PROGRESS SOFTWARE

Artix® ESB

Getting Started with Artix

Version 5.5, December 2008

Updated: December 5, 2008

Contents

List of Figures

List of Tables

Preface

What is Covered in This Book

Getting Started with Artix introduces IONA's Artix ESB technology, the Artix Designer GUI tool, and Web Services Description Language (WSDL).

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, "About Artix ESB," introduces the Artix ESB product, discussing key concepts in depth and describing the types of problems it is designed to solve.
- Chapter 2, "Understanding WSDL," explains the basics of WSDL.
- Chapter 3, "Artix Designer Tutorials," walks you through using the Java-first and WSDL-first techniques with Artix Designer to create SOA-network client-server applications.

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

About Artix ESB

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 products already present within your organization.

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.

SOAs take the Web services concept and extend it to the entire enterprise. Using a SOA, 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

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 over other ESBs 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, Artix ESB 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.

Artix ESB is highly configurable and easily extendable. You can configure it to load only 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 service, extend the contract definitions, and configure Artix to load it.

Artix ESB features

Artix ESB includes the following 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 a number of different containers
- 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
- 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

Runtimes and programming models

Artix ESB ships with two runtimes:

- The Artix ESB C++ Runtime supports development using either the Artix C++ API or the Java JAX-RPC API.
- The Java Runtime is based on the Apache CXF, which provides a JAX-WS API and a JavaScript API.

Using Artix ESB

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

- You can use Artix ESB to develop new applications using one of the supported APIs. In this situation, developers generate stub and skeleton code from WSDL, 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 difference between the two runtimes and the programming models available in Artix ESB.
2. The syntax for WSDL and the Artix ESB extensions to the WSDL specification.
3. The configuration mechanisms available in the two Artix runtimes.
4. The Artix APIs that you can use in your application.
5. 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 concepts. The other books in the Artix documentation library covers the same technologies in greater detail.

Runtime Comparison

C++ Runtime

Artix ESB C++ Runtime provides developers with both a C++ API and a JAX-RPC API with which to implement services. It is built on top of Progress Software's patented Adaptive Runtime Technology (ART).

Artix ESB C++ Runtime has a C++ core that provides a fast and stable platform for building applications. The JAX-RPC APIs are linked into the C++ core using a JNI layer.

Java Runtime

Artix ESB Java Runtime provides the developer with both a JAX-WS 2.0 API and a JavaScript API with which to implement services.

It is based on the Apache CXF services framework and provides a fast, modular, and extensible platform for implementing services that is built purely in Java.

Feature comparison

The table below gives a comparison of the bindings, transports, and qualify of service features supported by each runtime.

Table 1: *Artix ESB Feature Comparison*

Feature	Java Runtime	C++ Runtime
Supported APIs		
JAX-RPC 1.1	No	Yes
JAXB 2.0	Yes	No
JAX-WS 2.0	Yes	No
WSDL 1.1	Yes	Yes
JCA Connector	Yes	Yes
Developoment Language		
Java	Yes	Yes
C++	No	Yes

Table 1: *Artix ESB Feature Comparison (Continued)*

Feature	Java Runtime	C++ Runtime
JavaScript	Yes	No
Bindings		
SOAP (1.1 and 1.2)	Yes	Yes
MTOM/XOP	Yes	No
RESTful	Yes	No
CORBA	Yes	Yes
Pure XML	Yes	Yes
Fixed length records	Via Artix Data Services	Yes
Tagged data	Via Artix Data Services	Yes
Tibco Rendezvous messages	No	Yes
FML	No	Yes
Transports		
HTTP	Yes	Yes
JMS	Yes	Yes
FTP	Yes	Yes
WebSphere MQ	Yes	Yes
IIOP	No	Yes
Tuxedo	No	Yes
Quality of Service		
Message routing	Yes	Yes
Security	Yes	Yes

Table 1: *Artix ESB Feature Comparison (Continued)*

Feature	Java Runtime	C++ Runtime
JavaScript	Yes	No
Bindings		
SOAP (1.1 and 1.2)	Yes	Yes
MTOM/XOP	Yes	No
RESTful	Yes	No
CORBA	Yes	Yes
Pure XML	Yes	Yes
Fixed length records	Via Artix Data Services	Yes
Tagged data	Via Artix Data Services	Yes
Tibco Rendezvous messages	No	Yes
FML	No	Yes
Transports		
HTTP	Yes	Yes
JMS	Yes	Yes
FTP	Yes	Yes
WebSphere MQ	Yes	Yes
IIOP	No	Yes
Tuxedo	No	Yes
Quality of Service		
Message routing	Yes	Yes
Security	Yes	Yes

Table 1: *Artix ESB Feature Comparison (Continued)*

Feature	Java Runtime	C++ Runtime
Reliable messaging	Yes	Yes
High availability	Yes	Yes
Load balancing	Yes	Yes
Location resolution	Yes	Yes
Statefulness	No	Yes

Key Concepts in Depth

This section discusses key Artix ESB concepts in depth.

In this section

This section discusses the following topics:

Artix ESB Runtime Components

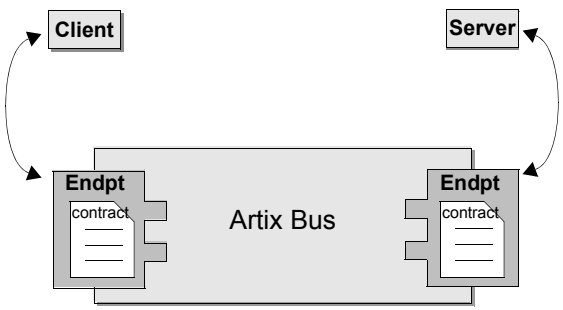
How it fits together

The Artix ESB runtime consists of the following components :

- 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 1 illustrates how the Artix ESB elements fit together.

Figure 1: *Artix ESB Runtime Components*



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.

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 1 on page 20. 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 one of the following containers:

- Artix container (proprietary)
- Servlet container, such as Apache Tomcat
- OSGi container, such as Apache ServiceMix or Equinox

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.

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.

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. It 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

Understanding WSDL

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

In this chapter

This chapter 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 23.

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 9 on page 88 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 1, 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 1: *personalInfo structure*

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

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

Example 2 shows one mapping of `personalInfo` into XSD. This mapping is a direct representation of the data types defined in Example 1. `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 2: *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 3. WIth this mapping, however, you cannot reuse the description of `hairColorType`.

Example 3: *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 3: *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 1 on page 34 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 4.

Example 4: *Method for Returning an Employee's Data*

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

This method signature could be mapped to the WSDL fragment shown in Example 5.

Example 5: *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 6.

Example 6: *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.
`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 44.

Example

For example, in implementing a server that stores personal information in the structure defined in Example 1 on page 34, you might use an interface similar to the one shown in Example 7.

Example 7: *personalInfo Lookup Interface*

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

This interface could be mapped to the portType in Example 8.

Example 8: *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.

CHAPTER 3

Artix Designer Tutorials

The tutorials in this chapter walk you though using the Java-first and WSDL-first techniques in Artix Designer to generate starting-point code for SOA-network client-server applications.

In this chapter

This chapter contains the following sections:

Introducing Artix Designer

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.

Generating WSDL

Artix Designer contains wizards that let you create WSDL files based on:

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

Editing WSDL

Artix Designer has a WSDL editor that is integrated with its code-generation tools and that thoroughly understands 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

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.

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 toolbar

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

Table 5: *Artix Designer Toolbar Buttons*

Button	Description
	Re-run the last-run Artix Tools configuration.[a]
	Import Artix demos into Artix Designer.
	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.
	Add `binding` element to currently selected WSDL file.
	Add `service` element to currently selected WSDL file.
	Add `route` element to currently selected WSDL file.
	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.

Artix Designer project types

In Eclipse, all development is performed within a project. When you create a 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

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 on the left side of the Table of Contents frame

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

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 for either the Java or C++ runtime
- Generate a client-server application
- Create a WSDL file's logical and physical elements
- Generate code for a service and deploy it to a 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. To view the available Artix Designer cheat sheets, select **Help|Cheat Sheets**.

Tutorial 1: Java First

Overview

This tutorial walks you through using the Java-first technique with Artix Designer to generate a “Hello World” client-server application written in Java using the JAX-WS API.

The Java code you start with resides here:

`ArtixInstallDir\java\samples\basic\java_first_jaxws`.

The generated code is written here:

`EclipseWorkspaceDir\javafirst-project`.

Tasks

This tutorial consists of the following tasks:

Task 1: Starting Artix Designer

To start Artix Designer:

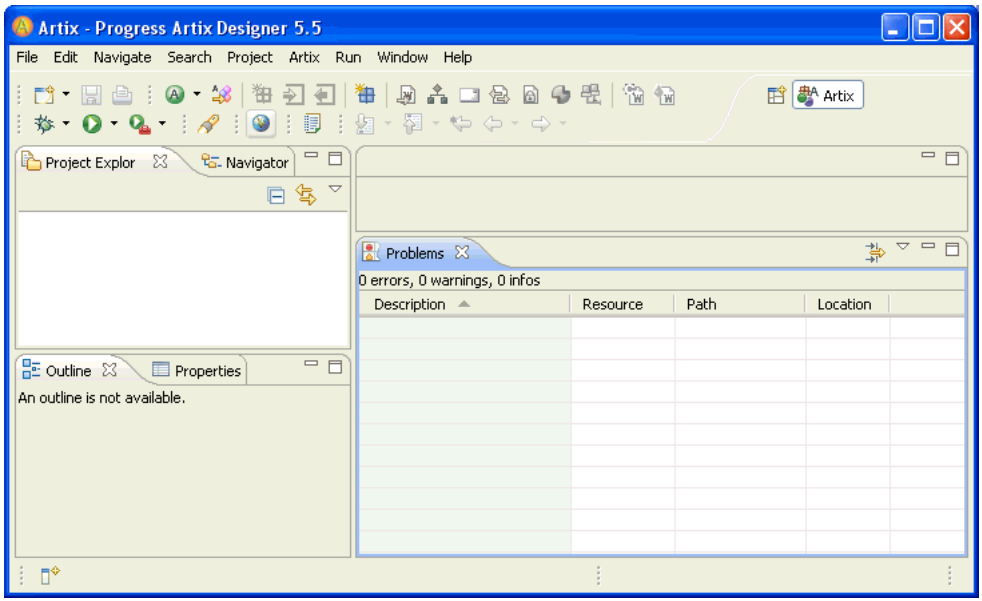
1. Use one of the following techniques:
   - On Windows, select **Start** | **[All Programs]** | **IONA** | **Artix *version*** | **Artix Designer**.
   - On Linux or Solaris, at a shell prompt, enter a command with the following syntax:

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

2. If prompted, specify a directory for the Eclipse workspace.

Artix Designer—that is, the Eclipse platform with the Artix Designer plug-ins loaded—launches, as shown in Figure 2.

Figure 2: *Artix Designer Newly Launched*

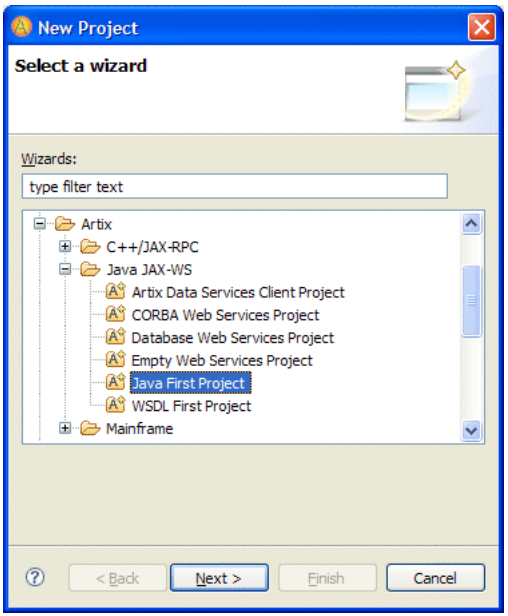


Task 2: Creating the project

To create an Artix Designer project for this tutorial:

1. From the main menu, select **File** | **New** | **Project**.
2. This opens the **Select a wizard** panel, as shown in Figure 3.

