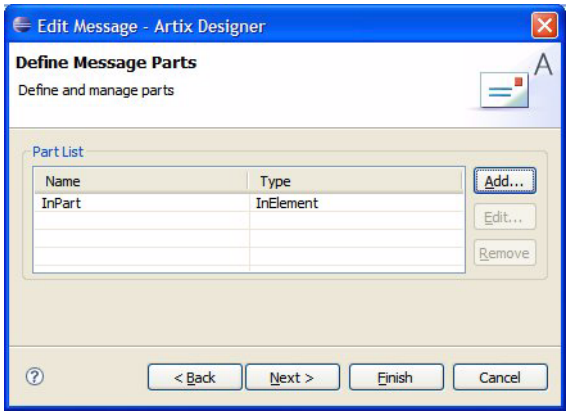
Figure 9: *The Message Part Data Dialog Box*



   iii. Click **OK** to add the `InPart` to the Part List in the Define Message Parts panel.

7. Back in the Define Message Parts panel, click **Next** to display the View Message Summary panel.

Figure 10: *The Define Message Parts Panel, with the InPart Added*



8. Click **Finish**.

To define the response message:

1. Repeat steps 1 to 3 above.
2. In the Define Message Properties panel:
   i. Type **`ResponseMessage`** in the **Name** text box
   ii. Click **Next**.
3. In the Define Message Parts panel, click **Add**.
4. In the Message Part Data dialog box:
   i. Type **`OutPart`** in the **Name** text box.
   ii. Select **OutElement** from the **Type** drop-down list.
   iii. Click **OK** to add the `OutPart` to the Part List in the Define Message Parts panel.
5. Back in the Define Message Parts panel, click **Next** to display the View Message Summary panel.
6. Click **Finish**.
7. Save the WSDL file.

Review

You have now added request and response messages to your WSDL file.

The request message includes an in part that maps to the `InElement` type, and the response message includes an out part that maps to the `OutElement` type.

```
<message name="RequestMessage">
  <part element="tns:InElement" name="InPart"/>
</message>
<message name="ResponseMessage">
  <part element="tns:OutElement" name="OutPart"/>
</message>
```

For a thorough explanation of creating messages, see *Understanding Artix Contracts*.

Defining Port Types

Overview

The `portType` element 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[1].

In this example, you will define a portType 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.

Defining a port type

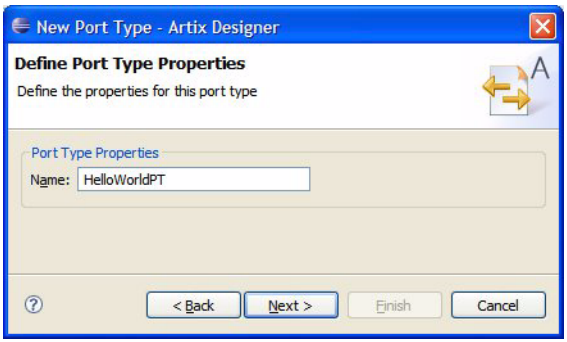
To define a port type:

1. With the `HelloWorld.wsdl` file open and the Diagram view displayed, right-click the **Port Types** node.
2. Select **New Port Type** from the pop-up menu. The New Port Type wizard opens.
3. In the Select Source Resources panel, make sure **HelloWorld.wsdl** is selected in the **Source File(s)** section.
4. Click **Next**.

1. Defining and coding fault messages is discussed in “Creating User-Defined Exceptions” in both *Developing Artix Applications in C++* and *Developing Artix Applications in Java*.

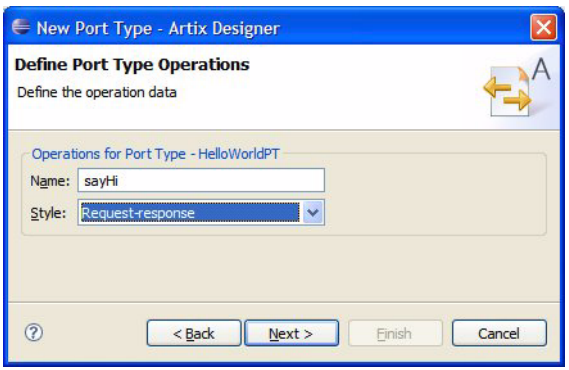
5. In the Define Port Type Properties panel:
   i. Type `HelloWorldPT` in the **Name** text box.
   ii. Click **Next**.

Figure 11: *The Define Port Type Properties Panel*



6. In the Define Port Type Operations panel:
   i. Type `sayHi` in the **Name** text box.
   ii. Select **Request-response** from the **Style** drop-down list.

Figure 12: *The Define Port Type Operations Panel*

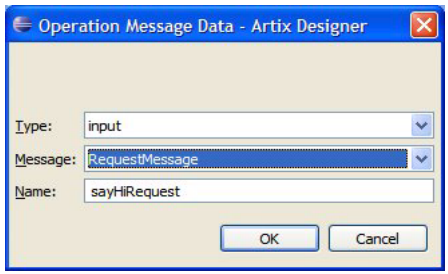


7. Click **Next**.
8. In the Define Operation Messages panel click **Add**.

9. In the Operation Message Data dialog box:
   i. In the **Type** drop-down list, select **input**.
   ii. In the **Message** drop-down list, select **RequestMessage**.

   The name `SayHiRequest` appears in the **Name** text box. You can change this to something more meaningful for your application if you prefer. For this tutorial, leave the suggested name as is.
   iii. Click **OK** to add the operation to the **Operation Messages** list in the Define Operation Messages panel.

Figure 13: *The Operation Message Data Dialog Box*



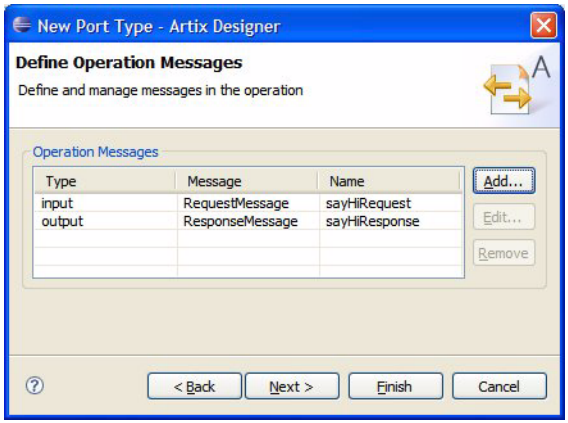
10. Back in the Define Operation Messages panel, click **Add** again.
11. In the Operation Message Data dialog box:
   i. Expand the **Type** drop-down list again. Note that **input** no longer appears in the list, because an operation can have only one input message.
   ii. Select **output** from the **Type** list.
   iii. Select **ResponseMessage** from the **Message** list.

   The name `sayHiResponse` appears in the **Name** text box. Leave the suggested name as is.
   iv. Click **OK**.

Note: You can also use the Operation Message Data dialog box to add fault messages to each operation. However, this example does not include any fault messages.

12. In the Define Operation Messages panel, click **Next** to display the Port Type Summary panel.

Figure 14: *The Define Operation Messages Panel*



13. Click **Finish** to close the wizard.
14. Save the WSDL file.

Review

You have now added the following `portType` element to your WSDL file.

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

Defining Bindings

Overview

The `binding` element in a WSDL file defines the message format and protocol details for each port. Each binding is associated with a single `portType` element, although the same `portType` can be associated with multiple bindings.

