MICRO FOCUS

Artix
Version 5.6.4

Configuring and Deploying Artix Solutions, C++ Runtime

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK

http://www.microfocus.com

2017-02-20

Contents

Preface ..... **vii**
Contacting Micro Focus ix

Part I Configuring Artix

Getting Started **3**
Setting your Artix Environment 3
Artix Environment Variables 5
Customizing your Environment Script 7

Artix Configuration **9**
Artix Configuration Concepts 9
Configuration Data Types 12
Artix Configuration Domain Files 13
Command-Line Configuration 16

Artix Logging **19**
Configuring Logging Filters 19
Configuring Log Stream Plugins 22
Logging for Subsystems and Services 25
Dynamic Artix Logging 33
Configuring Message Snoop 35
Configuring SNMP Logging 37

Enterprise Performance Logging **43**
Enterprise Management Integration 43
Configuring Performance Logging 44
Performance Logging Message Formats 47
Remote Performance Logging 48
Configuring Remote Performance Logging 50

Using Artix with International Codesets **55**
Introduction to International Codesets 55
Working with Codesets using SOAP 57
Working with Codesets using CORBA 58
Working with Codesets using Fixed Length Records 60
Working with Codesets using Message Interceptors 62
Routing with International Codesets 70

Part II Deploying Artix Services

Deploying Services in an Artix Container **75**
Introduction to the Artix Container 75
Generating a Plug-in and Deployment Descriptor 79

Running an Artix Container Server 82
Running an Artix Container Administration Client 85
Deploying Services on Restart 89
Running an Artix Container as a Windows Service 93
Debugging Plug-ins Deployed in a Container 96

Deploying an Artix Transformer 99
The Artix Transformer 99
Standalone Deployment 101
Deployment as Part of a Chain 104
Optional Configuration 106

Deploying a Service Chain 109
The Artix Chain Builder 109
Configuring the Artix Chain Builder 110

Deploying Artix Services for High Availability 115
Introduction 115
Setting up a Persistent Database 117
Configuring Persistent Services for High Availability 118
Configuring Locator High Availability 121
Configuring Client-Side High Availability 123

Deploying WS-Reliable Messaging 129
Introduction 129
Enabling WS-RM 131
Configuring WS-RM Attributes 132
Configuring WS-RM Threading 139
Configuring WS-RM Persistence 140

Part III Accessing Artix Services

Configuring WS-Addressing 145
Introduction 145
Configuring a WS-A Message Exchange Pattern 147

Publishing WSDL Contracts 151
Artix WSDL Publishing Service 151
Configuring the WSDL Publishing Service 152
Querying the WSDL Publishing Service 155

Accessing Contracts and References 159
Introduction 159
Enabling Server and Client Applications 160
Accessing WSDL Contracts 162
Accessing Endpoint References 166
Accessing Artix Services 170

Accessing Services with UDDI 173
Introduction to UDDI 173
Configuring UDDI Proxy 174

Embedding Artix in a BEA Tuxedo Container.....175
Embedding an Artix Process in a Tuxedo Container 175

Index.....177

Preface

What is Covered in this Book

Configuring and Deploying Artix Solutions, C++ Runtime explains how to configure and deploy Artix services in a C++ environment. This book provides detailed descriptions of the specific tasks involved in configuring and launching Artix applications and services.

For details of using Artix in a pure Java environment, see ***Configuring and Deploying Artix Solutions, Java Runtime***. This book applies to systems that use the Artix Java API for XML-Based Web Services (JAX-WS).

This book does not discuss the specifics of the different middleware and messaging products that Artix interacts with. Any discussion about the features of specific middleware products or transports relates to how Artix interacts with these features. It is assumed that you have a working knowledge of the specific middleware products and transports you are using.

Who Should Read this Book

The main audience of ***Configuring and Deploying Artix Solutions, C++ Runtime*** is Artix system administrators. However, anyone involved in designing a large scale Artix solution will find this book useful.

Knowledge of specific middleware or messaging transports is not required to understand the general topics discussed in this book. However, if you are using this book as a guide to deploying runtime systems, you should have a working knowledge of the middleware transports that you intend to use in your Artix solutions.

Note: When deploying Artix in a distributed architecture with other middleware, please see the documentation for that middleware product. You may require access to an administrator. For example, a Tuxedo administrator is required to complete a Tuxedo distributed architecture.

How to Use this Book

Part I, Configuring Artix

This part includes the following:

- Getting Started describes how to set an Artix system environment using the `artix_env` script.
- Artix Configuration describes Artix configuration concepts such as configuration scopes, namespaces, and variables. It also explains how to use configuration files and commands to deploy your applications.

- Artix Logging explains how to configure Artix logging. It also explains Artix support for Java log4j and SNMP (Simple Network Management Protocol).
- Enterprise Performance Logging explains how to configure integration with third-party Enterprise Management Systems (EMS), such as IBM Tivoli and BMC Patrol.
- Using Artix with International Codesets explains how to configure Artix support for internationalization.

Part II, Deploying Artix Services

If you are deploying Artix services, you may want to read one or more of the following:

- Deploying Services in an Artix Container explains how to use the Artix container to deploy and manage Artix Web services.
- Deploying an Artix Transformer explains how to deploy the Artix transformer service.
- Deploying a Service Chain explains how to deploy an Artix service chain.
- Deploying Artix Services for High Availability explains how to deploy Artix high availability (for example, server-side replication and client-side failover).
- Deploying WS-Reliable Messaging explains how to deploy WS-Reliable Messaging in Artix.

Part III, Accessing Artix Services

This part describes several different ways to access Artix services:

- Configuring WS-Addressing explains how to configure WS-Addressing Message Exchange Patterns in Artix.
- Publishing WSDL Contracts explains how to use the Artix WSDL Publishing service to publish WSDL contracts.
- Accessing Contracts and References explains how to use Artix configuration to access Artix WSDL contracts and endpoint references.
- Accessing Services with UDDI explains how to use Universal Description, Discovery and Integration (UDDI) with Artix.
- Embedding Artix in a BEA Tuxedo Container describes how to deploy Artix into a BEA Tuxedo environment.

The Artix Documentation Library

For information on the organization of the Artix library, the document conventions used, and where to find additional resources, see ***Using the Artix Library***, available with the Artix documentation at https://supportline.microfocus.com/productdoc.aspx.

Contacting Micro Focus

Our Web site gives up-to-date details of contact numbers and addresses.

Further Information and Product Support

Additional technical information or advice is available from several sources.

The product support pages contain a considerable amount of additional information, such as:

- The WebSync service, where you can download fixes and documentation updates.
- The Knowledge Base, a large collection of product tips and workarounds.
- Examples and Utilities, including demos and additional product documentation.

To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page.

Note:
Some information may be available only to customers who have maintenance agreements.

If you obtained this product directly from Micro Focus, contact us as described on the Micro Focus Web site, http://www.microfocus.com. If you obtained the product from another source, such as an authorized distributor, contact them for help first. If they are unable to help, contact us.