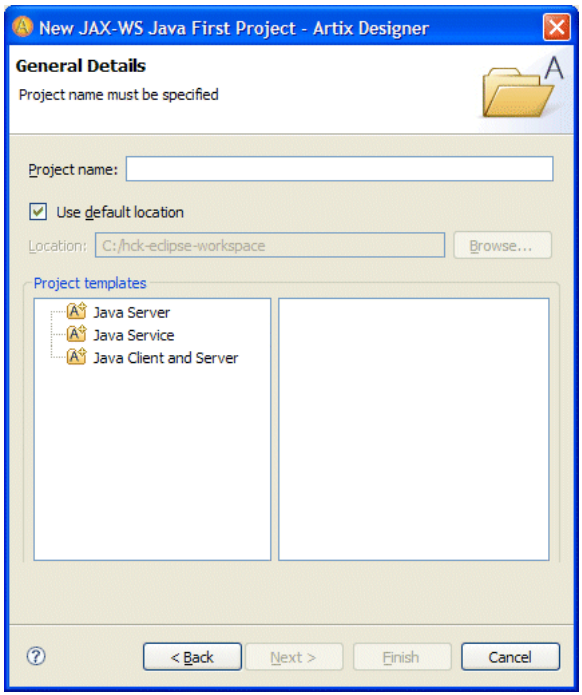
Figure 3: *Select a wizard Panel*



3. In the **Select a wizard panel**, navigate **Artix** | **Java JAX-WS** | **Java First Project**, then click **Next**.

This opens the **General Details** panel, as shown in Figure 4.

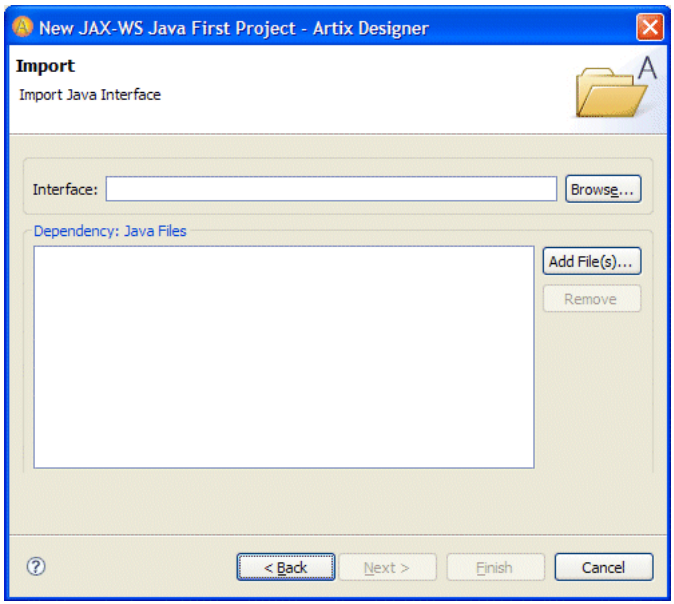
Figure 4: *General Details Panel*



4. In the **General Details** panel, do the following:
   i. For **Project Name**, type `javafirst-project`.
   ii. Under **Project templates**, click **Java Client and Server**.
   iii. Click **Next**.

This opens the **Import** panel, as shown in Figure 5.

Figure 5: *Import Panel*



Task 3: Importing the interface file

Import the Java interface file, which is supplied with the sample code, as follows:

1. In the **Import** panel, click **Browse**, and in the navigation window:
2. Navigate to *ArtixInstallDir*`\java\samples\basic\java_first_jaxws\src\demo\hw\server`.
3. Click **HelloWorld.java**.
4. Click **Open**.

   The navigation window closes.

Task 4: Generating the code

To generate the code, in the **Import** panel, click **Finish**. The code is generated and written to *EclipseWorkspaceDir*`\javafirst-project` or to a subordinate directory.

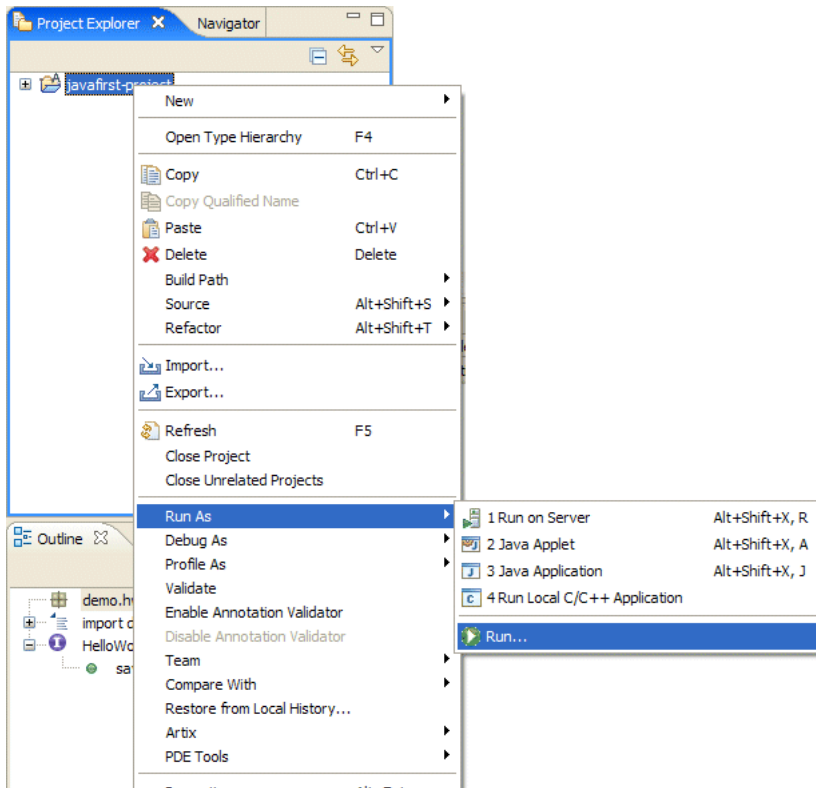
Task 5: Starting the server

To start the generated server:

1. In **Project Explorer**, right-click **javafirst-project**, and from the context menu:
   i. Select **Run As** | **Run**.
   ii. Click **Run**.

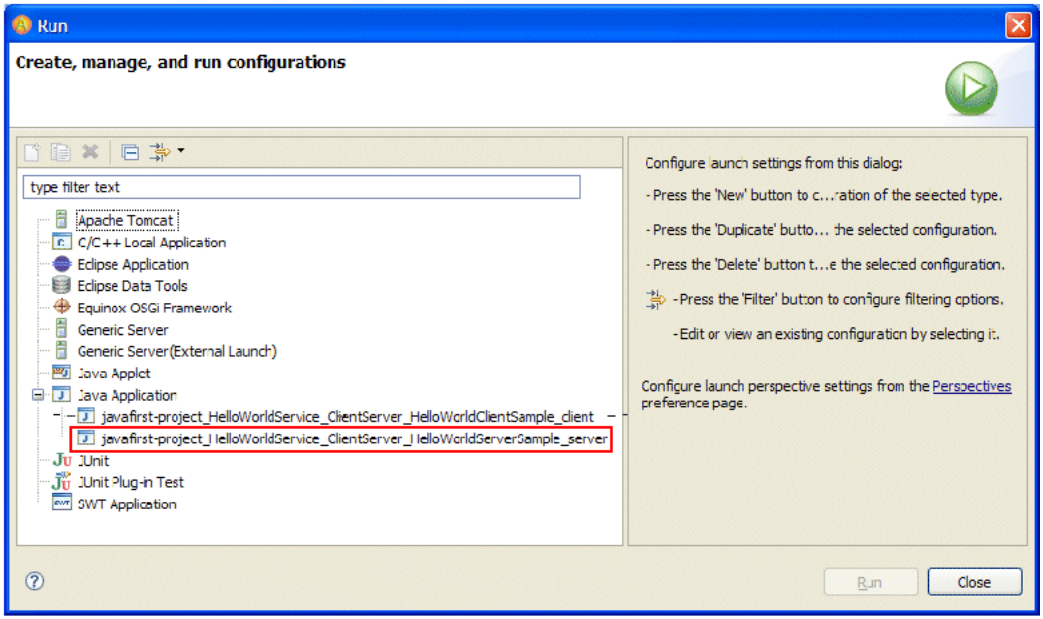
   This is shown in Figure 6.

Figure 6: *Starting the Generated Server*



This opens the **Create, manage, and run configurations** panel, as shown in Figure 7.

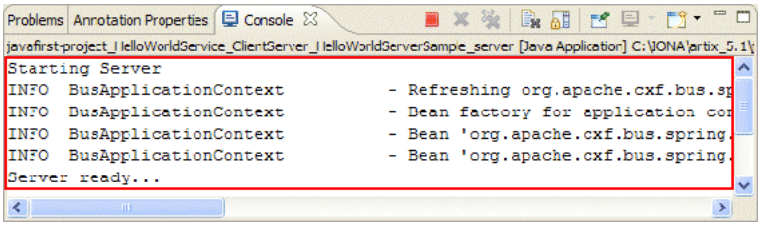
Figure 7: *Create, manage, and run configurations Panel*



2. In this panel, do the following:
   i. Navigate **Java Application** | **javafirst-project_HelloWorldService_ClientServer_HelloWorldServerSample_server**.
   ii. Click **Run**.

After a few seconds, the **Console** (located in the lower right of the window) displays a message indicating that the server is ready, as shown in Figure 8.

Figure 8: *Server Ready*



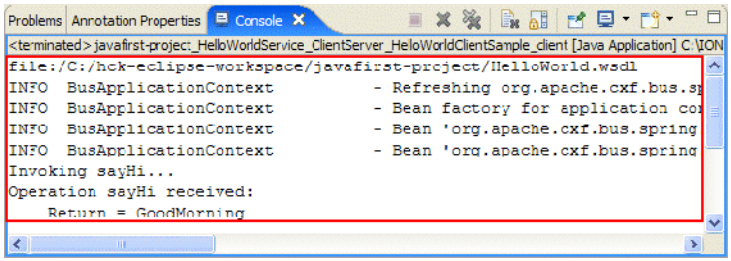
Task 6: Starting the client

In the same way, start the generated client. Specifically:

1. In **Project Explorer**, right-click the project name, and in the context menu, click **Run As** | **Run**.
2. In the **Create, manage, and run configurations** panel, do the following:
   i. Navigate to **Java Application** | **javafirst-project_HelloWorldService_ClientServer_HelloWorldServerSample_client**.
   ii. Click **Run**.

After a few seconds, the **Console** view displays messages indicating that the client ran OK, as shown in Figure 9.

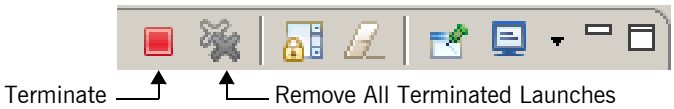
Figure 9: *Client Ran OK*



Task 7: Stopping the server and clearing the console

To stop the generated server and clear the **Console**, use the Eclipse-toolbar buttons shown in Figure 10.