Artix Designer supports a number of binding types. In this example, you will specify a SOAP 1.1 binding with the document/literal binding style, which is required when message parts are element types.

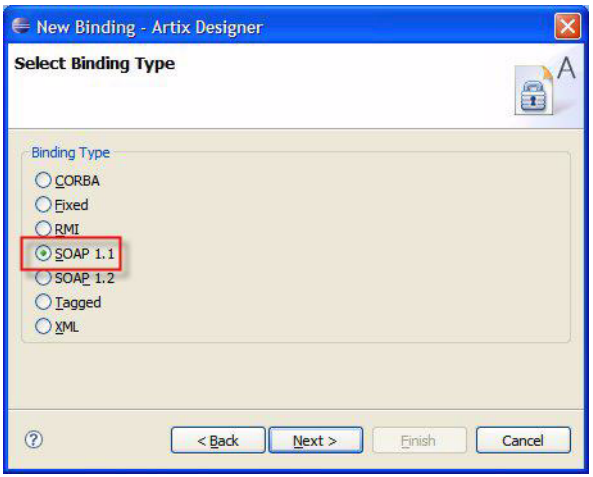
Defining a binding

To define a binding:

1. With the `HelloWorld.wsdl` file open and the Diagram view displayed, right-click the **Bindings** node.
2. Select **New Binding** from the pop-up menu. The New Binding wizard opens.
3. In the Select Source Resources panel, make sure **HelloWorld.wsdl** is selected in the **Source File(s)** section.
4. Click **Next**.

5. In the Select Binding Type panel:
   i. Select **SOAP 1.1**from the list of binding types.
   ii. Click **Next**.

Figure 15: *The Select Binding Type Panel*



6. In the Set Binding Defaults panel:
   i. Select **HelloWorldPT** from the **Port Type** drop-down list.

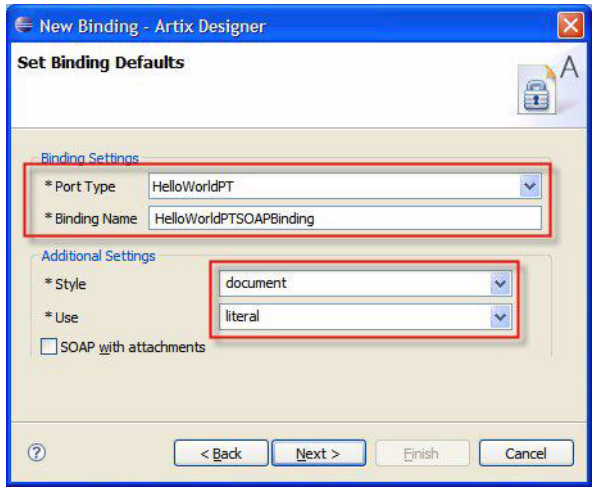
      In this case, your WSDL file contains only one `portType` element. If there were multiple port types, you would select one from the drop-down list.

> **Note:** A name is already entered in the **Binding Name** text box. You can change this entry, but be sure to give each binding in the WSDL file a unique name.

   ii. In the **Additional Settings** section, select **document** from the **Style** drop-down list.

iii. Select **literal** from the **Use** drop-down list.

Figure 16: *The Set Binding Defaults Panel*



iv. Click **Next**.

7. In the Edit Operation panel:
   i. In the Operations Editor on the left, expand the **Operations** node.
   ii. Click each **sayHi** operation node to review its binding details.

Figure 17: *Edit Operation panel*

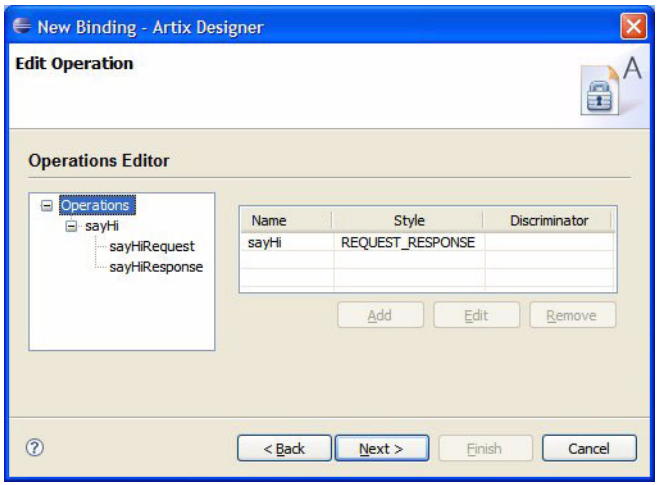
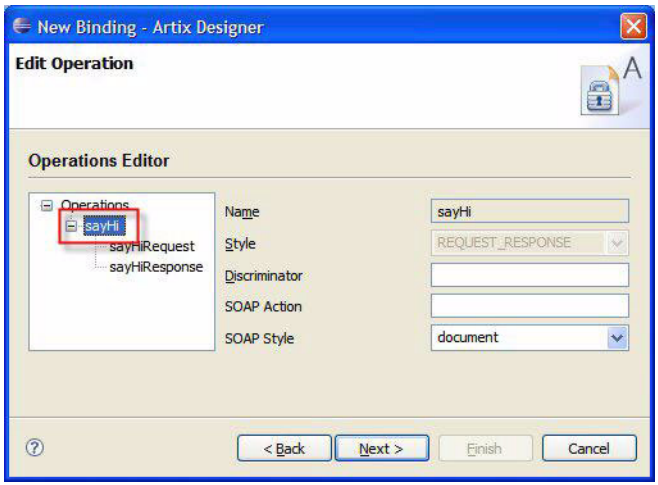


Figure 18: *Edit Operation panel, sayHi node selected*



8. Click **Next** to display the View Binding Summary panel.
9. Click **Finish** to close the wizard.
10. Save the WSDL file.

Review

You have now added the following `binding` element to your WSDL file.

```
<binding name="HelloWorldPTSOAPBinding" type="tns:HelloWorldPT">
  <soap:binding style="document" transport="http://
   schemas.xmlsoap.org/soap/http"/>
  <operation name="sayHi">
    <soap:operation soapAction="" style="document"/>
      <input name="sayHiRequest">
        <soap:body use="literal"/>
      </input>
      <output name="sayHiResponse">
         <soap:body use="literal"/>
      </output>
  </operation>
</binding>
```

Defining a Service

Overview

The `service` element of a WSDL file provides transport-specific information. Each service element can include one or more `port` elements. Each `port` element must be uniquely identified by the value of its 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 can contain multiple `service` elements.

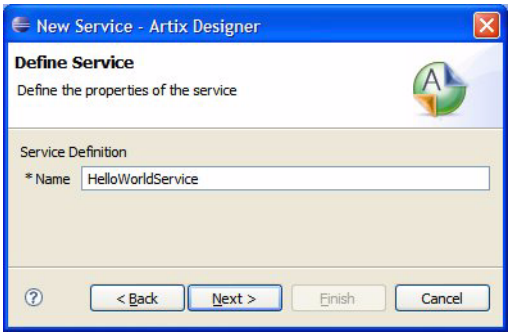
In this example, the WSDL file contains one `service` element, which contains a single `port` element.