Information We Need

However you contact us, please try to include the information below, if you have it. The more information you can give, the better Micro Focus SupportLine can help you. But if you don't know all the answers, or you think some are irrelevant to your problem, please give whatever information you have.

- The name and version number of all products that you think might be causing a problem.
- Your computer make and model.
- Your operating system version number and details of any networking software you are using.
- The amount of memory in your computer.
- The relevant page reference or section in the documentation.
- Your serial number. To find out these numbers, look in the subject line and body of your Electronic Product Delivery Notice email that you received from Micro Focus.

Contact information

Our Web site gives up-to-date details of contact numbers and addresses.

Additional technical information or advice is available from several sources.

The product support pages contain considerable additional information, including the WebSync service, where you can download fixes and documentation updates. To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page.

If you are a Micro Focus SupportLine customer, please see your SupportLine Handbook for contact information. You can download it from our Web site or order it in printed form from your sales representative. Support from Micro Focus may be available only to customers who have maintenance agreements.

You may want to check these URLs in particular:

- http://www.microfocus.com/products/corba/artix.aspx (trial software download and Micro Focus Community files)
- https://supportline.microfocus.com/productdoc.aspx. (documentation updates and PDFs)

To subscribe to Micro Focus electronic newsletters, use the online form at:

http://www.microfocus.com/Resources/Newsletters/infocus/newsletter-subscription.asp

Part I

Configuring Artix

In this part

This part contains the following chapters:

Getting Started	page 3
Artix Configuration	page 9
Artix Logging	page 19
Enterprise Performance Logging	page 43
Using Artix with International Codesets	page 55

Getting Started

This chapter explains how to set up an Artix C++ runtime environment.

Setting your Artix Environment

To use the Artix tools and runtime environment, the host computer must have several Artix-specific environment variables set. These variables can be configured during installation, or later using the `artix_env` script, or configured manually. This section shows how to run the `artix_env` script and explains the available options.

Running the artix_env script

The Artix installation process creates a script named `artix_env` on UNIX systems, or `artix_env.bat` on Windows, which captures the information required to set your host's environment variables. Running this script configures your system to use Artix. The script is located in the following directory:

```
ARTIX_PRODUCT_DIR\bin
```

Command-line arguments

On Windows, `artix_env.bat` takes two command parameters:

- `preserve`
- `-verbose`

On UNIX `artix_env` takes the following optional arguments:

- `preserve`
- `-verbose`
- `-bits`
- `-compiler`

These options have the following effects:

Table 1: *Options to artix_env Script*

Option	Description
preserve	Preserves the settings of any environment variables that have already been set. By default this option is off. When it is set to on, artix_env does not overwrite the values of variables that are already set. This option applies to the following environment variables: • IT_PRODUCT_DIR • IT_LICENSE_FILE • IT_CONFIG_DIR • IT_CONFIG_DOMAINS_DIR • IT_DOMAIN_NAME • IT_ART_ADMIN_PATH • IT_IDL_CONFIG_FILE • CLASSPATH • PATH • LIBPATH (AIX) • LD_LIBRARY_PATH (Solaris, Linux) • LD_PRELOAD (Linux) • SHLIB_PATH (HP-UX) For more detailed information, see "Artix Environment Variables" on page 5. **Note:** Before using the -preserve option, always ensure that the existing environment variable values are set correctly.
-verbose	artix_env outputs verbose messages to stdout. By default this option is off.
-bits [32|64]	Sets the Artix environment for the specified C++ compiler width. The default is 32.
-compiler	Specifies the C++ compiler to use; for example, acca0331cios).

Artix Environment Variables

This section describes the following environment variables in more detail:

- JAVA_HOME
- IT_PRODUCT_DIR
- IT_LICENSE_FILE
- IT_CONFIG_DIR
- IT_CONFIG_DOMAINS_DIR
- IT_DOMAIN_NAME
- IT_IDL_CONFIG_FILE
- IT_WSDLGEN_CONFIG_FILE
- IT_ART_ADMIN_PATH
- PATH

Note: You do not have to manually set your environment variables; you can do so by running the provided `artix_env` script.

The environment variables are explained in Table 2:

Table 2: *Artix Environment Variables*

Variable	Description
JAVA_HOME	The directory path to your system's JDK is specified with the system environment variable JAVA_HOME. You may wish to specify the JVM bundled with Artix, which is installed in IT_PRODUCT_DIR\jre. Alternatively you can specify a previously installed JVM using the Artix installer. If not specified, this defaults to the first JRE found on the system.
IT_PRODUCT_DIR	IT_PRODUCT_DIR points to the top level of your product installation. For example, on Windows, if you install Artix into the C:\Artix directory, IT_PRODUCT_DIR should be set to that directory. **Note:** If you have any other products installed that use this variable, and you choose not to install them into the same directory tree, you must reset IT_PRODUCT_DIR each time you switch products. You can override this variable using the -BUSproduct_dir command-line parameter when running Artix applications.
IT_LICENSE_FILE	IT_LICENSE_FILE specifies the location of your Artix license file. The default value is IT_PRODUCT_DIR\etc\licenses.txt. You can override this variable using the -BUSlicense_file command-line parameter when running Artix applications.

Table 2: *Artix Environment Variables*

Variable	Description
IT_CONFIG_DIR	IT_CONFIG_DIR specifies the root configuration directory. The default root configuration directory on UNIX is /etc/opt/iona, and IT_PRODUCT_DIR\etc on Windows. You can override this variable using the -BUSconfig_dir command-line parameter when running Artix applications.
IT_CONFIG_DOMAINS_DIR	IT_CONFIG_DOMAINS_DIR specifies the directory where Artix searches for its configuration files. The configuration domain's directory defaults to IT_CONFIG_DIR\domains. You can override it using the -BUSconfig_domains_dir command-line parameter when running Artix applications.
IT_DOMAIN_NAME	IT_DOMAIN_NAME specifies the name of the configuration domain used by Artix to locate its C++ configuration. This variable also specifies the name of the file in which the configuration is stored. For example, the artix domain is stored in IT_CONFIG_DIR\domains\artix.cfg. You can override this variable with the -BUSdomain_name command-line parameter when running Artix applications.
IT_IDL_CONFIG_FILE	IT_IDL_CONFIG_FILE specifies the configuration used by the Artix IDL compiler. If this variable is not set, you will be unable to run the IDL to WSDL tools provided with Artix. This variable is required for an Artix development installation.The default location is: IT_PRODUCT_DIR\etc\idl.cfg **Note:** Do not modify the default IDL configuration file.
IT_WSDLGEN_CONFIG_FILE	IT_WSDLGEN_CONFIG_FILE specifies the location of the WSDLGen configuration file. WSDLGen is a tool used to generate C++ code from WSDL. The default location of the WSDLGen configuration file is: IT_PRODUCT_DIR\tools\etc\wsdlgen.cfg This file is used to specify the location of templates used for C++ code generation.
IT_ART_ADMIN_PATH	IT_ART_ADMIN_PATH specifies the location of an internal configuration script used by administration tools. Defaults to IT_CONFIG_DIR\admin.