Figure 10: *Eclipse Toolbar*



Specifically:

1. Clear the client output by clicking the **Remove All Terminated Launches** button.
2. Stop the server process by clicking the **Terminate** button.
3. Clear the server output by clicking the **Remove All Terminated Launches** button.

Tutorial 2: WSDL First, Starting with Filled-In WSDL

Overview

This tutorial walks you through using the WSDL-first technique with Artix Designer to generate a “Hello World” SOA-network client-server application written in Java that calls JAX-WS.

The WSDL file you start with, which is completely filled in, is *ArtixInstallDir*\java\samples\basic\wsdl_first\wsdl\helloworld.wsdl.

The generated code is written to *EclipseWorkspaceDir*\wsdlfirst-filledin-project or to a subordinate directory.

Tasks

This tutorial consists of the following tasks:

Task 1: Starting Artix Designer

Start Artix Designer if it is not already running. To do so:

1. Use one of the following techniques:
   - On in Windows, select **Start | [All Programs] |IONA| Artix *version*| Artix Designer**.

- On Linux or Solaris, at a shell prompt, enter a command with the following syntax:

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

2. If prompted, specify a directory for the Eclipse workspace.

   Artix Designer—that is, the Eclipse workspace with the Artix Designer plug-ins loaded.

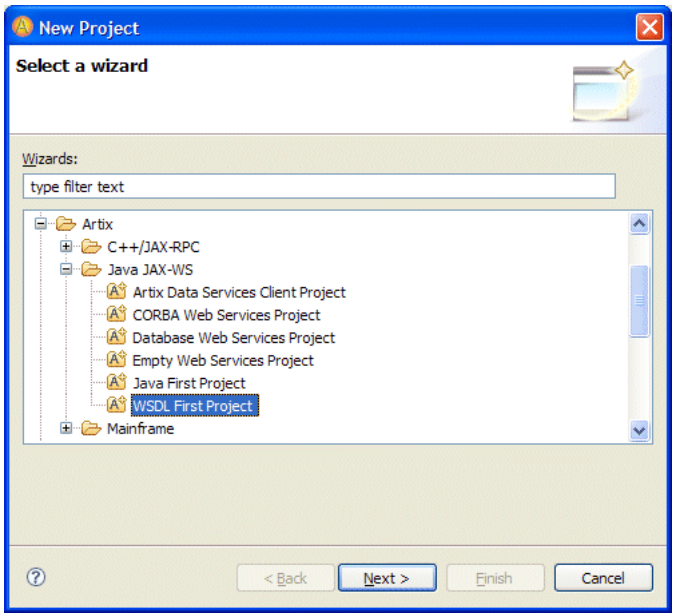
Task 2: Creating the project

To create a project for this tutorial:

1. From the main menu, navigate **File** | **New** | **Project**.

   This opens the **Select a wizard** panel, as shown in Figure 11.

Figure 11: *Select a wizard Panel*

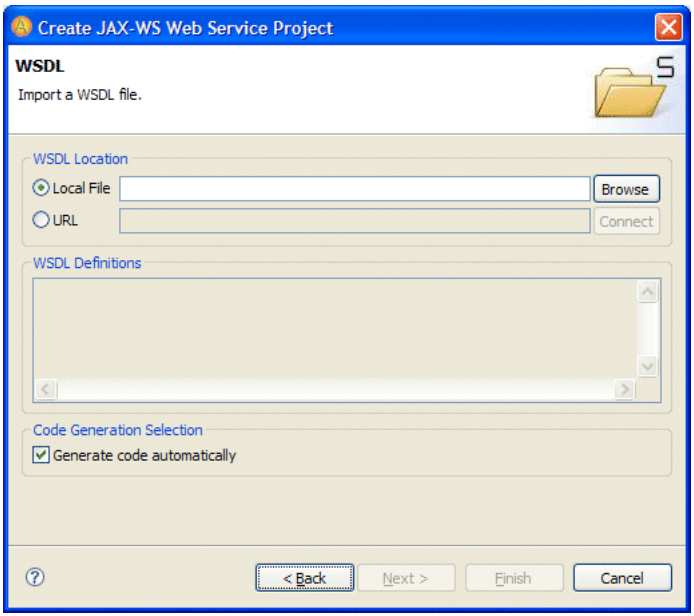


2. In the **Select a wizard panel**, navigate **Artix** | **Java JAX-WS** | **WSDL First Project**, then click **Next**.

3. In the **General Details** panel, do the following:
   i. For Project Name, type **`wsdlfirst-filledin-project`**.
   ii. Select **Use default location**.
   iii. Click **Next**.

This opens the **WSDL** panel, as shown in Figure 12.

Figure 12: *WSDL Panel*



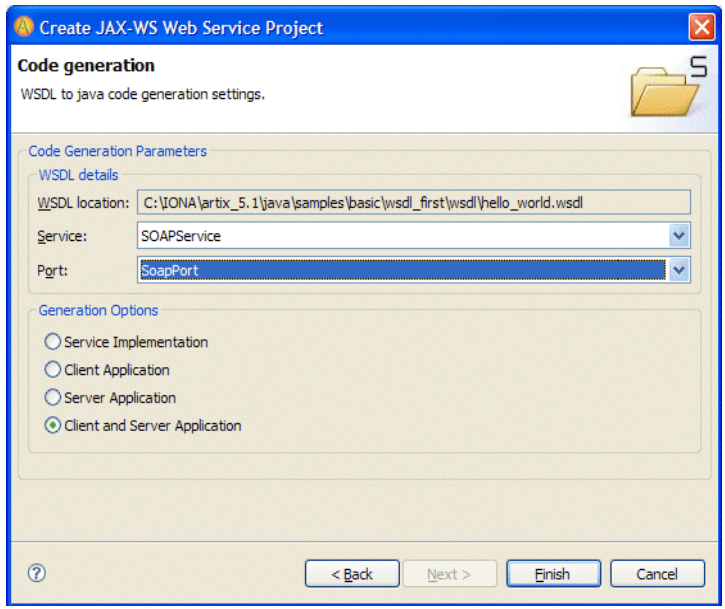
Task 3: Importing the WSDL file

To import the WSDL file:

1. In the **WSDL** panel, click **Browse**, which opens a navigation window.
2. In the navigation window, do the following:
   i. Navigate to
   ArtixInstallDir\java\samples\basic\wsdl_first\wsdl.
   ii. Click **HelloWorld.wsdl**.
   iii. Click **Open**.
   The navigation window closes.

3. Back in the **WSDL** panel, click **Next**, which opens the **Code generation** panel, as shown in Figure 13.

Figure 13: *Code generation Panel*



Task 4: Generating the code

To generate the code, in the **Code generation** panel, do the following:

1. For **Service**, select **SOAPService**.
2. For **Generation Options**, select **Client and Server Application**.
3. For **Port**, select **SoapPort**.
4. Click **Finish.**

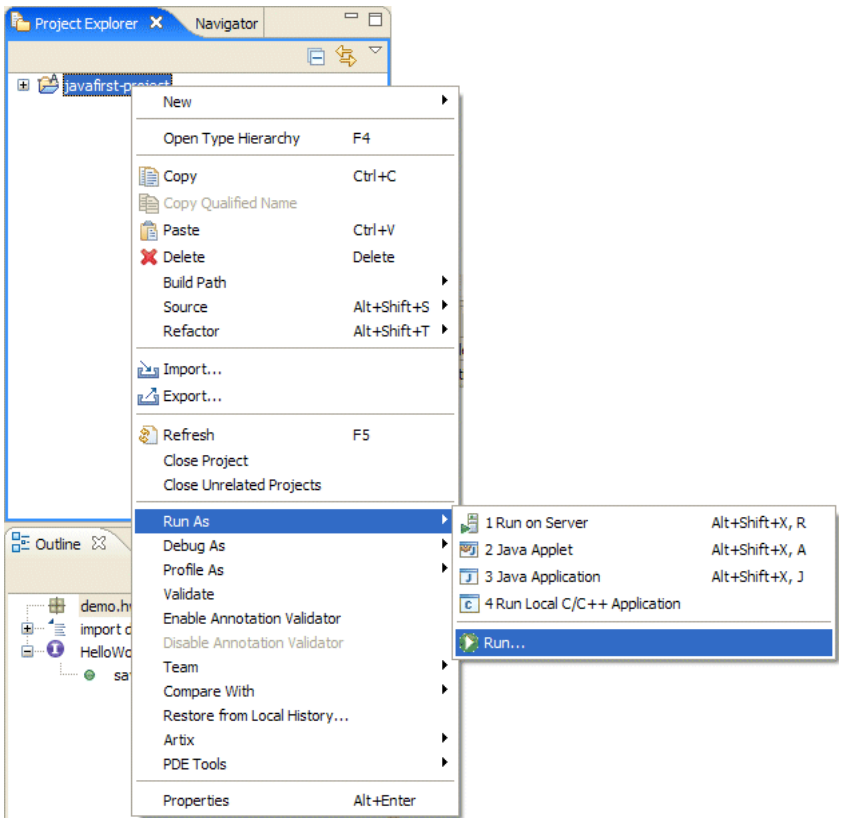
   Artix generates the code.

Task 5: Starting the server

To start the generated server:

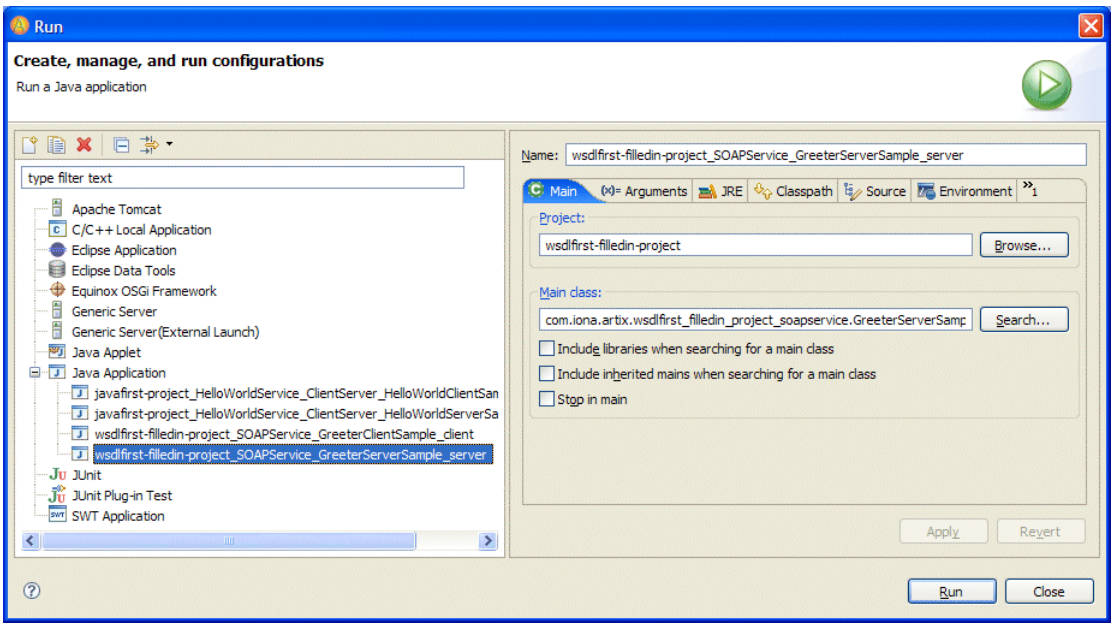
1. In Project Explorer, right-click the project name and in the context menu, select **Run As** | **Run**, as shown in Figure 14.