Defining a service

To define a service:

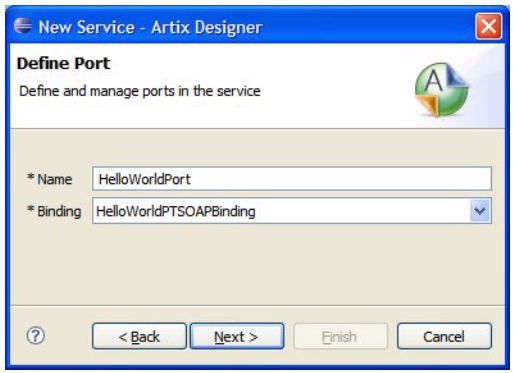
1. With the `HelloWorld.wsdl` file open and the Diagram view displayed, right-click the **Services** node.
2. Select **New Service** from the pop-up menu. The New Service wizard opens.
3. In the Select Source Resources panel, make sure **HelloWorld.wsdl** is selected in the **Source File(s)** section.
4. Click **Next**.
5. In the Define Service panel:
   i. Type **`HelloWorldService`** in the **Name** text box.
   ii. Click **Next**.

Figure 19: *The Define Service Panel*



6. In the Define Port panel:
   i. Type `HelloWorldPort` in the **Name** text box.
   ii. Select **HelloWorldPTSOAPBinding** from the **Binding** drop-down list.
   iii. Click **Next**.

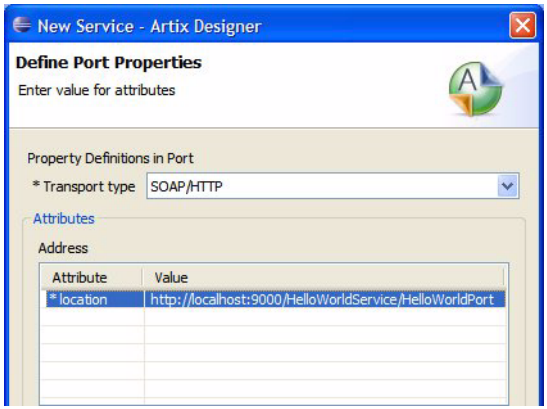
Figure 20: *The Define Port Panel*



7. In the Define Port Properties panel:
   i. From the **Transport Type** drop-down list, select **SOAP/HTTP**.
   ii. In the **Address** section, click below the **Value** header and type the following as the value for the location attribute:

```
http://localhost:9000/HelloWorldService/HelloWorldPort
```

Figure 21: *The Define Port Properties panel*



8. Click **Next** to display the View Service and Port Summary panel.
9. Click **Finish** to close the wizard.
10. Save the WSDL file.

Review

You have now completed your WSDL contract and are ready to use it to develop an application.

Click the **Source** tab in the WSDL Editor to review the WSDL that you have created. It should look like the WSDL shown in Example 1.

Example 1: *The completed HelloWorld.wsdl file*

```
<?xml version="1.0" encoding="UTF-8"?>
<!--WSDL file template-->
<!--Created by IONA Artix Designer-->
<definitions name="HelloWorld.wsdl"
  targetNamespace="http://www.iona.com/artix/HelloWorld"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap12/"
  xmlns:tns="http://www.iona.com/artix/HelloWorld"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

Example 1: *The completed HelloWorld.wsdl file (Continued)*