Table 2: *Artix Environment Variables*

Variable	Description
PATH	The Artix bin directories are prepended on the PATH to ensure that the proper libraries, configuration files, and utility programs (for example, the IDL compiler) are used. These settings avoid problems that might otherwise occur if Orbix and/or Tuxedo (both include IDL compilers and CORBA class libraries) are installed on the same host computer. The default Artix bin directory is: **UNIX** $IT_PRODUCT_DIR/bin **Windows** %IT_PRODUCT_DIR%\bin

Customizing your Environment Script

The artix_env script sets the Artix environment variables using values obtained from the Artix installer and from the script's command-line options. The script checks each one of these settings in sequence, and updates them, where appropriate.

The artix_env script is designed to suit most needs. However, if you want to customize it for your own purposes, please note the following points in this section.

Before you begin

You can only run the artix_env script once in any console session. If you run this script a second time, it exits without completing. This prevents your environment from becoming bloated with duplicate information (for example, on your PATH and CLASSPATH).

In addition, if you introduce any errors when customizing the artix_env script, it also exits without completing. This feature is controlled by a variable called IT_ARTIXENV on Windows or IT_ARTIX_ENV_SET on UNIX.The variable is set to true the first time you run the script in a console; this causes the script to exit when run again.

Environment variables

The following applies to the environment variables set by the artix_env script:

- The JAVA_HOME environment variable defaults to the value obtained from the Artix installer. If you do not manually set this variable before running artix_env, it takes its value from the installer. The default location for the JRE supplied with Artix is ARTIX_PRODUCT_DIR\jre.

- The following environment variables are all set with default values relative to `IT_PRODUCT_DIR`:
  - `JAVA_HOME`
  - `IT_CONFIG_FILE`
  - `IT_IDL_CONFIG_FILE`
  - `IT_CONFIG_DIR`
  - `IT_CONFIG_DOMAINS_DIR`
  - `IT_LICENSE_FILE`
  - `IT_ART_ADMIN_PATH`

  If you do not set these variables manually, `artix_env` sets them with default values based on `IT_PRODUCT_DIR`. For example, the default for `IT_CONFIG_DIR` on Windows is `IT_PRODUCT_DIR\etc`.
- The `IT_IDL_CONFIG_FILE` environment variable is a required only for an Artix Development installation. All other environment variables are required for both Development and Runtime installations.
- Before `artix_env` sets each environment variable, it checks if the `preserve` command-line option was supplied when the script was run. This ensures that your preset values are not overwritten. Before using the `preserve` option, always check the existing values for these variables are set correctly.

Artix Configuration

This chapter introduces the main concepts and components in the Artix C++ runtime configuration (for example, configuration domains, scopes, variables, and data types). It also explains how to use Artix configuration files and commands to manage your applications.

Artix Configuration Concepts

The Artix C++ runtime is built upon the Adaptive Runtime architecture (ART).

Runtime behavior is established through common and application-specific configuration settings that are applied during application startup. As a result, the same application code can be run, and can exhibit different capabilities, in different configuration environments. This section includes the following:

- Configuration domains.
- Configuration scopes.
- Specifying configuration scopes.
- Configuration namespaces.
- Configuration variables.

Configuration domains

An Artix *configuration domain* is a collection of configuration information in an Artix C++ runtime environment. This information consists of configuration variables and their values. A default Artix configuration is provided when Artix is installed. The default Artix configuration domain file has the following location:

Windows	`%IT_PRODUCT_DIR%\etc\domains\artix.cfg`
UNIX	`$IT_PRODUCT_DIR/etc/domains/artix.cfg`

The contents of this file can be modified to affect aspects of Artix behavior (for example, logging or routing).

Configuration scopes

An Artix configuration domain is subdivided into *configuration scopes*. These are typically organized into a hierarchy of scopes, whose fully-qualified names map directly to bus names. By organizing configuration variables into various scopes, you can provide different settings for individual services, or common settings for groups of services.

Local configuration scopes

Configuration scopes apply to a subset of services or to a specific service in an environment. For example, the Artix `demo` configuration scope includes example local configuration scopes for demo applications.

Application-specific configuration variables either override default values assigned to common configuration variables, or establish new configuration variables. Configuration scopes are localized through a name tag and delimited by a set of curly braces terminated with a semicolon, for example, *scopeNameTag* {…};

A configuration scope may include nested configuration scopes. Configuration variables set within nested configuration scopes take precedence over values set in enclosing configuration scopes.

In the `artix.cfg` file, there are several predefined configuration scopes. For example, the `demo` configuration scope includes nested configuration scopes for some of the demo programs included with the product.

Example 1: *Demo Configuration Scope*

```
demo
{
 fml_plugin
 {
     orb_plugins = ["local_log_stream", "iiop_profile",
            "giop", "iiop", "soap", "http", "G2", "tunnel",
            "mq", "ws_orb", "fml"];
 };
 telco
 {
    orb_plugins = ["local_log_stream", "iiop_profile",
                  "giop", "iiop", "G2", "tunnel"];
    plugins:tunnel:iiop:port = "55002";
    poa:MyTunnel:direct_persistent = "true";
    poa:MyTunnel:well_known_address = "plugins:tunnel";

    server
      {
        orb_plugins = ["local_log_stream", "iiop_profile",
                    "giop", "iiop", "ots", "soap", "http", "G2:,
                     "tunnel"];
        plugins:tunnel:poa_name = "MyTunnel";
     };
 };
 tuxedo
 {
     orb_plugins = ["local_log_stream", "iiop_profile",
                    "giop", "iiop", "soap", "http", "tuxedo"];

     event_log:filters = ["*=FATAL+ERROR"];
 };
};
```

Note: The `orb_plugins` list is redefined within each configuration scope.

Specifying configuration scopes

To make an Artix process run under a particular configuration scope, you specify that scope using the `-BUSname` parameter. Configuration scope names are specified using the following format

`scope.subscope`

For example, the scope for the `telco` server demo shown in Example 1 is specified as `demo.telco.server`. During process initialization, Artix searches for a configuration scope with the same name as the `-BUSname` parameter.

There are two ways of supplying the `-BUSname` parameter to an Artix process:

- Pass the argument on the command line.
- Specify the `-BUSname` as the third parameter to `IT_Bus::init()`.

For example, to start an Artix process using the configuration specified in the `demo.tuxedo` scope, you can start the process using the following syntax:

```
processName [application parameters] -BUSname demo.tuxedo
```

Alternately, you can use the following code to initialize the Artix bus:

```
IT_Bus::init (argc, argv, “demo.tuxedo”);
```

If a corresponding scope is not located, the process starts under the highest level scope that matches the specified scope name. If there are no scopes that correspond to the `-BUSname` parameter, the Artix process runs under the default global scope. For example, if the nested `tuxedo` scope does not exist, the Artix process uses the configuration specified in the `demo` scope; if the `demo` scope does not exist, the process runs under the default global scope.