Figure 14: *Running the Application*



This opens the **Create, manage, and run configurations** panel, as shown in Figure 15.

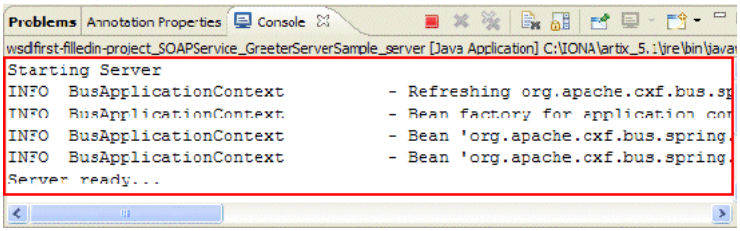
Figure 15: *Create, manage, and run configurations Panel*



2. In this panel, start the server by doing the following:
   i. Navigate **Java Application** | **wsdlfirst-project_SOAPService_GreeterServerSample_server**.
   ii. Click **Run**.

After a few seconds, the **Console** displays a message indicating that the server is ready, as shown in Figure 16.

Figure 16: *Server is Ready*



Task 6: Starting the client

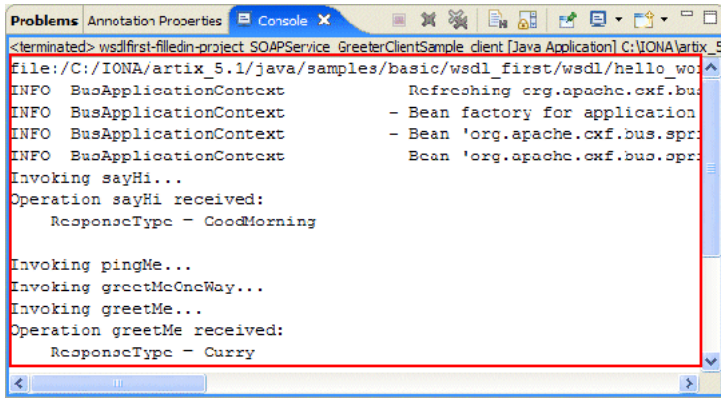
In the same way, start the generated client. Specifically:

1. In the **Project Explorer**, right-click the project name, and in the context menu, select **Run As** | **Run**.

   This opens the **Create, manage, and run configurations** panel.
2. In this panel, start the server by doing the following:
   i. Navigate **Java Application** | **wsdlfirst-project_SOAPService_GreeterServerSample_client**.
   ii. Click **Run**.

After a few seconds, the **Console** displays messages indicating that the client ran OK, as shown in Figure 17.

Figure 17: *Client Ran OK*



Task 7: Stopping the server and clearing the console

To stop the server and clear the console in this tutorial, do what you did in the previous tutorial in "Task 7: Stopping the server and clearing the console" on page 58.

Tutorial 3: WSDL First, Starting With Boilerplate WSDL

Overview

This tutorial walks you through using the WSDL-first technique with Artix Designer to generate three versions of a “Hello World” SOA-network client-server application:

- One written in Java that calls JAX-WS
- One written in Java that calls JAX-RPC
- One written in C++

Unlike Tutorial 2: WSDL First, Starting with Filled-In WSDL, this tutorial starts with a boilerplate WSDL file, which you fill in.

The generated code is written to `*EclipseWorkspaceDir*\wsdlfirst-boilerplate-project` or to a subordinate directory.

Tasks

This tutorial consists of the following tasks:

Task 1: Creating Empty Projects

Overview

In this task, you create an empty Artix Designer project for each version of the application: one for the Java/JAX-WS version, one for the Java/JAX-RPC version, and one for the C++ version.

Starting Artix Designer

To start and initialize Artix Designer:

1. Use one of the following techniques:

   On Windows, select **Start | [All Programs] |IONA| Artix *version*| Artix Designer**.

   On Linux or Solaris, at a shell prompt, enter a command with the following syntax:

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

2. If prompted, specify a directory for the Eclipse workspace.

   Artix Designer—that is, the Eclipse platform with the Artix Designer plug-ins loaded—launches.

Creating the project for the JAX-WS version

Create a project to contain the JAX-WS version of the application. To do so:

1. From the main menu, select **File** | **New** | **Project**.
2. In the **Select a Wizard** panel, select **Artix** | **Java JAX-WS** | **Empty Web Services Project**, then click **Next**.
3. In the **General Details** panel, do the following:
   i. For **Project name**, type **`JaxWsHello`**.
   ii. Select **Use default location**.
   iii. Click **Finish**

   Navigator now displays an entry for the JaxWsHello project.

Creating the project for the JAX-RPC version

Create a project to contain the JAX-RPC version of the application. To do so:

1. From the main menu, select **File** | **New** | **Project**.
2. In the **Select a Wizard** panel, select **Artix** | **C++/JAX-RPC** | **Empty Web Services Project**, then click **Next**.

3. In the **General Details** panel, do the following:
   i. For **Project name**, type `JaxRpcHello`.
   ii. Select **Use default location**.
   iii. Click **Finish**.

   Navigator now displays an entry for the JaxRpcHello project.

Creating a project for the C++ version

Create a project to contain the C++ version of the application. To do so:

1. From the main menu, select **File** | **New** | **Project**.
2. In the **Select a Wizard** panel, do the following:
   i. Select **Artix** | **C++/JAX-RPC** | **Empty Web Services Project**.
   ii. Click **Next**.
3. In the **General Details** panel, do the following:
   i. For **Project name**, type `CppHello`.
   ii. Select **Use default location**.
   iii. Click **Finish**.

   Navigator now displays an entry for the CppHello project.

Task 2: Creating a Boilerplate WSDL File

Overview

In this task, you create a boilerplate WSDL file within the JAX-WS project, then link to this WSDL file from the JAX-RPC project and from the C++ project.

Creating an empty WSDL file within the JAX-WS project

To create a skeleton WSDL file within the JAX-WS project:

1. From the main menu, select **File** | **New** | **WSDL File**.
2. For **parent folder**, click **JaxWeHello**.
3. For **File name**, type **`HelloWorld`**.
4. Click **Finish**.

   Artix Designer displays the boilerplate `HelloWorld.wsdl` file.

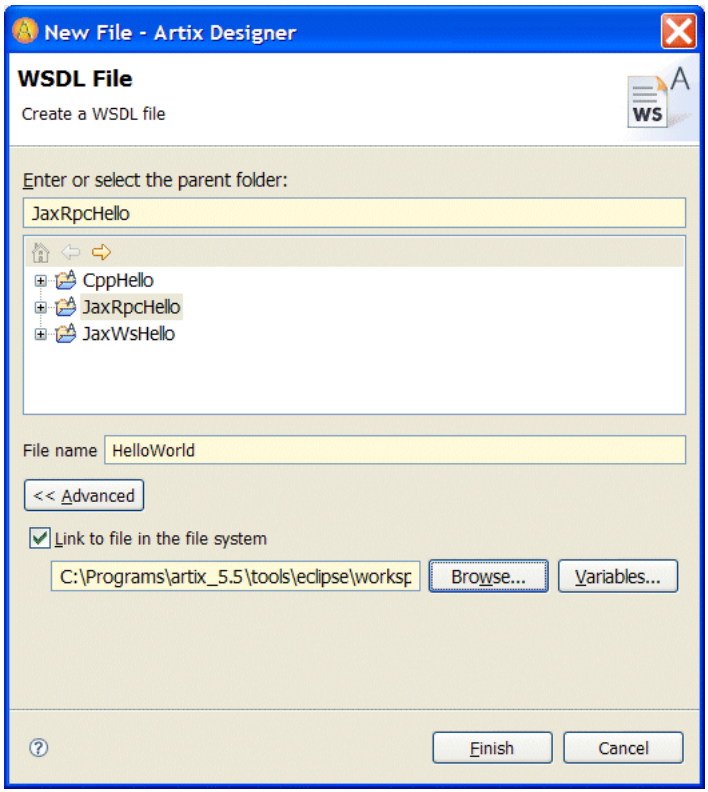
Linking to the WSDL file from the JAX-RPC project

To link to the new WSDL file from the JAX-RPC project:

1. From the main menu, select **File** | **New** | **WSDL File**, which opens the **WSDL File** panel.
2. For **parent folder**, click **JaxRpcHello**.
3. For **File name**, type **`HelloWorld`**.
4. Click **Advanced**.
5. Select **Link to file in the file system**, click **Browse**, and in the navigation window:
   i. Navigate to *`EclipseWorkspaceDir`*`/JaxWsHello`.
   ii. Select **HelloWorld.wsdl**.
   iii. Click **Open**.

The **WSDL File** panel now looks like Figure 18.

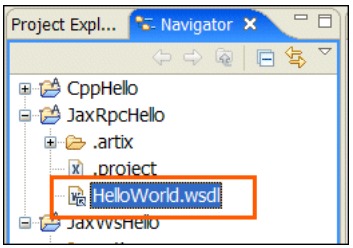
Figure 18: *WSDL File Panel*



6. In the **WSDL File** panel, click **Finish.**

In **Navigator**, under **JaxRpcHello**, the icon for `HelloWorld.wsdl` now indicates a link, as shown in Figure 19.

Figure 19: *HelloWorld.wsdl as a Link*



Linking to the WSDL file from the C++ project

To link to the new WSDL file from the C++project:

1. From the main menu, select **File** | **New** | **WSDL File**, which opens the **WSDL File** panel.
2. For **parent folder**, click **CppHello**.
3. For **File name**, type **`HelloWorld`**.
4. Click **Advanced**.
5. Select **Link to file in the file system**, click **Browse**, and in the navigation window:
   i. Navigate to *`EclipseWorkspaceDir`*`/JaxWsHello`.
   ii. Click **HelloWorld.wsdl**.
   iii. Click **Open**.
6. Back in the **WSDL File** panel, click **Finish.**

   In **Navigator**, under **CppHello**, the icon for `HelloWorld.wsdl` now indicates a link.

Task 3: Defining Types in the WSDL File

Overview

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

In this task, you create two objects of type `string`:

- `InElement`, which maps to the *in* part of the request message.
- `OutElement`, which maps to the *out* part of the response message.

Defining InElement

To define `InElement`:

1. In **Navigator**, navigate **JaxWsHello** | **HelloWorld.wsdl**, then double-click **HelloWorld.wsdl**.
2. At the bottom of the **WSDL Editor** view, click the **Diagram** tab.
3. In the **Diagram** view, do the following:
   i. Right-click **Types**.
   ii. In the context menu, select **New Type**.
4. In the **Select Source Resources** panel, do the following:
   i. Under **Source File(s)**, select **HelloWorld.wsdl**.
   ii. Click **Next**.
5. In the **Define Type Properties** panel, do the following:
   i. For **Name**, type **`InElement`**.
   ii. For **Kind**, select **Element**.
   iii. Click **Next**.
6. In the **Define Element Data** panel, do the following:
   i. For **Type Definition**, select **Pre-declared Type** and **string**.
   ii. Click **Next**.
7. In the **View Type Summary** panel, click **Finish**.
8. Save the changes by pressing **`CTRL-S`**.