```
  <types>
    <schema
    targetNamespace="http://www.iona.com/artix/HelloWorld"
     xmlns="http://www.w3.org/2001/XMLSchema">
      <element name="InElement" type="string"/>
      <element name="OutElement" type="string"/>
    </schema>
  </types>
  <message name="RequestMessage">
    <part element="tns:InElement" name="InPart"/>
  </message>
  <message name="ResponseMessage">
    <part element="tns:OutElement" name="OutPart"/>
  </message>
  <portType name="HelloWorldPT">
    <operation name="sayHi">
      <input message="tns:RequestMessage" name="sayHiRequest"/>
      <output message="tns:ResponseMessage" name="sayHiResponse"/>
    </operation>
  </portType>
  <binding name="HelloWorldPTSOAPBinding" type="tns:HelloWorldPT">
    <soap:binding style="document" transport="http:schemas.xmlsoap.org/soap/
   http"/>
    <operation name="sayHi">
      <soap:operation soapAction="" style="document"/>
      <input name="sayHiRequest">
        <soap:body use="literal"/>
      </input>
      <output name="sayHiResponse">
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="HelloWorldService">
    <port binding="tns:HelloWorldPTSOAPBinding"
     name="HelloWorldPort">
      <soap:address location="http://localhost:9000/HelloWorldService/
   HelloWorldPort"/>
    </port>
  </service>
</definitions>
```

Tutorial: Generating Code

Overview

In this section, you will generate starting-point code based on the `HelloWorld.wsdl` file for the JAX-WS, JAX-RPC, and C++ projects. The Artix Tools component of Artix Designer generates the code and any other necessary configuration files for you.

Compiling the code automatically

Because the Artix Tools are integrated with the Eclipse JDT and CDT, you can make sure your code is compiled automatically as soon as it is generated. In addition, any changes you make to a Java or C++ file will be recompiled as soon as you save the file.

To make sure your code is compiled automatically, select **Build Automatically** from the **Project** menu in Eclipse.

Note: For automatic building to work for C++ with the Windows version of Artix Designer, the environment for supported version of Visual C++ in must be set before starting starts Artix Designer, as described in the Artix Installation Guide.

Creating code generation configurations

Artix Tools

Before you can generate code, Artix Designer prompts you to define a code generation configuration. You can use the saved configuration to re-generate the code, or you can copy one saved configuration to a new one, and edit it for a different project.

In this step, we will create separate code generation configurations for a

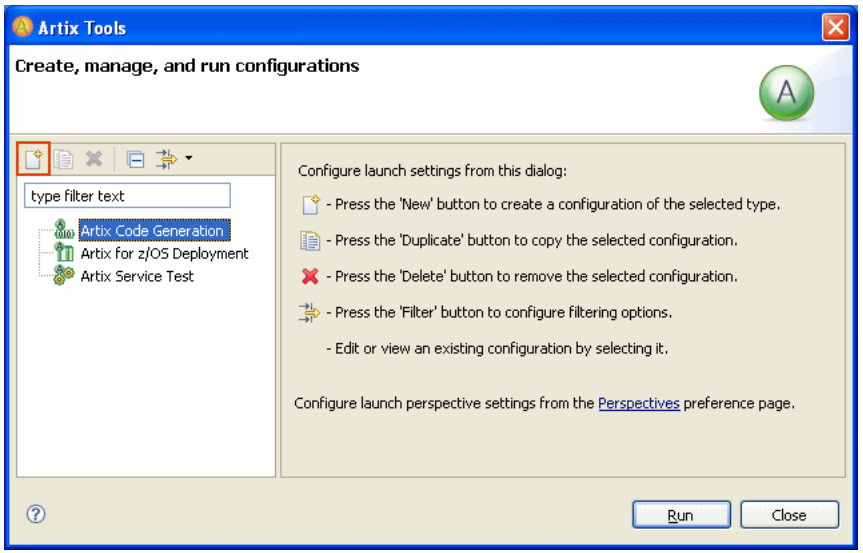
- JAX-WS client and server,
- JAX-RPC client and server,
- C++ client and server.

Creating the JAX-WS client and server configuration

To create the JAX-WS client and server code generation configuration:

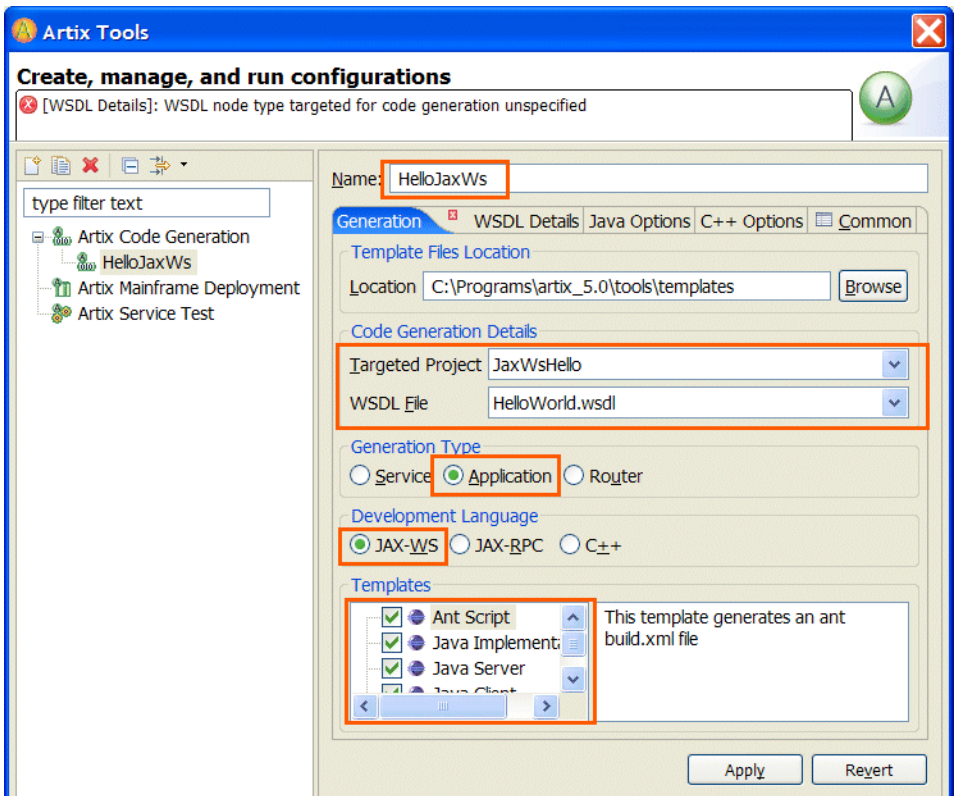
1. Select **Artix | Artix Tools | Artix Tools**.
2. In the Artix Tools panel, select **Artix Code Generation**.

Figure 22: *The Artix Tools Panel*



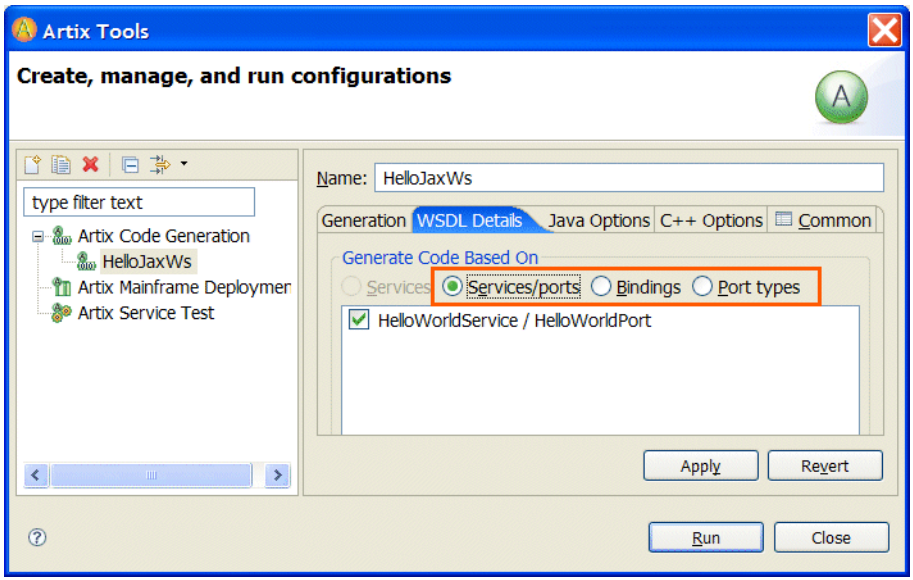
3. From the toolbar on the left, click the **New launch configuration** button (marked in red in Figure 22).
4. In the **Name** text box, replace the default name with a more meaningful one for this configuration, such as `HelloJaxWs`.
5. In the **Generation** tabbed page:
   i. From the **Targeted Project** drop-down list, select **JaxWsHello**.
   ii. From the **WSDL File** drop-down list, select **HelloWorld.wsdl**.
   iii. In the **Generation Type** section, select **Application**.
   iv. In the **Development Language** section, select **JAX-WS**.
   v. In the **Templates** section, check all of the checkboxes.

Figure 23: *Artix Tools Panel, HelloJ configuration, Generation Tab*



6. Click the **WSDL Details** tab. Then:
   i. Select the **Services/ports** control.
   ii. Make sure the **HelloWorldService/HelloWorldPort** checkbox is checked.
   iii. Select the **Bindings** control.
   iv. Make sure the **HelloWorldPTSOAPBinding** checkbox is checked.
   v. Select the **Port types** control.
   vi. Make sure the **HelloWorldPT** checkbox is checked.
   vii. Click the **Apply** button to save the configuration.

Figure 24: *Artix Tools panel, WSDL Details tab*



Generating the Java code

To generate Java code from the configuration saved in the previous steps:

7. Click **Run**.

The Artix Tools create all the Java classes and configuration files for your client and server application. The Eclipse JDT compiles the code automatically.

Unless you have changed your Artix Designer code generation preferences, the source code is written to the following location:

```
EclipseWorkspace/JaxWsHello/HelloJaxWs/src/com/iona/artix/HelloJaxWs
```

The compiled bytecode is written to the following location:

```
EclipseWorkspace/JaxWsHello/bin
```

Creating the JAX-RPC client and server configuration

You can quickly create a JAX-RPC build configuration by duplicating and editing the JAX-WS configuration.

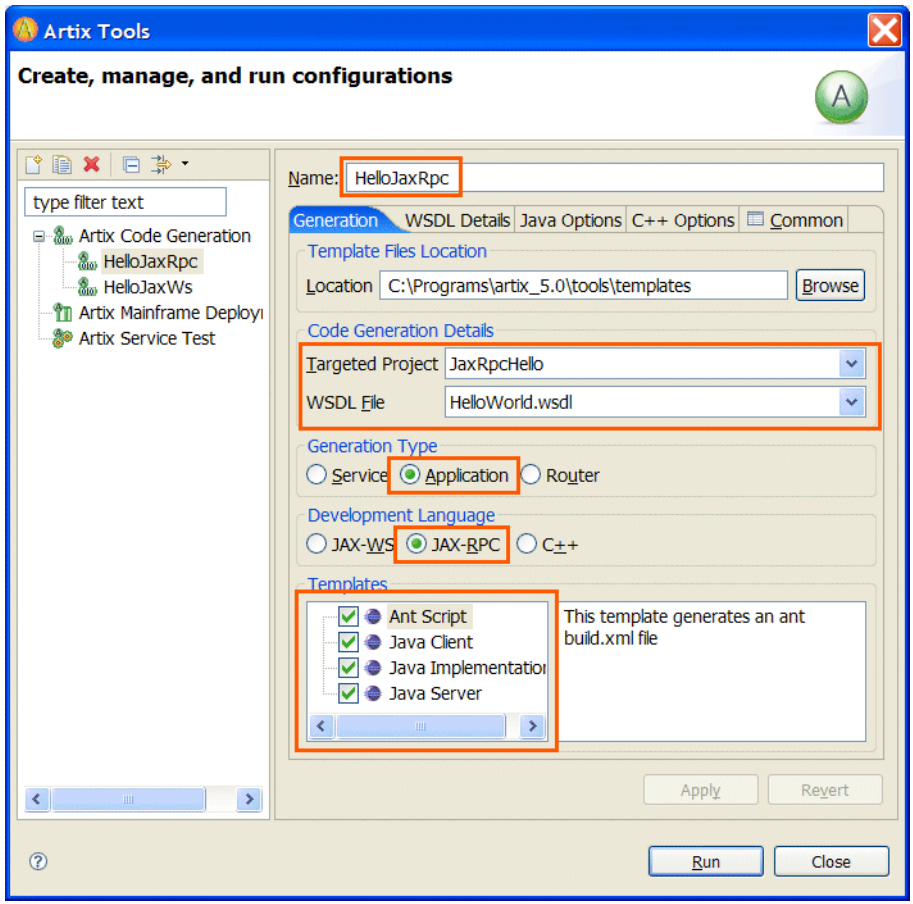
To create the JAX-RPC client and server code generation configuration:

1. In the Eclipse menu, select the **Artix| Artix Tools | Artix Tools** menu.
2. In the Artix Tools window, select the **HelloJaxWs** configuration.
3. From the toolbar, click the **Duplicate launch configuration** button.

Figure 25: *The Duplicate Launch Configuration Button*

4. In the **Name** text box, change the name to **`HelloJaxRpc`**.
5. In the **Generation** tabbed page:
   i. From the **Targeted Project** drop-down list, select **JaxRpcHello**.
   ii. From the **WSDL File** drop-down list, select **HelloWorld.wsdl**.
   iii. In the **Generation Type** section, select **Application**.
   iv. In the **Development Language** section, select **JAX-RPC**.
   v. In the **Templates** section, check all of the checkboxes.
   vi. Click **Apply**.

Figure 26: *Generating a JAX-RPC Configuration*



Generating the JAX-RPC code

To generate starting point JAX-RPC code from the configuration saved in the previous steps:

6. Click **Run**.

Creating the C++ client and server configuration

You can quickly create a C++ build configuration by duplicating and editing the JAX-WS configuration.

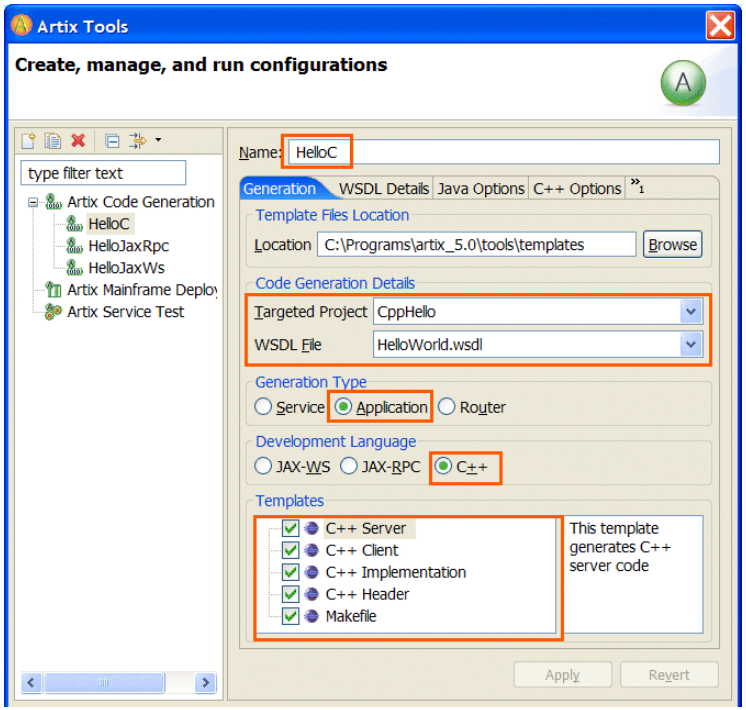
Note: If you are using the Windows version of Artix Designer, make sure you have set the environment for a supported version of Visual C++ before creating the C++ configuration. See the Artix Installation Guide for information on setting this up.

To create the C++ client and server code generation configuration:

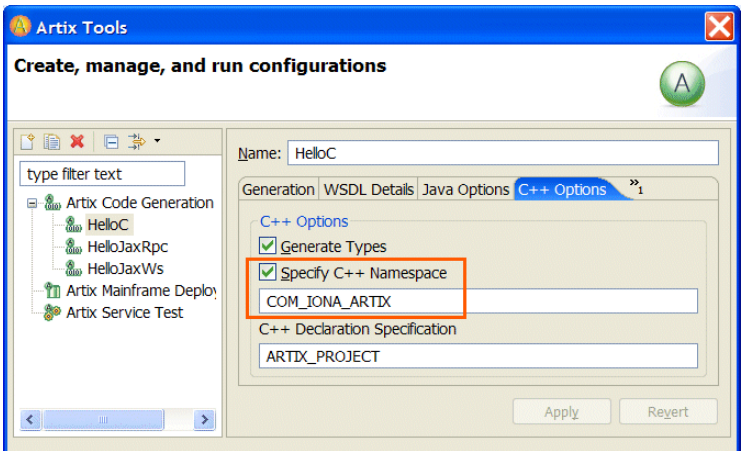
1. In the Eclipse menu, select the **Artix| Artix Tools | Artix Tools** menu.
2. In the Artix Tools window, select the **HelloJaxWs** configuration.
3. From the toolbar, click the **Duplicate launch configuration** button.

Figure 27: *The Duplicate Launch Configuration Button*

4. In the **Name** text box, change the name to `HelloC`.
5. In the **Generation** tabbed page:
   i. From the **Targeted Project** drop-down list, select **CppHello**.
   ii. From the **WSDL File** drop-down list, select **HelloWorld.wsdl**.
   iii. In the **Generation Type** section, select **Application**.
   iv. In the **Development Language** section, select **C++**.
   v. In the **Templates** section, check all of the checkboxes.
   vi. Click **Apply**.



6. Click the **WSDL Details** tab. Then:
   i. Select the **Services/ports** control.
   ii. Make sure the **HelloWorldService** / **HelloWorldPort** checkbox is checked.
   iii. Select the **Bindings** control.
   iv. Make sure the **HelloWorldPTSOAPBinding** checkbox is checked.
   v. Select the Port type control.
   vi. Make sure the HelloWorldPT checkbox is checked.
   vii. Click **Apply** to save the configuration.
7. Click the **C++ Options** tab.
   i. Check the **Specify C++ Namespace** checkbox.
   ii. Accept the default namespace, COM_IONA_ARTIX.
   iii. Click **Apply** to save the configuration.



Generating the C++ code

To generate starting point C++ code from the configuration saved in the previous steps:

8. Click **Run**.

The Artix Tools create all the C++ source and header files for your client and server applications in the following location:

```
EclipseWorkspace\CppHello\HelloC\src
```

Tutorial: Running the Applications

Overview

You are now ready to run the client and server applications for the JAX-WS, JAX-RPC, and C++ projects.

You can launch Java and C++ applications from within the Eclipse environment, although the procedures for each are different.

Editing run configurations

Artix Designer automatically creates run configurations for each generated executable. You can edit and save a run configuration in much the same way that you saved the code generation configurations in "Tutorial: Generating Code" on page 66.

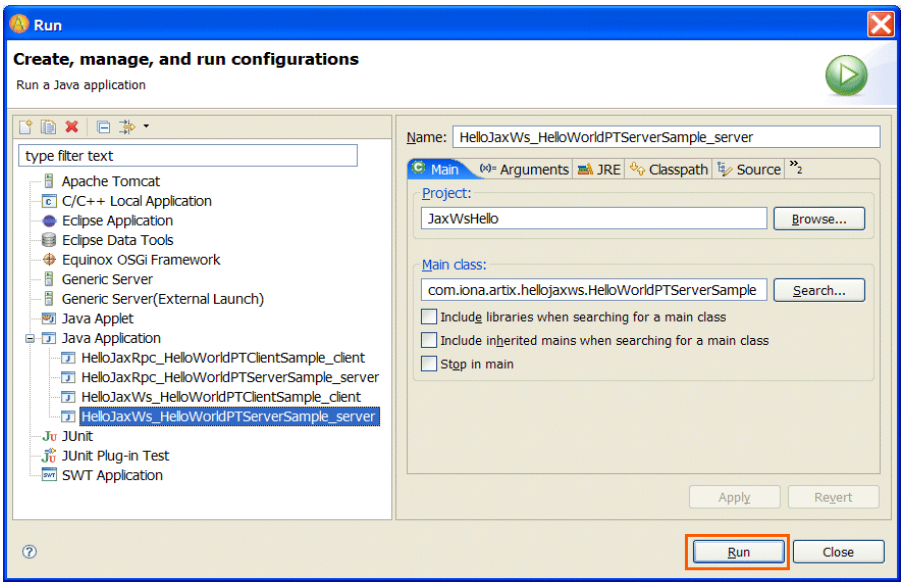
A saved run configuration saves time when re-running your application, because it saves the environment and any arguments necessary for each invocation. You can copy a saved run configuration and edit it to create a new run configuration.

Running the JAX-WS server

To run the JAX-WS server:

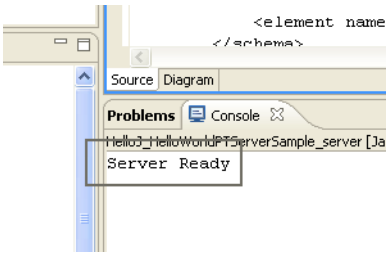
1. Right-click the **JaxWsHello** project folder and select **Run As|Run** from the context menu (or invoke **Run | Run** from the main Eclipse menu).
2. In the Run dialog, select **Java Application | HelloJaxWs_HelloWorldPTServerSample_server** from the list of run configurations on the left.

Figure 28: *Java Application Launch Configurations in the Run Window*



3. Click **Run**.

The server process starts running in the Eclipse Console view. After a moment, the words “Server Ready” appear in the Eclipse Console view.



Running the JAX-WS client

To run the JAX-WS client:

1. Right-click the **JaxWsHello** project folder and select **Run As|Run** from the context menu.
2. In the Run window, select **Java Application | HelloJaxWs_HelloWorldPTClientSample_client** from the list of Java launch configurations.
3. Click **Run**.

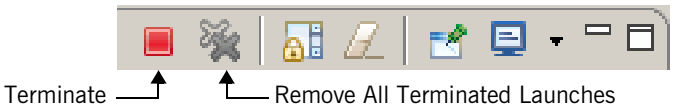
The words `Operation sayHi received: Curry` appear in the Eclipse Console view.

Stopping applications started within Eclipse

You can stop applications that you started within Eclipse by using the toolbar buttons above the Eclipse Console view.

To clear the client output, click the **Remove All Terminated Launches** button on the Console view's toolbar.

Figure 29: *Eclipse Console View toolbar*



To stop the server process, click the **Terminate** button.

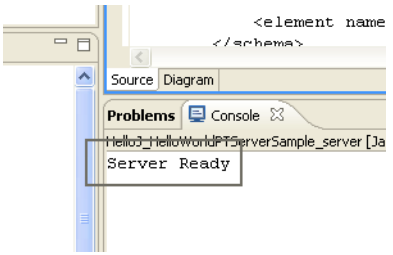
Click the **Remove All Terminated Launches** button again to clear the server output.

Running the JAX-RPC server

To run the JAX-RPC server:

1. Right-click the **JaxRpcHello** project folder and select **Run As|Run** from the context menu (or invoke **Run | Run** from the main Eclipse menu).
2. In the **Run** dialog, select **Java Application | HelloJaxRpc_HelloWorldPTServerSample_server** from the list of run configurations on the left.
3. Click **Run**.

The server process starts running in the Eclipse Console view. After a moment, the words “Server Ready” appear in the Eclipse Console view.

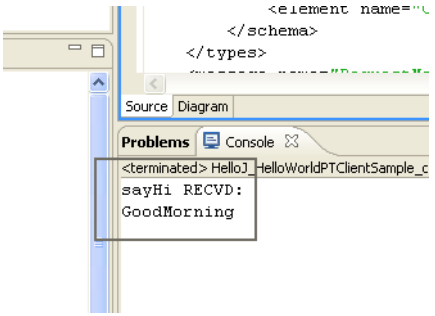


Running the JAX-RPC client

To run the JAX-RPC client:

1. Right-click the **JaxRpcHello** project folder and select **Run As|Run** from the context menu.
2. In the Run window, select **Java Application | HelloJaxRpc_HelloWorldPTClientSample_client** from the list of Java launch configurations.
3. Click **Run**.

The words “sayHi RECVD: GoodMorning” appear in the Eclipse Console view.

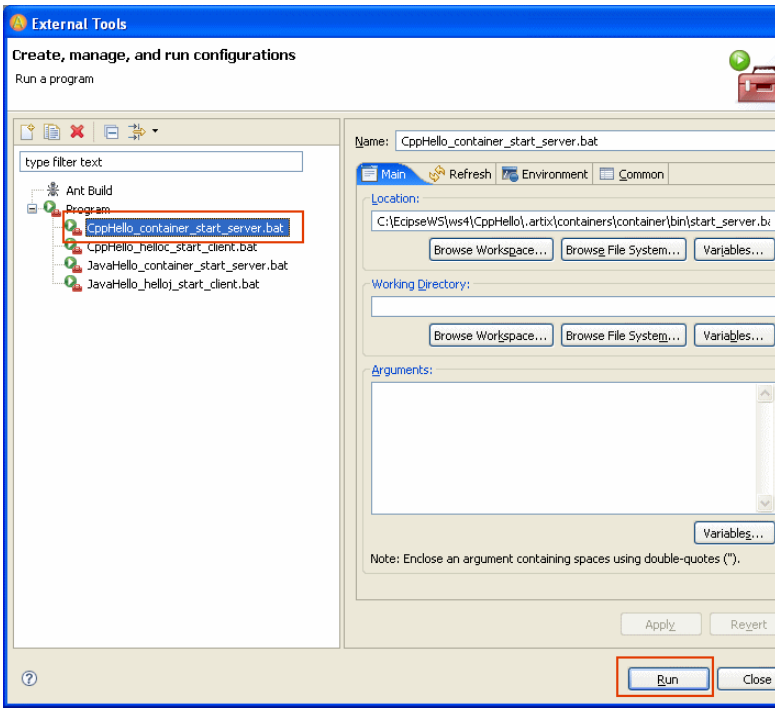


Running the C++ server

To run the C++ server:

1. From the Eclipse menu bar, select **Run | External Tools | External Tools**.
2. In the **External Tools** window, expand the **Program** node in the list of configurations on the left.
3. Select the launch configuration entry for **CppHello_container_start_server.bat**.

4. Click **Run**.



5. If you are using Windows XP SP2 with the Windows Firewall enabled, the firewall displays a Security Alert.

6. Click **Unblock** to allow the server to run. This opens a small Command Prompt window.

Running the C++ client

Continuing the steps above:

7. Select the launch configuration entry for **CppHello_consumer_instance_start_client.bat**.
8. Click **Run**.

Stop and clear the C++ server and client

To terminate the C++ server, close its Command Prompt window.

To clear the C++ client, use the methods described in "Stopping applications started within Eclipse" on page 77.

Command-line alternatives

When you use an Artix Designer code generation configuration to create an Artix application, start and stop scripts are added to the project.

You can the C++ application by running the appropriate start script from the command prompt.

Note: If an application takes any arguments, you must edit its start script accordingly.

To run your C++ application from the command line:

1. Open a command prompt and change to the following directory:

```
EclipseWorkspace\CppHello\.artix\containers\container\bin
```

2. Run the `start_server` script.

   The server application launches in a new command window.

3. From the command prompt, change to the following directory:

```
EclipseWorkspace\CppHello\.artix\containers\consumer_instan
   ce\bin
```

4. Run the `start_client` script.

   The client application launches and displays the text, `Operation sayHi received: OutPart = GoodMorning`.

Press **Ctrl+C** to close the client and server command windows, in that order.

APPENDIX A

Understanding WSDL

Artix contracts use WSDL documents to describe services and the data they use.

In this appendix

This appendix discusses the following topics:

WSDL Basics

Overview

Web Services Description Language (WSDL) is an XML document format used to describe services offered over the Web. WSDL is standardized by the World Wide Web Consortium (W3C) and is currently at revision 1.1. You can find the standard on the W3C website at www.w3.org/TR/wsdl.

Elements of a WSDL document

A WSDL document is made up of the following elements:

- `import` allows you to import another WSDL or XSD file.
- Logical contract elements:
  - `types`
  - `message`
  - `operation`
  - `portType`
- Physical contract elements:
  - `binding`
  - `port`
  - `service`

These elements are described in "WSDL elements" on page 25.

Abstract operations

The abstract definition of *operations* and *messages* is separated from the concrete data formatting definitions and network protocol details. As a result, the abstract definitions can be reused and recombined to define several endpoints. For example, a service can expose identical operations with slightly different concrete data formats and two different network addresses. Alternatively, one WSDL document could be used to define several services that use the same abstract messages.

The portType

A *portType* is a collection of abstract operations that define the actions provided by an endpoint.

Concrete details

When a portType is mapped to a concrete data format, the result is a concrete representation of the abstract definition.A *port* is defined by associating a network address with a reusable *binding*, in the form of an endpoint. A collection of ports (or endpoints) define a service.

Because WSDL was intended to describe services offered over the Web, the concrete message format is typically SOAP and the network protocol is typically HTTP. However, WSDL documents can use any concrete message format and network protocol. In fact, Artix contracts bind operations to several data formats and describe the details for a number of network protocols.

Namespaces and imported descriptions

WSDL supports the use of XML namespaces defined in the `definition` element as a way of specifying predefined extensions and type systems in a WSDL document. WSDL also supports importing WSDL documents and fragments for building modular WSDL collections.

Example

Example 1 on page 64 shows a simple WSDL document.

Abstract Data Type Definitions

Overview

Applications typically use data types that are more complex than the primitive types, like `int`, defined by most programming languages. WSDL documents represent these complex data types using a combination of schema types defined in referenced external XML schema documents and complex types described in `types` elements.

Complex type definitions

Complex data types are described in a `types` element. The W3C specification states that XSD is the preferred canonical type system for a WSDL document. Therefore, XSD is treated as the intrinsic type system. Because these data types are abstract descriptions of the data passed over the wire, and are not concrete descriptions, there are a few guidelines on using XSD schemas to represent them:

- Use elements, not attributes.
- Do not use protocol-specific types as base types.
- Define arrays using the SOAP 1.1 array encoding format.

WSDL does allow for the specification and use of alternative type systems within a document.

Example

The structure, `personalInfo`, defined in Example 2, contains a `string`, an `int`, and an `enum`. The `string` and the `int` both have equivalent XSD types and do not require special type mapping. The enumerated type `hairColorType`, however, does need to be described in XSD.

Example 2: *personalInfo structure*

```
enum hairColorType {red, brunette, blonde};

struct personalInfo
{
 string name;
 int age;
 hairColorType hairColor;
}
```

Example 3 shows one mapping of `personalInfo` into XSD. This mapping is a direct representation of the data types defined in Example 2. `hairColorType` is described using a named `simpleType` because it does not have any child elements. `personalInfo` is defined as an `element` so that it can be used in messages later in the contract.

Example 3: *XSD type definition for personalInfo*

```
<types>
  <xsd:schema targetNamespace="http://iona.com/personal/schema"
    xmlns:xsd1="http://iona.com/personal/schema"
    xmlns="http://www.w3.org/2000/10/XMLSchema"/>
    <simpleType name="hairColorType">
      <restriction base="xsd:string">
        <enumeration value="red"/>
        <enumeration value="brunette"/>
        <enumeration value="blonde"/>
      </restriction>
    </simpleType>
    <element name="personalInfo">
      <complexType>
        <sequence>
            <element name="name" type="xsd:string"/>
            <element name="age" type="xsd:int"/>
           <element name="hairColor" type="xsd1:hairColorType"/>
        </sequence>
      </complexType>
    </element>
</types>
```

Another way to map personalInfo is to describe `hairColorType` in-line as shown in Example 4. WIth this mapping, however, you cannot reuse the description of `hairColorType`.

Example 4: *Alternate XSD Mapping for personalInfo*

```
<types>
  <xsd:schema targetNamespace="http://iona.com/personal/schema"
    xmlns:xsd1="http://iona.com/personal/schema"
    xmlns="http://www.w3.org/2000/10/XMLSchema"/>
    <element name="personalInfo">
      <complexType>
        <sequence>
          <element name="name" type="xsd:string"/>
          <element name="age" type="xsd:int"/>
```

Example 4: *Alternate XSD Mapping for personalInfo (Continued)*