Configuration namespaces

Most configuration variables are organized within namespaces, which group related variables. Namespaces can be nested, and are delimited by colons (`:`). For example, configuration variables that control the behavior of a plug-in begin with `plugins:` followed by the name of the plug-in for which the variable is being set. For example, to specify the port on which the Artix standalone service starts, set the following variable:

```
plugins:artix_service:iiop:port
```

To set the location of the routing plug-in’s contract, set the following variable:

```
plugins:routing:wsdl_url
```

Configuration variables

Configuration data is stored in variables that are defined within each namespace. In some instances, variables in different namespaces share the same variable names.

Variables can also be reset several times within successive layers of a configuration scope. Configuration variables set in narrower configuration scopes override variable settings in wider scopes. For example, a `company.operations.orb_plugins` variable would override a `company.orb_plugins` variable. Plug-ins specified at the `company` scope would apply to all processes in that scope, except those processes that belong specifically to the `company.operations` scope and its child scopes.

Further information

For detailed information on Artix configuration namespaces and variables, see the Artix Configuration Reference.

Configuration Data Types

Each Artix configuration variable has an associated data type that determines the variable's value.

Data types can be categorized as follows:

- Primitive types
- Constructed types

Primitive types

Artix supports the following three primitive types:

- `boolean`
- `double`
- `long`

Constructed types

Artix supports two constructed types: `string` and `ConfigList` (a sequence of strings).

- In an Artix configuration domain file (`.cfg`), the `string` character set is ASCII.
- The `ConfigList` type is simply a sequence of `string` types. For example:

```
orb_plugins = ["local_log_stream", "iiop_profile",
   "giop","iiop"];
```

Artix Configuration Domain Files

This section explains how to use Artix configuration domain files to manage applications in your environment. These files use the `.cfg` extension. This section includes the following:

- "Default configuration file".
- "Importing configuration settings".
- "Working with multiple installations".
- "Using symbols as configuration file parameters".

Default configuration file

The Artix configuration domain file contains all the configuration settings for the domain. The default configuration domain file is found in the following location:

Windows `%IT_PRODUCT_DIR%\etc\domains\artix.cfg`

UNIX `$IT_PRODUCT_DIR/etc/domains/artix.cfg`

You can edit the settings in an Artix configuration domain file to modify different aspects of Artix behavior (for example, routing, or levels of logging).

Importing configuration settings

You can manually create new Artix configuration domain files to compartmentalize your applications. These new configuration domain files can import information from other configuration domains using an `include` statement in your configuration file.

This provides a convenient way of compartmentalizing your application-specific configuration from the global ART configuration information that is contained in the default configuration domain file. It also means that you can easily revert to the default settings in the default Artix configuration domain file. Using separate application-specific configuration files is the recommended way of working with Artix configuration.

Example 2 shows an `include` statement that imports the default configuration file. The include statement is typically the first line the configuration file.

Example 2: *Configuration file include statement*

```
include "../../../../../etc/domains/artix.cfg";

my_app_config {
...
}
```

For complete working examples of Artix applications that use this import mechanism, see the configuration files provided with Artix demos. These demo applications are available from the following directory:

ArtixInstallDir`\samples`

Working with multiple installations

If you are using multiple installations or versions of Artix, you can use your configuration files to help manage your applications as follows:

1. Install each version of Artix into a different directory.
2. Install your applications into their own directory.
3. Copy the `artix.cfg` file from whichever Artix release you want to use into another directory (for example, an application directory).
4. In your application's local configuration file, include the `artix.cfg` file from your copy location.

This enables you to switch between Artix versions by copying the corresponding `artix.cfg` file into a common location. This avoids having to update the directory information in your configuration file whenever you want to switch between Artix versions.

Using symbols as configuration file parameters

You can define arbitrary symbols for use in Artix `.cfg` files, for example:

```
SERVER_LOG = "my_server_log";
```

These symbols can then be reused as parameters in configuration settings, for example:

```
plugins:local_log_stream:filename = SERVER_LOG;
```

You can use configuration symbols to customize your file depending on the environment. This enables you to use the same basic configuration file in different environments (for example, development, test, and production).

Using configuration symbols in a string

You can use symbols within a string using a syntax of %{*SYMBOL_NAME*}. For example, if you define the following symbol:

```
LOG_LEVEL = "FATAL+ERROR+WARNING+INFO_MED+INFO_HI";
```

This can be used within a string as follows:

```
event_log:filters = ["*=%{LOG_LEVEL}"];
```

You can also combine multiple symbols within a string as follows:

```
plugins:local_log_stream:filename =
   "%{APP_NAME}-%{CLIENT_LOG}";
```

Configuration example

The configuration file in Example 3 contains some user-defined symbols:

Example 3: *Defining Configuration Symbols*

```
#mydomain.cfg

INSTALL_CFG = "../../artix.cfg";

CLIENT_LOG = "my_client.log";
SERVER_LOG = "my_server.log";
APP_NAME = "myapp";
LOG_LEVEL = "FATAL+ERROR+WARNING+INFO_MED+INFO_HI";

include "template.cfg";
```

The configuration file in Example 4 uses the predefined symbols in configuration variable settings:

Example 4: *Using Configuration Symbols*

```
#template.cfg

include INSTALL_CFG

myapps {
  orb_plugins = ["local_log_stream", "soap", "http"];

   server {
     #Simple user-defined symbol.
     plugins:local_log_stream:filename = SERVER_LOG;

     #Using a symbol within a string.
     event_log:filters = ["*=%{LOG_LEVEL}"];
  }

  client {
     #Combining symbols within a string.
     plugins:local_log_stream:filename =
   "%{APP_NAME}-%{CLIENT_LOG}";
  };
};
```

This example shows a user-defined symbol in an `include` statement. It shows a simple example of using a symbol in an configuration setting, and more complex examples of using symbols in strings.

For details of using configuration symbols on the command line, see “Command-Line Configuration” on page 16.

Command-Line Configuration

This section explains how to configure the following options on the command line:

- Configuration variables
- Configuration scopes
- User-defined configuration symbols
- Environment variables
- Location of WSDL and references
- Multiple bus instances

Setting configuration variables

Artix enables you to override configuration variables at runtime by using arguments on the command line. These arguments are then passed to the Artix `IT_Bus::init()` call. Setting configuration variables on the command line takes precedence over variables in a configuration file.

Command-line arguments for configuration variables take the following format:

```
-BUSCONFIG_VariableName Value
```

For example:

```
client -BUSCONFIG_plugins:local_log_stream:filename
   client.log  -BUSCONFIG_orb_plugins
   ["local_log_stream","soap","http"]
   -BUSCONFIG_event_log:filters ["*=*"]
```

For detailed information on Artix configuration variable settings, see the ***Artix Configuration Reference***.

Setting configuration scopes

You can specify configuration scopes when starting an application on the command line using the `-BUSname` argument.