Defining OutElement

To define `OutElement`:

1. In **Navigator**, navigate **JaxWsHello** | **HelloWorld.wsdl**, then double-click **HelloWorld.wsdl**.
2. At the bottom of the **WSDL Editor** view, click the **Diagram** tab.
3. In the **Diagram** view, do the following:
   i. Right-click **Types**.
   ii. In the context menu, select **New Type**.
4. In the **Select Source Resources** panel, do the following:
   i. Under **Source File(s)**, select **HelloWorld.wsdl**.
   ii. Click **Next**.
5. In the **Define Type Properties** panel, do the following:
   i. For **Name**, type **`OutElement`**.
   ii. For **Kind**, select **Element**.
   iii. Click **Next**.
6. In the **Define Element Data** panel, do the following:
   i. For **Type Definition**, select **Pre-declared Type** and **string**.
   ii. Click **Next**.
7. In the **View Type Summary** panel, click **Finish**.
8. Save the changes by pressing **`CTRL-S`**.

Review

At the bottom of the **WSDL Editor**, click the **Source** tab to view the WSDL file created so far.

In the lower left of the Eclipse window, in the **Outline** view , 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>
```

Task 4: Defining Messages in the WSDL File

Overview

In this task you define the request and response messages for your Web service.

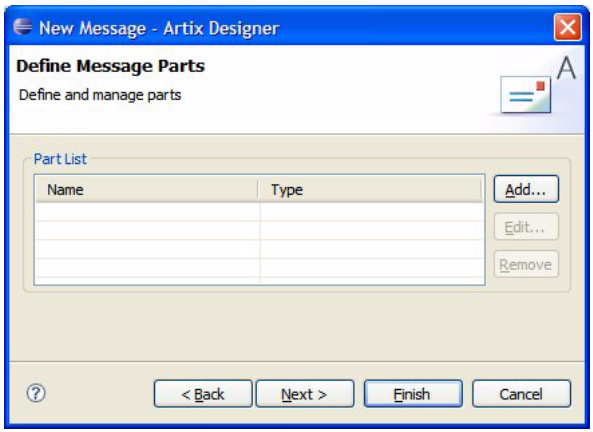
For message parts, you will use the types you already created.

Defining the request message

To define the request message:

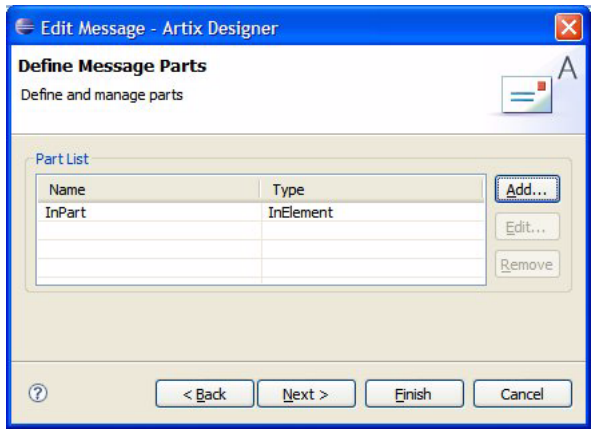
1. With the `HelloWorld.wsdl` file open and the **Diagram** view displayed, right-click **Messages**, and in the context menu, select **New Message**.
2. In the **Select Source Resources** panel, do the following:
   i. For **Source File(s)**, select **HelloWorld.wsdl**.
   ii. Click **Next**.
3. In the **Define Message Properties** panel, do the following:
   i. For **Name,** type **`RequestMessage`**.
   ii. Click **Next.**
4. In the **Define Message Parts** panel, click **Add**, as shown in Figure 20.

Figure 20: *Define Message Parts Panel*



5. In the **Message Part Data** dialog, do the following:
   i. For **Name**, type `InPart`.
   ii. For **Type**, select **InElement**.
   iii. Click **OK**, which adds `InPart` to the **Part List** in the **Define Message Parts** panel, as shown in Figure 21.

Figure 21: *Define Message Parts Panel (with InPart)*



6. In the **Define Message Parts** panel, click **Next**, which opens the **View Message Summary** panel.
7. In the **View Message Summary** panel, click **Finish**.
8. Save the changes by pressing **CTRL-S**.

Defining the response message

To define the response message:

1. With the `HelloWorld.wsdl` file open and the **Diagram** view displayed, right-click **Messages**.
2. From the pop-up menu, select **New Message.**
3. In the **Select Source Resources** panel, do the following:
   i. For **Source File(s)**, select **HelloWorld.wsdl**.
   ii. Click **Next**.
4. In the **Define Message Properties** panel, do the following:
   i. For **Name,** type **`ResponseMessage`**.
   ii. Click **Next.**
5. In the **Define Message Parts** panel, click **Add**.
6. In the **Message Part Data** dialog, do the following:
   i. For **Name**, type **`OutPart`**.
   ii. For **Type**, select **OutElement**.
   iii. Click **OK**, which adds `OutPart` to the Part List in the **Define Message Parts** panel.
7. Back in the **Define Message Parts** panel, click **Next**, which opens the **View Message Summary** panel.
8. In the **View Message Summary** panel, click **Finish**.
9. Save the changes by pressing **`CTRL-S`**.

Review

You 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., as shown in the following fragment:

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

For complete information on creating messages, see Understanding Artix Contracts.

Task 5: Defining Port Types in the WSDL File

Overview

The `portType` element contains operations, each of which includes one or more messages, as follows:

- A one-way operation includes an input message only; 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.

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

Defining a port type

To define a port type:

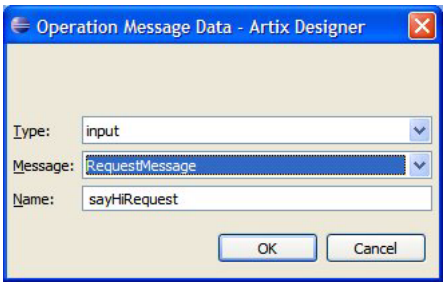
1. With the `HelloWorld.wsdl` file open and the **Diagram** view displayed, right-click **Port Types**, and in the context menu, select **New Port Type**.
2. In the **Select Source Resources** panel, do the following:
   i. For **Source File(s)**, select **HelloWorld.wsdl**.
   ii. Click **Next**.
3. In the **Define Port Type Properties** panel, do the following:
   i. For **Name**, type **`HelloWorldPT`**.
   ii. Click **Next.**
4. In the **Define Port Type Operations** panel, do the following:
   i. For **Name**, type **`sayHi`**.
   ii. For **Style**, select **Request-response**.
   iii. Click **Next**.

Defining the request message

Define the request message as follows:

1. In the **Define Operation Messages** panel, click **Add**, which opens the **Operation Message Data** dialog.
2. In the **Operation Message Data** dialog, do the following:
   i. For **Type**, select **input**.
   ii. For **Message**, select **RequestMessage**.
   The name **`SayHiRequest`** appears in the **Name** text box, as shown in Figure 22.

Figure 22: *Operation Message Data Dialog*



   iii. Click **OK,** which adds the operation to the **Operation Messages** list in the **Define Operation Messages** panel.

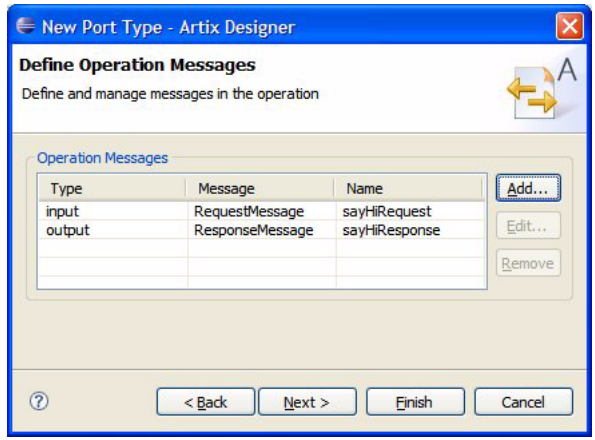
Defining the response message

Define the response message as follows:

1. Back in the **Define Operation Messages** panel, click **Add** again.
2. In the **Operation Message Data** dialog, do the following:
   i. For **Type**, select **output**.
   ii. For **Message**, select **ResponseMessage**.
   The name `sayHiResponse` appears in the **Name** text box.
   iii. Click **OK**.

   The Define Operation Messages panel now looks like Figure 23.

Figure 23: *Define Operation Messages Panel*



3. Back in the **Define Operation Messages** panel, click **Next**, which opens the **View Port Type and Operation Summary** panel.
4. In the **View Port Type and Operation Summary** panel, click **Finish**.
5. Save the changes by pressing `CTRL-S`.

Review

You 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>
```

Task 6: Defining Bindings in the WSDL File

Overview

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

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

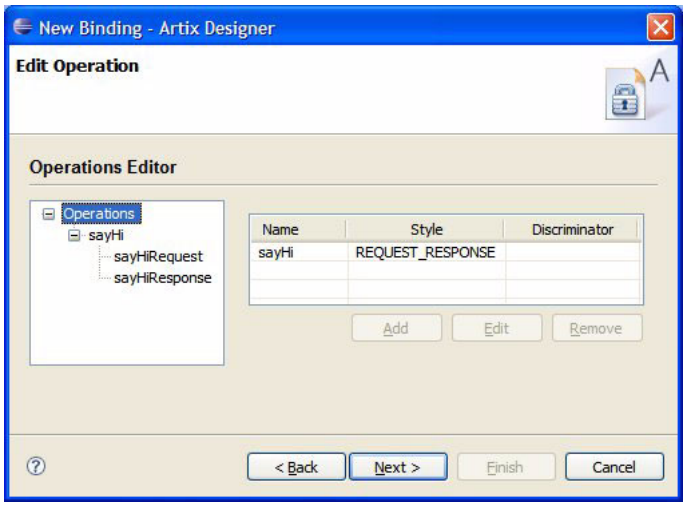
Defining a binding

To define a binding:

1. With the `HelloWorld.wsdl` file open and the **Diagram** view displayed, right-click **Bindings**, and in the context menu, select **New Binding**.
2. In the **Select Source Resources** panel, do the following:
   i. Select **HelloWorld.wsdl**.
   ii. Click **Next**.
3. In the **Select Binding Type** panel, do the following:
   i. For **Binding Type**, select **SOAP 1.1**.
   ii. Click **Next**.
4. In the **Set Binding Defaults** panel, do the following:
   i. For **Port Type**, select **HelloWorldPT**.
   ii. For **Style**, select **document**.
   iii. For **Use**, select **literal**.
   iv. Click **Next**.

5. In the **Edit Operation** panel, do the following:
   i. In the **Operations Editor** on the left, expand the **Operations** node, as shown in Figure 24.

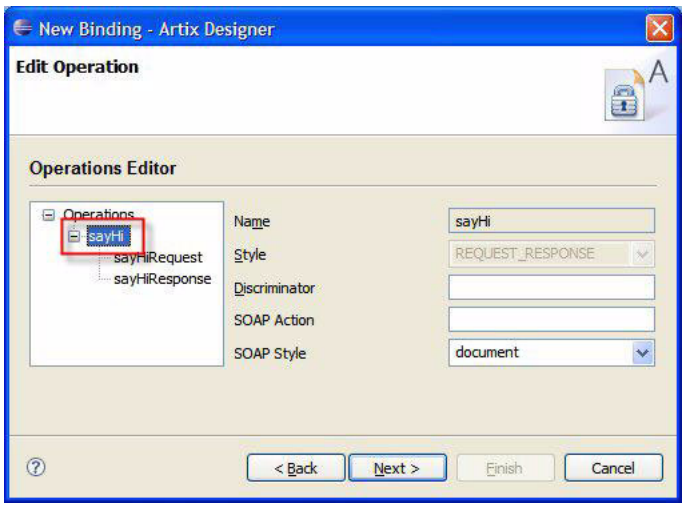
Figure 24: *Edit Operation panel*



   ii. Click each **sayHi** operation to review its binding.