```
          <element name="hairColor">
            <simpleType>
              <restriction base="xsd:string">
                <enumeration value="red"/>
                <enumeration value="brunette"/>
                <enumeration value="blonde"/>
              </restriction>
            </simpleType>
          </element>
        </sequence>
      </complexType>
    </element>
</types>
```

Abstract Message Definitions

Overview

WSDL is designed to describe how data is passed over a network. It describes data that is exchanged between two endpoints in terms of abstract messages described in `message` elements.

Each abstract message consists of one or more parts, defined in `part` elements.

These abstract messages represent the parameters passed by the operations defined by the WSDL document and are mapped to concrete data formats in the WSDL document's `binding` elements.

Messages and parameter lists

For simplicity in describing the data consumed and provided by an endpoint, WSDL documents allow abstract operations to have only one input message, the representation of the operation's incoming parameter list, and only one output message, the representation of the data returned by the operation.

In the abstract message definition, you cannot directly describe a message that represents an operation's return value. Therefore, any return value must be included in the output message.

Messages allow for concrete methods defined in programming languages like C++ to be mapped to abstract WSDL operations. Each message contains a number of `part` elements that represent one element in a parameter list.

Therefore, all of the input parameters for a method call are defined in one message and all of the output parameters, including the operation's return value, are mapped to another message.

Example

For example, imagine a server that stores personal information as defined in Example 2 on page 88 and provides a method that returns an employee's data based on an employee ID number.

The method signature for looking up the data would look similar to Example 5.

Example 5: *Method for Returning an Employee's Data*

```
personalInfo lookup(long empId)
```

This method signature could be mapped to the WSDL fragment shown in Example 6.

Example 6: *WSDL Message Definitions*

```
<message name="personalLookupRequest">
  <part name="empId" type="xsd:int" />
</message>
<message name="personalLookupResponse">
  <part name="return" element="xsd1:personalInfo" />
</message>
```

Message naming

Each message in a WSDL document must have a unique name within its namespace. Choose message names that show whether they are input messages (requests) or output messages (responses).

Message parts

Message parts are the formal data elements of the abstract message. Each part is identified by a `name` attribute and by either a `type` or an `element` attribute that specifies its data type. The data type attributes are listed in Table 2.

Table 2: *Part Data Type Attributes*

Attribute	Description
`type="`*`type_name`*`"`	The data type of the part is defined by a `simpleType` or `complexType` called *`type_name`*
`element="`*`elem_name`*`"`	The data type of the part is defined by an `element` called *`elem_name`*.

Messages are allowed to reuse part names. For instance, if a method has a parameter, `foo`, which is passed by reference or is an in/out, it can be a part in both the request message and the response message. An example of parameter reuse is shown in Example 7.

Example 7: *Reused Part*

```
<message name="fooRequest">
  <part name="foo" type="xsd:int"/>
</message>
<message name="fooReply">
  <part name="foo" type="xsd:int"/>
</message>
```

Abstract Interface Definitions

Overview

WSDL `portType` elements define, in an abstract way, the operations offered by a service. The operations defined in a portType list the input, output, and any fault messages used by the service to complete the transaction the operation describes.

PortTypes

A `portType` can be thought of as an interface description. In many Web service implementations there is a direct mapping between portTypes and implementation objects. PortTypes are the abstract unit of a WSDL document that is mapped into a concrete binding to form the complete description of what is offered over a port.

PortTypes are described using the `portType` element in a WSDL document. Each portType in a WSDL document must have a unique name, specified using the `name` attribute, and is made up of a collection of operations, described in `operation` elements. A WSDL document can describe any number of portTypes.

Operations

Operations, described in `operation` elements in a WSDL document, are an abstract description of an interaction between two endpoints. For example, a request for a checking account balance and an order for a gross of widgets can both be defined as operations.

Each operation within a portType must have a unique name, specified using the required `name` attribute.

Elements of an operation

Each operation is made up of a set of elements. The elements represent the messages communicated between the endpoints to execute the operation.

The elements that can describe an operation are listed in Table 3.

Table 3: *Operation Message Elements*

Element	Description
`input`	Specifies a message that is received from another endpoint. This element can occur at most once for each operation.

Table 3: *Operation Message Elements (Continued)*

Element	Description
`output`	Specifies a message that is sent to another endpoint. This element can occur at most once for each operation.
`fault`	Specifies a message used to communicate an error condition between the endpoints. This element is not required and can occur an unlimited number of times.

An operation is required to have at least one `input` or `output` element. The elements are defined by two attributes listed in Table 4.

Table 4: *Attributes of the Input and Output Elements*

Attribute	Description
`name`	Identifies the message so it can be referenced when mapping the operation to a concrete data format. The name must be unique within the enclosing port type.
`message`	Specifies the abstract message that describes the data being sent or received. The value of the `message` attribute must correspond to the `name` attribute of one of the abstract messages defined in the WSDL document.

It is not necessary to specify the `name` attribute for all input and output elements; WSDL provides a default naming scheme based on the enclosing operation's name.

If only one element is used in the operation, the element name defaults to the name of the operation. If both an `input` and an `output` element are used, the element name defaults to the name of the operation with `Request` or `Response`, respectively, appended to the name.

Return values

Because the portType is an abstract definition of the data passed during an operation, WSDL does not provide for return values to be specified for an operation. If a method returns a value, it is mapped into the output message as the last part of that message. The concrete details of how the message parts are mapped into a physical representation are described in "Bindings" on page 97.

Example

For example, in implementing a server that stores personal information in the structure defined in Example 2 on page 88, you might use an interface similar to the one shown in Example 8.

Example 8: *personalInfo Lookup Interface*

```
interface personalInfoLookup
{
  personalInfo lookup(in int empID)
  raises(idNotFound);
}
```

This interface could be mapped to the portType in Example 9.

Example 9: *personalInfo Lookup Port Type*

```
<types>
...
  <element name="idNotFound" type="idNotFoundType">
  <complexType name="idNotFoundType">
    <sequence>
      <element name="ErrorMsg" type="xsd:string"/>
      <element name="ErrorID" type="xsd:int"/>
    </sequence>
  </complexType>
</types>
<message name="personalLookupRequest">
  <part name="empId" type="xsd:int" />
</message>
<message name="personalLookupResponse">
  <part name="return" element="xsd1:personalInfo" />
</message>
<message name="idNotFoundException">
  <part name="exception" element="xsd1:idNotFound" />
</message>
<portType name="personalInfoLookup">
  <operation name="lookup">
    <input name="empID" message="personalLookupRequest" />
    <output name="return" message="personalLookupResponse" />
    <fault name="exception" message="idNotFoundException" />
  </operation>
</portType>
```

Mapping to the Concrete Details

Overview

The abstract definitions in a WSDL document are intended to be used in defining the interaction of real applications that have specific network addresses, use specific network protocols, and expect data in a particular format. To fully define these real applications, the abstract definitions discussed in the previous section must be mapped to concrete representations of the data passed between applications. The details describing the network protocols in use must also be added.

This is accomplished in the WSDL `bindings` and `ports` elements. WSDL binding and port syntax is not tightly specified by the W3C. A specification is provided that defines the mechanism for defining these syntaxes. However, the syntaxes for bindings other than SOAP and for network transports other than HTTP are not defined in a W3C specification.

Bindings

Bindings describe the mapping between the abstract messages defined for each portType and the data format used on the wire. Bindings are described in `binding` elements in the WSDL file. A binding can map to only one portType, but a portType can be mapped to any number of bindings.

It is within the bindings that you specify details such as parameter order, concrete data types, and return values. For example, a binding can reorder the parts of a message to reflect the order required by an RPC call. Depending on the binding type, you can also identify which of the message parts, if any, represent the return type of a method.

Services

To define an endpoint that corresponds to a running service, the `port` element in the WSDL file associates a binding with the concrete network information needed to connect to the remote service described in the file. Each port specifies the address and configuration information for connecting the application to a network.

Ports are grouped within `service` elements. A service can contain one or many ports. The convention is that the ports defined within a particular service are related in some way. For example, all of the ports might be bound to the same portType, but use different network protocols, like HTTP and WebSphere MQ.

Index