For example, to start a process using the configuration specified in the `demo.myapp` scope, you would start the process with the following syntax:

```
ProcessName [application parameters] -BUSname demo.myapp
```

For more details, see “Specifying configuration scopes” on page 11.

Setting configuration symbols

You can also override user-defined configuration symbols on the command line. Setting configuration symbols on the command line takes precedence over symbols in a configuration file.

For example, you can override the log file name in Example 3 on page 15 using command-line arguments as follows:

```
client -BUSCONFIG_CLIENT_LOG test2.log
```

This successfully creates a log file named `test2.log`. For more details, see “Using symbols as configuration file parameters” on page 14.

Setting environment variables

You can use command-line arguments to pass the value of environment variables to configuration files.

For example, you can specify the directory where Artix searches for its configuration files using the `-BUSconfig_domains_dir` argument. For more details on Artix environment variables, see “Getting Started”.

Specifying locations of WSDL and references

You can specify the location of WSDL contracts and Artix references using the following command-line arguments:

```
-BUSservice_contract URL
-BUSservice_contract_dir Directory
-BUSinitial_reference url
```

For example:

```
./server -BUSservice_contract ../../etc/hello.wsdl
```

For more details, see “Accessing Contracts and References”.

Artix Logging

This chapter describes how to configure Artix logging. It shows how to configure logging for specific Artix subsystems and services, how to control dynamic logging on the command line and Artix message snoop. It also explains the Artix support for Java log4j and the Simple Network Management Protocol.

Configuring Logging Filters

Logging in Artix is controlled by the `event_log:filters` configuration variable, and by the log stream plug-ins (for example, `local_log_stream` and `xmlfile_log_stream`). This section explains the following:

- "Configuring logging levels"
- "Logging severity levels"
- "Filtering passwords from logs"

Configuring logging levels

You can set the `event_log:filters` configuration variable to provide a wide range of logging levels. The `event_log:filters` variable can be set in your Artix configuration domain file: *`ArtixInstallDir`*`\etc\domains\artix.cfg`.

Displaying errors

The default `event_log:filters` setting displays errors only:

```
event_log:filters = ["*=FATAL+ERROR"];
```

Displaying warnings

The following setting displays errors and warnings only:

```
event_log:filters = ["*=FATAL+ERROR+WARNING"];
```

Displaying request/reply messages

Adding `INFO_MED` causes all request/reply messages to be logged (for all transport buffers):

```
event_log:filters = ["*=FATAL+ERROR+WARNING+INFO_MED"];
```

Displaying trace output

The following setting displays typical trace statement output (without the raw transport buffers):

```
event_log:filters = ["*=FATAL+ERROR+WARNING+INFO_HI"];
```

Displaying all logging

The following setting displays all logging:

```
event_log:filters = ["*=*"];
```

The default configuration settings enable logging of only serious errors and warnings. For more exhaustive information, select a different filter list at the default scope, or include a more expansive `event_log:filters` setting in your configuration scope.

Logging severity levels

Artix supports the following levels of log message severity:

- Information
- Warning
- Error
- Fatal error

Information

Information messages report significant non-error events. These include server startup or shutdown, object creation or deletion, and details of administrative actions.

Information messages provide a history of events that can be valuable in diagnosing problems. Information messages can be set to low, medium, or high verbosity.

Warning

Warning messages are generated when Artix encounters an anomalous condition, but can ignore it and continue functioning. For example, encountering an invalid parameter, and ignoring it in favor of a default value.

Error

Error messages are generated when Artix encounters an error. Artix might be able to recover from the error, but might be forced to abandon the current task. For example, an error message might be generated if there is insufficient memory to carry out a request.

Fatal error

Fatal error messages are generated when Artix encounters an error from which it cannot recover. For example, a fatal error message is generated if Artix cannot find its configuration file.

Table 3 shows the syntax used by the `event_log:filters` variable to specify Artix logging severity levels.

Table 3: *Artix Logging Severity Levels*

Severity Level	Description
`INFO_LO[W]`	Low verbosity informational messages.
`INFO_MED[IUM]`	Medium verbosity informational messages.
`INFO_HI[GH]`	High verbosity informational messages.
`INFO[_ALL]`	All informational messages.
`WARN[ING]`	Warning messages.

Table 3: *Artix Logging Severity Levels*

Severity Level	Description
ERR[OR]	Error messages.
FATAL[_ERROR]	Fatal error messages.
*	All messages.

Filtering passwords from logs

You can also use event log filters to control whether plain-text passwords are printed in the log.

To enable filtering of Web Services Security (WS-S) plain-text passwords, specify the following configuration setting:

```
event_log:filter_sensitive_info =
   ["event_log:filter_sensitive_info:wss_password"];
event_log:filter_sensitive_info:wss_password =
   ["#PasswordText$%''$%>", "</", "*"];
```

This setting changes the characters in the log of a WS-S plain-text password to * characters.

The `event_log:filter_sensitive_info` configuration variable can also be used to filter other types of sensitive logging information, and multiple filters can be enabled in a single setting. The general format for this configuration variable is as follows:

```
event_log:filter_sensitive_info = ["foo"];
foo = [ "Start", "End", "#"];
```

In this general example, the first line provides the list of pattern names to consider for replacement, and the second line provides the actual pattern using the following syntax:

```
["Start_Pattern", "End_Pattern", "Replacement_Character"];
```

This replaces anything in the log between `Start_pattern` and `End_pattern` with the `#` character.

Because Artix configuration files do not support the escaped " character in configuration, any pattern that has the " character should instead replace this character with the following:

```
$%''$%
```

You must specify two single quotes and not a double quote. These are then treated as the " character during the filtering of logging information.

Configuring Log Stream Plugins

In addition to setting the event log filter, you must ensure that a log stream plug-in is set in your `artix.cfg` file. These include the `local_log_stream` and the `xmlfile_log_stream`. This section explains how to use log stream plugins to perform the following tasks:

- "Configuring logging output"
- "Using a rolling log file"
- "Buffering the output stream"
- "Configuring HTTP trace logging"
- "Configuring precision logging"
- "Logging the thread ID"

Configuring logging output

The `local_log_stream` sends logging to a text file; while the `xmlfile_log_stream` outputs logging to an XML file. The `local_log_stream` is set by default.

Using text log files

To configure the `local_log_stream`, set the following variables in your configuration file:

```
//Ensure these plug-ins exist in your orb_plugins list
orb_plugins = ["local_log_stream", ... ];

//Optional text filename
plugins:local_log_stream:filename = "/var/mylocal.log";
```

If you do not specify a text log file name, logging is sent to `stderr`.

Using XML log files

To configure the `xmlfile_log_stream`, set the following variables in your configuration file:

```
//Ensure this plug-in is in your orb_plugins list
orb_plugins = ["xmlfile_log_stream", ... ];

// Optional filename; can be qualified.
plugins:xmlfile_log_stream:filename =
   "artix_logfile.xml";

// Optional process ID added to filename (default is
   false).
plugins:xmlfile_log_stream:use_pid = "false";
```

You must ensure that your application can detect the configuration settings for the log stream plugins. You can either set them at the global scope, or configure a unique scope for use by your application, for example:

```
IT_Bus::init(argc, argv, "demo.myscope");
```

This enables you to place the necessary configuration in the `demo.myscope` scope.