The panel now looks like Figure 25.

Figure 25: *Edit Operation Panel—sayHi Node Selected*



iii. Click **Next**, which opens the **View Binding Summary** panel.

iv. Click **Finish**, which closes the wizard.

6. Save the changes by pressing **CTRL-S**.

Review

You 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>
```

Task 7: Defining a Service in the WSDL File

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 maps to a single `binding` element, although the same `binding` element can map to one or more `port` elements. In addition, a WSDL file can contain multiple `service` elements.

In this task, you add to the WSDL file one `service` element that contains one `port` element.

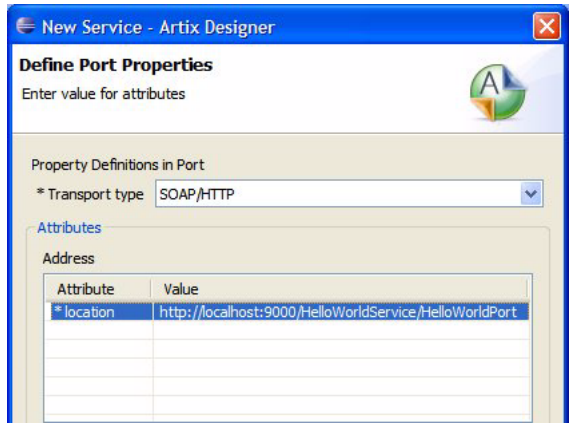
Defining a service

To define a service:

1. With the `HelloWorld.wsdl` file open and the Diagram view displayed, right-click **Services**, and in the context menu, select **New Service**.
2. In the **Select Source Resources** panel, do the following:
   i. For **Source File(s)**, select **HelloWorld.wsdl**.
   ii. Click **Next**.
3. In the **Define Service** panel, do the following:
   i. For **Name**, type **`HelloWorldService`**.
   ii. Click **Next**.
4. In the **Define Port** panel, do the following:
   i. For **Name**, type **`HelloWorldPort`**.
   ii. For **Binding**, select **HelloWorldPTSOAPBinding**.
   iii. Click **Next**.

5. In the **Define Port Properties** panel, do the following:
   i. For Transport Type, select **SOAP/HTTP**.
   ii. For the `location` attribute, click just below the Value header, and type **`http://localhost:9000/HelloWorldService/HelloWorldPort`**, as shown in Figure 26.

Figure 26: *Define Port Properties Panel*



   iii. Click **Next**, which opens the **View Service and Port Summary** panel.
6. Click **Finish**, which closes the wizard.
7. Save the changes by pressing **`CTRL-S`**.

Review

You have completed your WSDL contract. To review the WSDL, in the WSDL Editor, click **Source**. The WSDL should look like Example 9.

Example 9: *HelloWorld.wsdl Filled In Completely*

```
<?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">
  <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>
```

Example 9: *HelloWorld.wsdl Filled In Completely (Continued)*

```
  <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>
```

Task 8: Turning on the Build Automatically Option

Overview

Because Artix Tools integrate with the Eclipse JDT and CDT, Artix can automatically compile Java or C++ code as soon as it is generated and can automatically recompile any Java or C++ code as soon as it is changed and saved. This requires the Build Automatically option to be on.

Turning on Build Automatically

To turn on the Build Automatically option:

1. From the main menu, navigate **Project** | **Build Automatically**.

   If **Build Automatically** has a check to the left of it, the option is on.

2. If the option is off, click **Build Automatically** to turn the option on.

Note: If you are running the Windows version of Artix Designer and generating C++ code, in order for Build Automatically to work, before you start Artix Designer, you must set the Windows environment for the version of Visual C++ invoked thorugh Artix Designer. For details, see the Artix Installation Guide.

Task 9: Generating Code

Overview

Before you can generate code, Artix Designer prompts you to define a code-generation configuation. Once you create one, you can copy it to a different name and then edit and save the copy.

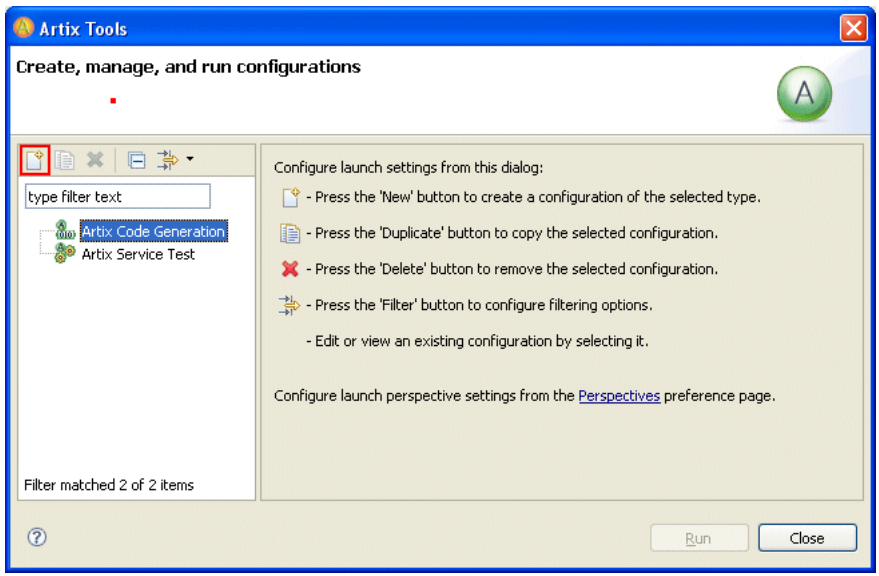
In this task, you create a code-generation configuration from scratch for JAX-WS, clone the original for JAX-RPC, and clone the original a second time for C++.

Creating the code-generation configuration for JAX-WS

To create the code-generation configuration for the JAX-WS project:

1. From the main menu, select **Artix | Artix Tools | Artix Tools**, which opens the **Create, manage, and run configurations panel**, as shown in Figure 27.

Figure 27: *Create, manage, and run configurations panel*

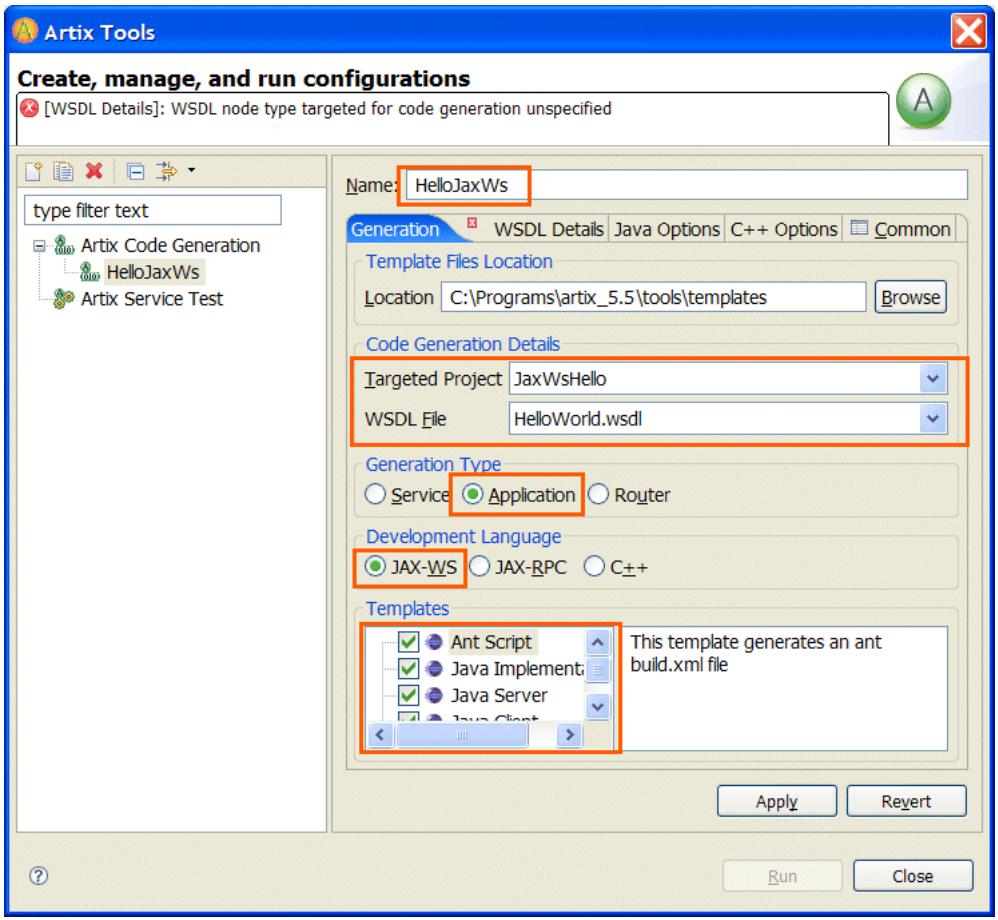


2. Select **Artix Code Generation.**

3. In the toolbar just above the **Artix Code Generation** label, click the **New launch configuration** button (marked in red in Figure 27).

   The right-pane displays a series of tabs with the **Generation** tab highlighted, as shown in Figure 28

Figure 28: *Generation Tab Highlighted*

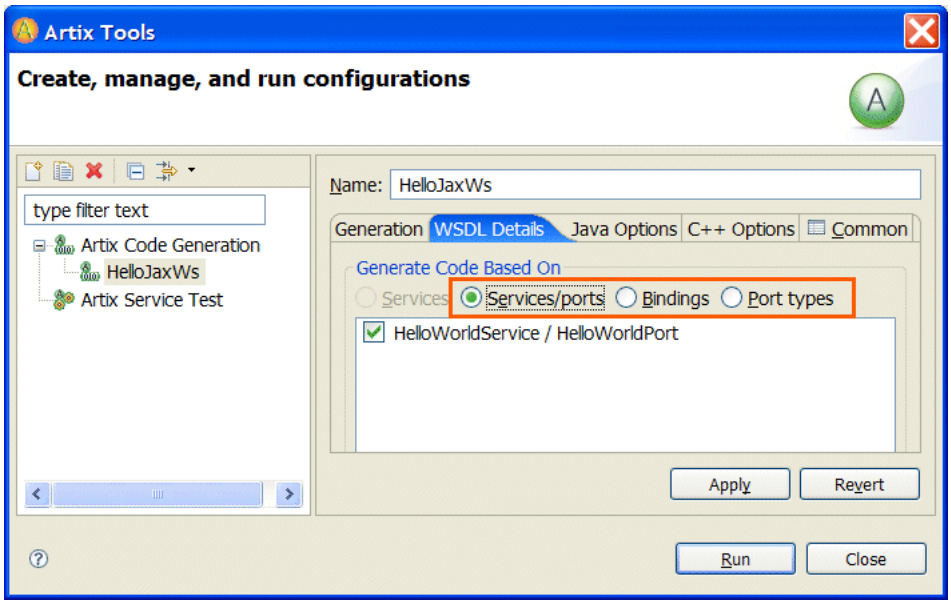


4. For **Name**, type `HelloJaxWs`.

5. In the **Generation** tabbed page, do the following:
   i. For **Targeted Project**, select **JaxWsHello**.
   ii. For **WSDL File**, select **HelloWorld.wsdl**.
   iii. For **Generation Type**, select **Application**.
   iv. For **Development Language**, select **Jax-WS**.
   v. For **Templates**, check all checkboxes.
6. Click the **WSDL/Containers Details** tab.

The **WSDL Details** tabbed page opens, as shown in Figure 29.

Figure 29: *WSDL Details Tabbed Page*



7. In the **WSDL Details** tabbed page, do the following:
   i. For **Services/ports**, select **HelloWorldService/HelloWorldPort**.
   ii. For **Bindings**, select **HelloWorldPTSOAPBinding**.
   iii. For **Port Types**, select **HelloWorldPT**.
   iv. Click **Apply**, which saves the code-generation configuration.

Generating the JAX-WS code

To generate JAX-WS Java code from the saved code-generation configuration, in the **Create, manage, and run configurations** panel, click **Run**.

Artix tools generate all source code—Java classes and configuration files—for the client-server application automatically.

Eclipse compiles the generated code automatically.

By default, the generated source code is written to the following location:

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

Also by default, the compiled bytecode is written to the following location:

```
EclipseWorkspaceDir/JaxWsHello/bin
```

Creating a code-generation configuration for JAX-RPC

Create a code-generation configuration for JAX-RPC by cloning the one for JAX-WS. To do so:

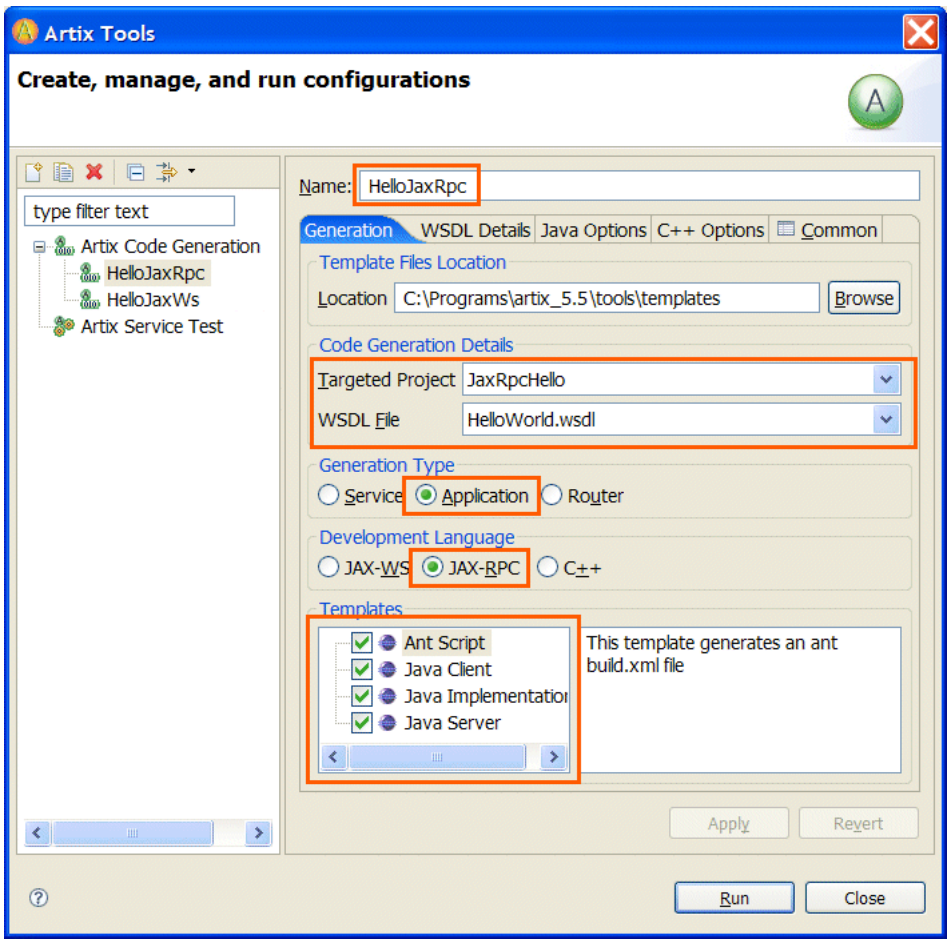
1. In the Eclipse menu, select the **Artix| Artix Tools | Artix Tools**.
2. In the **Artix Tools** window, select **HelloJaxWs**.
3. From the toolbar, click the **Duplicate launch configuration** button, shown in Figure 30.

Figure 30: *Duplicate Launch Configuration Button*

4. For **Name**, type `HelloJaxRpc`.
5. In the **Generation** tabbed page, do the following:
   i. For **Targeted Project**, select **JaxRpcHello**.
   ii. For **WSDL File**, select **HelloWorld.wsdl**.
   iii. For **Generation Type**, select **Application**.
   iv. For **Development Language**, select **JAX-RPC**.
   v. For **Templates**, check all checkboxes.

The **Generation** tabbed page now looks like Figure 31.

Figure 31: *Generation Tabbed Page*



vi. Click **Apply**, which saves the code-generation configuration.

Generating the JAX-RPC code

To generate JAX-RPC code from the JAX-RPC code-generation configuration, in the **Create, manage, and run configurations panel**, click **Run**.

Creating a code-generation configuration for C++

Create a code-generation configuration for C++ by cloning the one for JAX-WS.

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. For more information, see the Artix Installation Guide.

To create a code-generation configuration for C++ by cloning the one for JAX-WS:

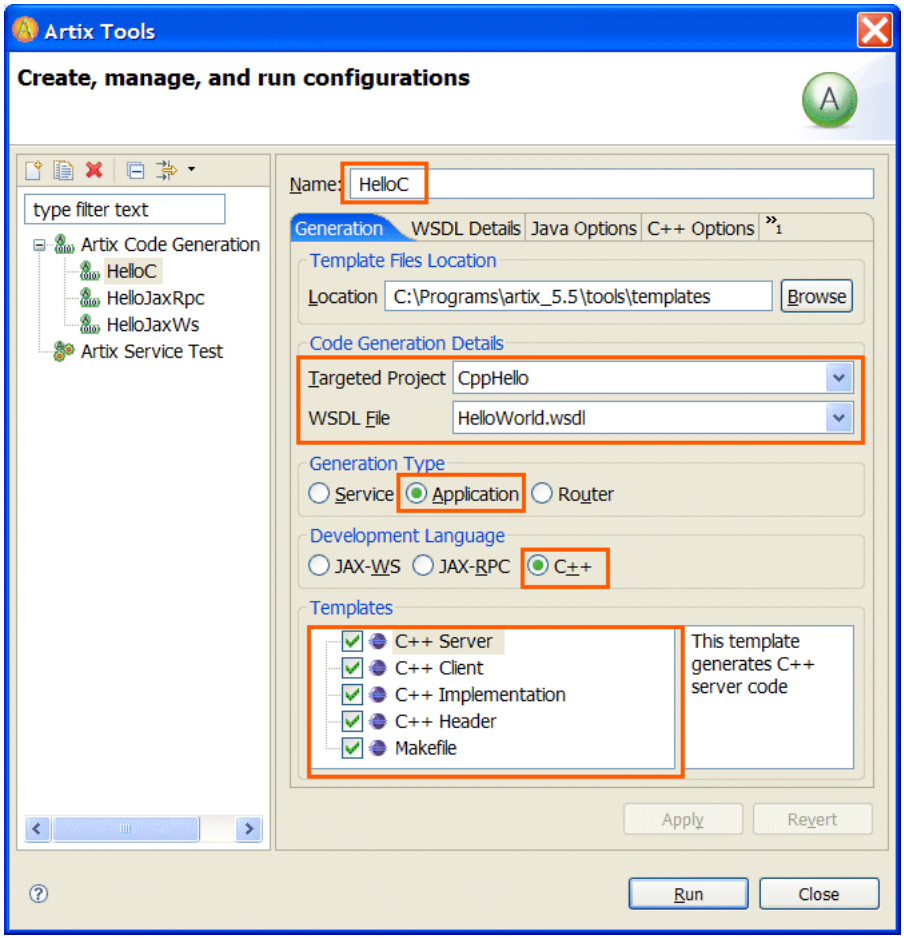
1. In the Eclipse menu, select the **Artix| Artix Tools | Artix Tools** menu.
2. In the Artix Tools window, select the **HelloJaxWs** configuration.
3. From the toolbar, click the **Duplicate Launch Configuration** button, shown in Figure 32.

Figure 32: *Duplicate Launch Configuration Button*

4. In the **Name** text box, change the name to `HelloCpp`.
5. In the **Generation** tabbed page, do the following:
   i. For **Targeted Project**, select **CppHello**.
   ii. For **WSDL File**, select **HelloWorld.wsdl**.
   iii. For **Generation**, select **Application**.
   iv. For **Development Language**, select **C++**.
   v. For **Templates**, check every checkbox.

The Generation tabbed page now looks like Figure 33.

Figure 33: *Generation Tabbed Page*

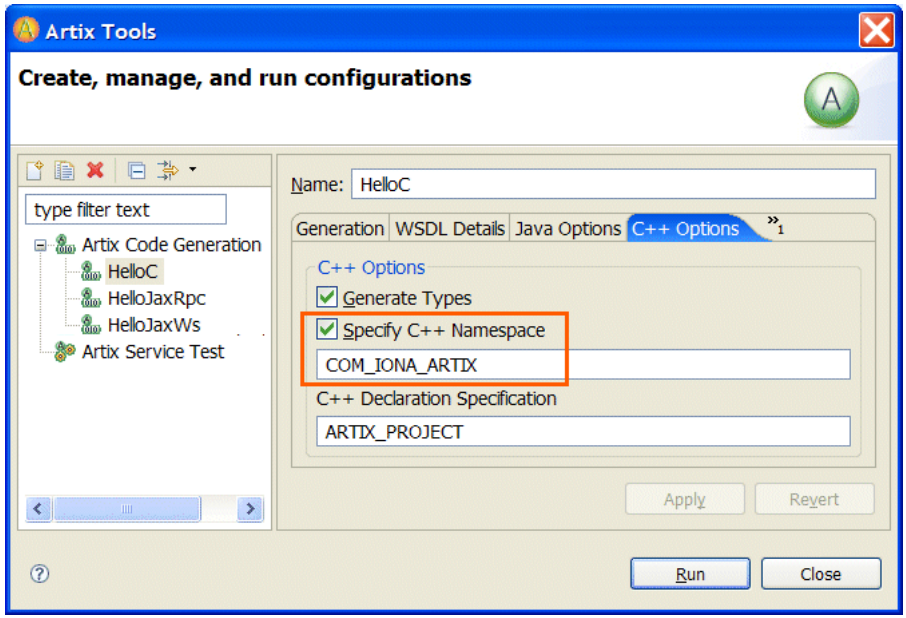


vi. Click **Apply**.

6. Click the **WSDL Details** tab. In the tabbed page, do the following:
   i. For **Services/ports**, select **HelloWorldService/HelloWorldPort**.
   ii. For **Bindings**, select **HelloWorldPTSOAPBinding**.
   iii. For **Port Types**, select **HelloWorldPT**.
   iv. Click **Apply**, which saves the code-generation configuration.
7. Click the **C++ Options** tab. In the tabbed page, do the following:
   i. For **Specify C++ Namespace**, accept the default (COM_IONA_ARTIX).
   ii. For **Port Types**, select **HelloWorldAcceptPT**.

   The tabbed page should look like Figure 34.

Figure 34: *C++ Options Tabbed Page*



   iii. Click **Apply**, which saves the code-generation configuration.

Generating the C++ code

To generate C++ code from the code-generation configuration you just created and saved, in the **Create, manage, and run configurations** panel, click **Run**.

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

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

Task 10: Running the Applications

Overview

You are now ready to run the JAX-WS, JAX-RPC, and C++ versions of the client-server application.

You can launch each version from within the Eclipse environment, although the procedures for each version are different.

Editing run configurations

Artix Designer automatically creates a run configuration for each generated executable. You can edit and save a run configuration in much the same way as you edited and saved code-generation configurations in "Task 9: Generating Code" on page 91.

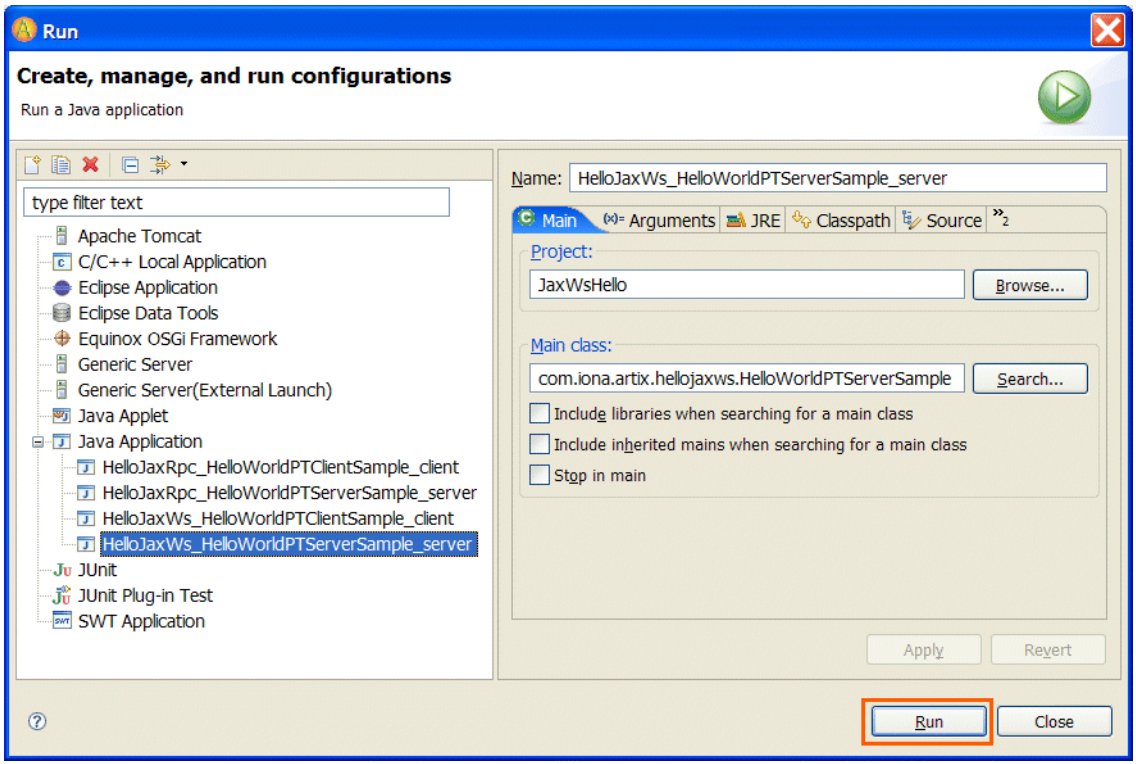
A saved run configuration saves time when rerunning your application, because it saves the environment and any arguments necessary for each invocation. You can copy a saved run configuration and edit it to create a new run configuration.

Running the JAX-WS server

To run the JAX-WS server:

1. Right-click the **JaxWsHello** project folder and in the context menu, select **Run As|Run**.
2. In the **Run** dialog, do the following:
   i. In the list of run configurations on the left, select **Java Application | HelloJaxWs_HelloWorldPTServerSample_server**, as shown in Figure 35.