Using a rolling log file

By default, a logging plug-in creates a new log file each day to prevent the log file from growing indefinitely. In this model, the log stream adds the current date to the configured filename. This produces a complete filename, for example:

```
/var/adm/my_artix_log.01312006
```

A new log file begins with the first event of the day, and ends each day at `23:59:59`.

Specifying the date format

You can configure the format of the date in the rolling log file, using the following configuration variables:

- `plugins:local_log_stream:filename_date_format`
- `plugins:xmlfile_log_stream:filename_date_format`

The specified date must conform to the format rules of the ANSI C `strftime()` function. For example, for a text log file, use the following settings:

```
plugins:local_log_stream:rolling_file="true";
plugins:local_log_stream:filename="my_log";
plugins:local_log_stream:filename_date_format="_%Y_%m_%d"
   ;
```

On the 31st January 2006, this results in a log file named `my_log_2006_01_31`.

The equivalent settings for an XML log file are:

```
plugins:xmlfile_log_stream:rolling_file="true";
plugins:xmlfile_log_stream:filename="my_log";
plugins:xmlfile_log_stream:filename_date_format="_%Y_%m_%
   d";
```

Disabling rolling log files

To disable rolling file behavior for a text log file, set the following variable to `false`:

```
plugins:local_log_stream:rolling_file = "false";
```

To disable rolling file behavior for an XML log file, set the following variable to `false`:

```
plugins:xmlfile_log_stream:rolling_file = "false";
```

Buffering the output stream

You can also set the output stream to a buffer before it writes to a local log file. To specify this behavior, use either of the following variables:

`plugins:local_log_stream:buffer_file`
`plugins:xmlfile_log_stream:buffer_file`

When set to `true`, by default, the buffer is output to a file every 1000 milliseconds when there are more than 100 messages logged. This log interval and number of log elements can also be configured.

Note: To ensure that the log buffer is sent to the log file, you must always shutdown your applications correctly.

For example, the following configuration writes the log output to a log file every 400 milliseconds if there are more than 20 log messages in the buffer.

Using text log files

```
plugins:local_log_stream:filename = "/var/adm/artix.log";
plugins:local_log_stream:buffer_file = "true";
plugins:local_log_stream:milliseconds_to_log = "400";
plugins:local_log_stream:log_elements = "20";
```

Using XML log files

```
plugins:xml_log_stream:filename = "/var/adm/artix.xml";
plugins:xml_log_stream:buffer_file = "true";
plugins:xml_log_stream:milliseconds_to_log = "400";
plugins:xml_log_stream:log_elements = "20";
```

Configuring HTTP trace logging

HTTP trace logging shows the full HTTP buffers (headers and body) as they go to and from the wire. This feature is disabled by default. You can enable HTTP-specific trace logging using the following setting:

```
policies:http:trace_requests:enabled="true";
```

You should also set log filtering as follows to pick up the HTTP additional messages, and then resend the logs:

```
event_log:filters = ["IT_HTTP=*"];
```

For example, you could enable HTTP trace logging to verify that basic authentication headers are written to the wire correctly.

Similarly, to enable HTTPS-specific trace logging, use the following setting:

```
policies:https:trace_requests:enabled="true";
```

Configuring precision logging

You can also specify whether events are logged with time precision in nanoseconds, or at the granularity of seconds. By default, precision logging is disabled, and Artix logs in seconds. To enable precision logging, use the following setting:

```
plugins:local_log_stream:precision_logging = "true";
```

Logging the thread ID

You can also specify whether a thread ID is logged in the log message, for example:

```
plugins:local_log_stream:log_thread_id = "true";
```

The default is `false`. When this setting has been enabled, the following example logging message shows the thread ID in bold:

```
Wed, 26 Sep 2007 12:22:26.0000000 [homer600:6870:1269287216]
(IT_BUS.CORE:0) I - Registering Bus plugin
   SOAPServicePluginFactory
```

Logging for Subsystems and Services

You can use the `event_log:filters` configuration variable to set fine-grained logging for specified Artix logging subsystems. For example, you can set logging for the Artix core, specific transports, bindings, or services. You can set logging for Artix services, such as the locator, and for services that you have developed.

This section lists the Artix-specific logging subsystems and those for the underlying Adaptive Runtime (ART) core, and shows examples of how to use them.

Artix logging subsystems

Artix logging subsystems are organized into a hierarchical tree, with the `IT_BUS` subsystem at the root. Example logging subsystems include:

```
IT_BUS.CORE
IT_BUS.TRANSPORT.HTTP
IT_BUS.BINDING.SOAP
```

Table 4 shows a list of available Artix logging subsystems.

Table 4: *Artix Logging Subsystems*

Subsystem	Description
`IT_BUS`	Artix bus
`IT_BUS.BINDING`	All bindings

Table 4: *Artix Logging Subsystems*

Subsystem	Description
IT_BUS.BINDING.COLOC	Collocated binding
IT_BUS.BINDING.CORBA	CORBA binding
IT_BUS.BINDING.CORBA.CONTEXT	CORBA context
IT_BUS.BINDING.FIXED	Fixed binding
IT_BUS.BINDING.HTTP	HTTP binding
IT_BUS.BINDING.SOAP	SOAP binding
IT_BUS.BINDING.SOAP12	SOAP 1.2 binding
IT_BUS.BINDING.SOAP_COMMON	Common SOAP binding
IT_BUS.BINDING.TAGGED	Tagged binding
IT_BUS.CORE	Artix core
IT_BUS.CORE.CONFIG	Artix core configuration
IT_BUS.CORE.CONTEXT	Artix core contexts
IT_BUS.CORE.INITIAL_REFERENCE	Artix initial references
IT_BUS.CORE.PLUGIN	Artix plug-ins
IT_BUS.CORE.RESOURCE_RESOLVER	Artix resource resolver
IT_BUS.FOUNDATION.AFC	Artix Foundation Classes (Artix-specific data type extensions)
IT_BUS.FOUNDATION.CONTEXT_LIBRARY	Artix Foundation context library
IT_BUS.I18N.INTERCEPTOR	Internationalization
IT_BUS.INTEGRATION.AP_NANO_AGENT	AmberPoint SOA management agent
IT_BUS.INTEGRATION.CA_WSDM_OBSERVER	CA Web Services Distributed Management observer
IT_BUS.JNI.GENERIC_PLUGIN	Java generic service
IT_BUS.JNI.JBUS	Java Message Service
IT_BUS.JNI.JBUS.TRANSACTION	JMS transactions
IT_BUS.JNI.JNI_UTIL	Java utilities
IT_BUS.JNI.TRANSACTION	Java transactions
IT_BUS.JVM_MANAGER	JVM manager
IT_BUS.LOGGING	Artix logging
IT_BUS.LOGGING.LOG4J	Log4J logging
IT_BUS.LOGGING.RESPONSE_TIME	Response time logging
IT_BUS.LOGGING.SNMP	Simple Network Management Protocol logging

Table 4: *Artix Logging Subsystems*

Subsystem	Description
IT_BUS.MANAGEMENT	Artix management
IT_BUS.MESSAGING_PORT	Artix messaging port
IT_BUS.SERVICE	All Artix services.
IT_BUS.SERVICE.ACTIVATOR.REGISTRY	Artix service activator registry
IT_BUS.SERVICE.CHAIN	Artix chain service
IT_BUS.SERVICE.CONTAINER	Artix container service
IT_BUS.SERVICE.DB	Artix database wrapper (server-side high availability based on Berkeley DB)
IT_BUS.SERVICE.DB.ENV	Artix database environment
IT_BUS.SERVICE.DB.REPLICA.IMPL	Artix database replication messages
IT_BUS.SERVICE.DB.REPLICA.MGR	Artix database replication manager
IT_BUS.SERVICE.DB.REPLICA.MONITOR	Artix database replication monitor
IT_BUS.SERVICE.DB.REPLICA.SYNC	Artix database synchronization manager
IT_BUS.SERVICE.LOCATOR	Artix locator service
IT_BUS.SERVICE.PEER_MANAGER	Artix peer manager service
IT_BUS.SERVICE.ROUTING	Artix router
IT_BUS.SERVICE.ROUTING.XPATH	XPath routing expressions
IT_BUS.SERVICE.SECURITY	Artix security service
IT_BUS.SERVICE.SECURITY.CERT_VALIDATOR	Security certificate validator
IT_BUS.SERVICE.SECURITY.LOGIN_SERVICE.CLIENT	Security login client
IT_BUS.SERVICE.SECURITY.LOGIN_SERVICE.SERVICE	Security login service
IT_BUS.SERVICE.SECURITY.SECURITY_INTERCEPTOR	Security interceptor
IT_BUS.SERVICE.SECURITY.WSS	SOAP Partial Message Protection
IT_BUS.SERVICE.SESSION_MANAGER	Artix session manager service
IT_BUS.SERVICE.WSDL_PUBLISH	Artix WSDL publishing service
IT_BUS.SERVICE.XSLT	Artix transformer service
IT_BUS.TRANSACTIONS	Transactions
IT_BUS.TRANSACTIONS.OTS	CORBA Object Transaction Service transactions
IT_BUS.TRANSACTIONS.WSAT	Web Services Atomic Transactions
IT_BUS.TRANSACTIONS.XA	XA transactions
IT_BUS.TRANSPORT.HTTP	HTTP transport

Table 4: *Artix Logging Subsystems*

Subsystem	Description
IT_BUS.TRANSPORT.MQ	MQ transport
IT_BUS.TRANSPORT.STUB_TRANSPORT	Artix simple stub transport
IT_BUS.TRANSPORT.TUNNELL	Tunnel transport
IT_BUS.TRANSPORT.TUXEDO	Tuxedo transport
IT_BUS.VERSION	Artix version
IT_BUS.WSRM	Web Services Reliable Messaging
IT_BUS.WSRM_DB	Web Services Reliable Messaging persistence
IT_BUS.XA_SWITCH	XA transactions switch
IT_WSRM	Web Services Reliable Messaging

Note: This list may change in future releases.

ART core logging subsystems

Table 4 shows a list of available logging subsystems for the underlying ART core.

Table 5: *ART Core Logging Subsystems*

Subsystem	Description
IT_ATLI2_IOP	Abstract Transport Layer Interface, version 2 with Inter-ORB Protocol
IT_ATLI2_IP	Abstract Transport Layer Interface 2.0 with Internet Protocol
IT_ATLI2_IP_TUNNEL	Abstract Transport Layer Interface, with Internet Tunnel Protocol
IT_ATLI_TLS	Abstract Transport Layer Interface with Transport Security Layer
IT_COBOL_PLI	Artix Mainframe only
IT_CODESET	Internationalization
IT_CONNECTION_FILTER	Connection filter
IT_CORE	ART core
IT_CSI	Common Secure Interoperability
IT_GSP	CORBA binding security
IT_GenericSecurityToolkit	Baltimore and z/OS SystemSLL toolkit
IT_GIOP	General Inter-ORB Protocol
IT_HTTP	Hypertext Transfer Protocol
IT_HTTPS	HTTP with Secure Socket Layer
IT_IIOP	Internet Inter-ORB Protocol
IT_IIOP_TLS	Internet Inter-ORB Protocol with Transport Layer Security
IT_LICENSING	Licensing
IT_MESSAGING	Messaging
IT_MGMT_LOGGING	Management service
IT_OBJECT_KEY_REPLACER	Object key replacer
IT_OTS	Object Transaction Layer
IT_OTS_LITE	Object Transaction Layer Lite
IT_POA	Portable Object Adaptor

Table 5: *ART Core Logging Subsystems*

Subsystem	Description
IT_POA_LOCATOR	Portable Object Adapter with locator
IT_REQUEST_LOGGER	Request logger
IT_SCHANNEL	Schannel security
IT_SECURITY	Security
IT_TLS	Transport Layer Security
IT_WORKQUEUE	Multi-threading
IT_XA	XA transactions
MESSAGE_SNOOP	Message snooping.

Note: This list may change in future releases.

Subsystem filter syntax

The `event_log:filters` variable takes a list of filters, where each filter sets logging for a specified subsystem using the following format:

```
Subsystem=SeverityLevel[+SeverityLevel]...
```

`Subsystem` is the name of the Artix subsystem that reports the messages; while *`SeverityLevel`* represents the severity levels that are logged by that subsystem. For example, the following filter specifies that only errors and fatal errors for the HTTP transport should be reported:

```
IT_BUS.TRANSPORT.HTTP=ERR+FATAL
```

In a configuration file, `event_log:filters` is set as follows:

```
event_log:filters=["LogFilter"[,"LogFilter"]...]
```

The following entry in a configuration file explicitly sets severity levels for a list of subsystem filters:

```
event_log:filters=["IT_BUS=FATAL+ERROR",

   "IT_BUS.BINDING.CORBA=WARN+FATAL+ERROR"];
```

Setting the Artix bus pre-filter

The Artix bus pre-filter provides filtering of log messages that are sent to the EventLog before they are output to the LogStream. This enables you to minimize the time spent generating log messages that will be ignored. For example:

```
event_log:filters:bus:pre_filter = "WARN+ERROR+FATAL";

event_log:filters = ["IT_BUS=FATAL+ERROR",
   "IT_BUS.BINDING=*"];
```

In this example, only WARNING, ERROR and FATAL priority log messages are sent to the EventLog. This means that no processing time is wasted generating strings for INFO log messages. The EventLog then only sends FATAL and ERROR log messages to the LogStream for the IT_BUS subsystem.