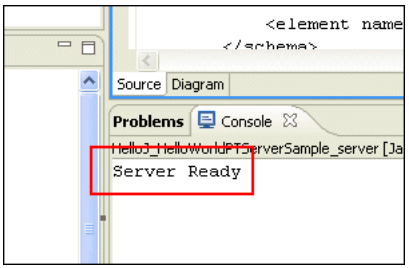
Figure 35: *Run Dialog*



ii. Click **Run**.

The server process starts running, displaying messages in the **Eclipse Console** view. After a moment, a message indicating that the server is ready appears, as shown in Figure 36.

Figure 36: *JAX-WS Server Ready*



Running the JAX-WS client

To run the JAX-WS client:

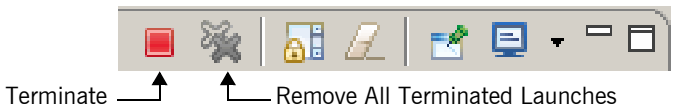
1. Right-click the **JaxWsHello** project folder and in the context menu, select **Run As|Run**.
2. In the **Run** window, do the following:
   i. In the list of Java launch configurations, select **Java Application | HelloJaxWs_HelloWorldPTClientSample_client**.
   ii. Click **Run**.

   The message, `Operation sayHi received: Curry`, appears in the **Eclipse Console** view.

Stopping the JAX_WS server and clearing the console

To stop the JAX-WS server and clear the console, use the Eclipse-toolbar buttons shown in Figure 10.

Figure 37: *Eclipse Toolbar*



Specifically:

1. Clear the client output by clicking the **Remove All Terminated Launches** button.
2. Stop the server process by clicking the **Terminate** button.
3. Clear the server output by clicking the **Remove All Terminated Launches** button.

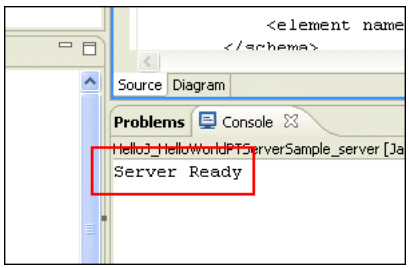
Running the JAX-RPC server

To run the JAX-RPC server:

1. Right-click the **JaxRpcHello** project folder and in the context menu, select **Run As|Run**.
2. In the **Run** dialog, do the following:
   i. From the list of run configurations on the left, select **Java Application | HelloJaxRpc_HelloWorldPTServerSample_server**.
   ii. Click **Run**.

   The server runs. In a few seconds, the **Eclipse Console** view displays the message, `Server Ready`, as shown in Figure 38.

Figure 38: *JAX-RPC Server Ready*



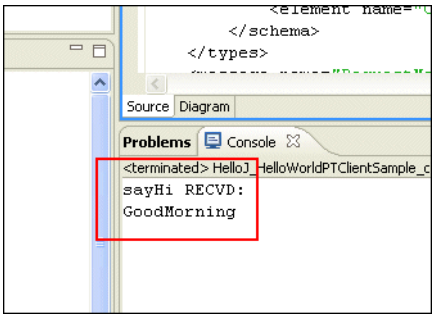
Running the JAX-RPC client

To run the JAX-RPC client:

1. Right-click the **JaxRpcHello** project folder and in the context menu, select **Run As|Run**.
2. Interact with the **Run** window as follows:
   i. From the list of Java launch configurations, select **Java Application | HelloJaxRpc_HelloWorldPTClientSample_client**.
   ii. Click **Run**.

   The message `say Hi RECVD: GoodMorning` appears in the Eclipse Console view, as shown in Figure 39.

Figure 39: *JAX-RPC Client Run-Messages*



Stopping the JAX_RPC server and clearing the console

To stop the JAX-RPC server and clear the console, follow the instructions in "Stopping the JAX_WS server and clearing the console" on page 105.

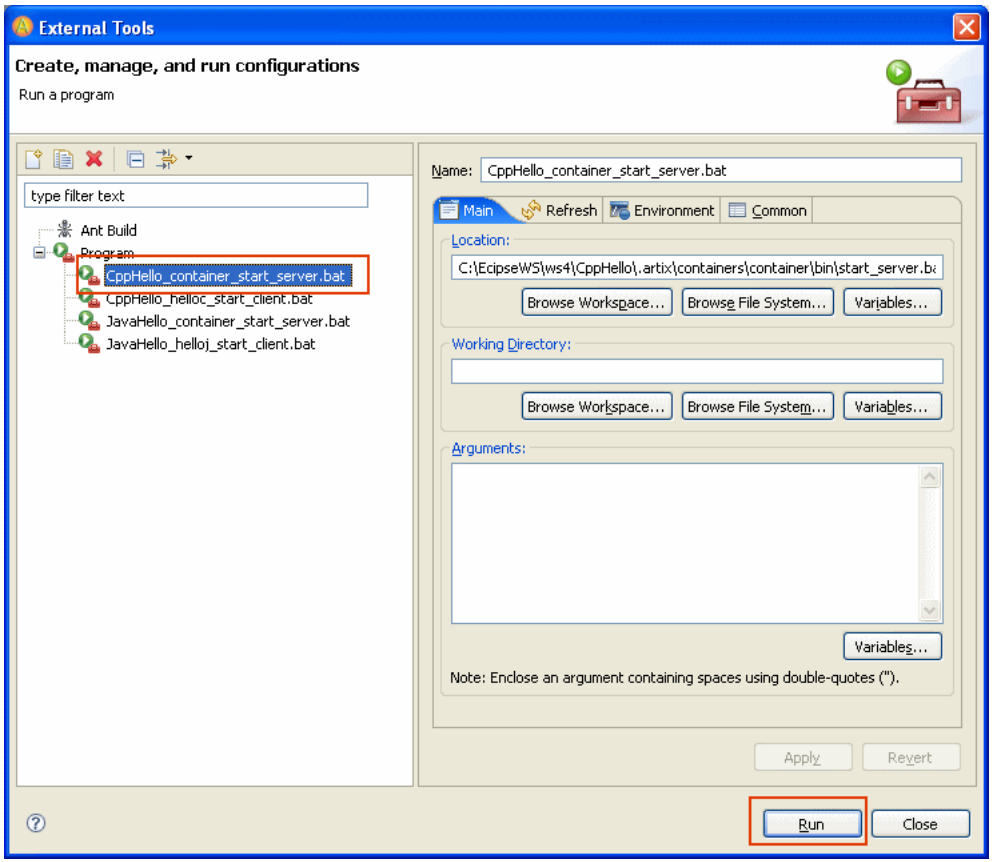
Running the C++ server

To run the C++ server:

1. From the Eclipse menu, select **Run | External Tools | External Tools**.
2. In the **External Tools** window, do the following:
   i. In the list of configurations on the left, expand the **Program** node.
   ii. Select the launch-configuration entry for **CppHello_container_start_server.bat**.

   The **External Tools** window now looks like Figure 40.

Figure 40: *External Tools Window*



iii. Click **Run**.

If you are using Windows XP SP2 with the Windows Firewall enabled, the firewall might display a Security Alert, as shown in Figure 41.

Figure 41: *Windows Security Alert Window*

3. If this window appears, click **Unblock**, which allows the server to run.

   In any case, a **Command Prompt** window opens, displaying messages from the running C++ server.

Running the C++ client

To run the C++ client:

1. Select the launch configuration entry for **CppHello_consumer_instance_start_client.bat**.
2. Click **Run**.

Stopping the C++ server and clearing the console

To stop the C++ server and clear the console, follow the instructions in "Stopping the JAX_WS server and clearing the console" on page 105.

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

Note: If an application takes any arguments, you must edit the start script to include the arguments.

To run a C++ application from a 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, which launches the server in a new command window.
3. In the command prompt, change to the following directory:

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

4. Run the `start_client` script.

   The client launches in a new command window.

   In a few seconds, the command window displays the message, `Operation sayHi received: OutPart = GoodMorning`.
5. Stop the client. To do so, in the client's command window, press **Ctrl+C**.
6. Stop the server. To do so, in the server's command window, press **Ctrl+C**.

Index

W

X