Note: event_log:filters:bus:pre_filter defaults to * (all messages). Setting this variable to WARN+ERROR+FATAL improves performance significantly.

Setting logging for specific subsystems

You can set logging filters for specific Artix subsystems. A subsystem with no configured filter value implicitly inherits the value of its parent. The default value at the root of the tree ensures that each node has an implicit filter value. For example:

```
event_log:filters = ["IT_BUS=FATAL+ERROR",
      "IT_BUS.BINDING.CORBA=WARN+FATAL+ERROR"];
```

This means that all subsystems under IT_BUS have a filter of FATAL+ERROR, except for IT_BUS.BINDING.CORBA which has WARN+FATAL+ERROR.

Setting multiple subsystems with a single filter

Using the IT_BUS subsystem means you can adjust the logging for Artix subsytems with a single filter. For example, you can turn off logging for the tunnel transport (IT_BUS.TRANSPORT.TUNNEL=FATAL) and/or turn up logging for the HTTP transport (IT_BUS.TRANSPORT.HTTP=INFO_LOW+...), as show in the following example:

```
event_log:filters= ["IT_BUS=FATAL+ERROR",
                    "IT_BUS.TRANSPORT.TUNNEL=FATAL",
       "IT_BUS.TRANSPORT.HTTP=INFO_LOW+INFO_HI+WARN"];
```

Configuring service-based logging

You can use Artix service subsystems to log for Artix services, such as the locator, and also for services that you have developed. This can be useful when you are running many services, and need to filter services that are particularly noisy. Using service-based logging involves some performance overheads and extra configuration. This feature is disabled by default.

To enable logging for specific services, perform the following steps:

1. Set the following configuration variables:

```
event_log:log_service_names:active = "true";
event_log:log_service_names:services = ["ServiceName1",
   "ServiceName2"];
```

2. Set the event log filters as appropriate, for example:

```
event_log:filters = ["IT_BUS=FATAL+ERROR",
   "ServiceName1=WARN+ERROR+FATAL", "ServiceName2=ERROR+FATAL",
   "ServiceName2.IT_BUS.BINDING.CORBA=INFO+WARN+ERROR+FATAL"
 ];
```

Service name format

In these examples, the service name must be specified in the following format:

"{*NamespaceURI*}*LocalPart*"

For example:

```
"{http://www.my-company.com/bus/tests}SOAPHTTPService"
```

Setting parameterized configuration

The following example shows setting service-based logging in your application using the `-BUSCONFIG_event_log:filters` parameter:

```
const char* bus_argv[] = {"-BUSname", "my_spp_logging",
                          "-BUSCONFIG_event_log:filters", "{IT_BUS=ERR},
   {{http://www.my-company/my_app}SOAPHTTPService.IT_BUS.BINDING.SOAP=INFO}"
```

Logging per bus

For C++ applications, you can configure logging per bus by specifying your logging configuration in an application-specific scope. However, you must also specify logging per bus in your server code, for example:

- Include the following file:
  ArtixInstallDir/include/it_bus/bus_logger.h
- Pass a valid bus to the `BusLogger` (for example, using `BusLogger` macros, such as `IT_INIT_BUS_LOGGER_MEM`).

For full details on how to specify that logging statements are sent to a particular Artix bus, see ***Developing Advanced Artix Plug-ins in C++***.

Programmatic logging configuration

C++ applications can use a logging API to query, add, or cancel logging filters for subsystems, as well as adding and removing services from per-service logging. For example, you can access a C++ `IT_Bus::Logging::LoggingConfig` class by calling `bus->get_pdk_bus()->get_logging_config()`.

For full details, see ***Developing Artix Applications in C++***.

Dynamic Artix Logging

At runtime, you can use `it_container_admin` commands to dynamically get and set logging levels for specific subsystems and services. This section explains how to use the `it_container_admin -getlogginglevel` and `-setlogginglevel` options.

Getting logging levels

The `-getlogginglevel` option gets the logging level for specified a subsystem or service. This command has the following syntax:

```
-getlogginglevel [-subsystem SubSystem] [-service
   {Namespace}LocalPart]
```

Get logging for a specific subsystem

The following example gets the logging level for the CORBA binding only:

```
it_container_admin -getlogginglevel -subsystem
   IT_BUS.BINDING.CORBA
```

Get logging for multiple subsystems

The following example uses a wildcard to get the logging levels for all subsystems:

```
it_container_admin -getlogginglevel -subsystem *
```

This outputs a list of subsystems that have been explicitly set in a configuration file or by `-setlogginglevel`.

For example, if `IT_BUS.BINDING=LOG_INFO` is output, this means that `IT_BUS.BINDING` is set to `LOG_INFO`, and that no child subsystems of `IT_BUS.BINDING` are explicitly set. In this case, all child subsystems inherit `LOG_INFO` from their parent.

Get logging for a specific service

The following example gets the logging level for a locator service that is running in a container:

```
it_container_admin -getlogginglevel -subsystem
   IT_BUS.BINDING.SOAP -service
   {http://ws.iona.com/locator}LocatorService
```

Setting logging levels

The `-setloggginglevel` option sets the logging level for a specified subsystem. This command has the following syntax:

```
-setloggginglevel -subsystem SubSystem -level Level
   [-propagate] [-service {Namespace}Localpart]
```

The possible logging levels are:

```
LOG_FATAL
LOG_ERROR
LOG_WARN
LOG_INFO_HIGH
LOG_INFO_MED
LOG_INFO_LOW
LOG_SILENT
LOG_INHERIT
```

Set logging for a specific subsystem

The following example sets the logging level for the HTTP transport only:

```
it_container_admin -getloggginglevel -subsystem
   IT_BUS.TRANSPORT.HTTP -level LOG_WARN
```

Set logging for multiple subsystems

You can set logging for multiple subsystems by using the `-propagate` option. The following example sets the logging level for all transports (IIOP, HTTP, and so on):

```
it_container_admin -setloggginglevel -subsystem
   IT_BUS.TRANSPORT -level LOG_WARN -propagate true
```

Override child subsystem levels

You can use the `-propagate` option to override child subsystem levels that have been set previously. For example, take the simple case where `IT_BUS` is set to `LOG_INFO`, and no other subsystems are set. If the `IT_BUS` level is changed, it is automatically propagated to all `IT_BUS` children.

However, take the case where `IT_BUS.CORE` is set to `LOG_WARN`, and `IT_BUS.TRANSPORT` is set to `LOG_INFO_LOW`. Setting `IT_BUS` to `LOG_ERROR` affects `IT_BUS` and all its children, except for `IT_BUS.CORE` and `IT_BUS.TRANSPORT`. In this case, you can use `-propagate true` to override the child subsystem levels set previously. For example:

```
it_container_admin -setloggginglevel -subsystem IT_BUS
   -level LOG_ERROR -propagate true
```

Set logging for services

The following example sets the logging level for the SOAP binding when used with the locator service:

```
it_container_admin -setloggginglevel -subsystem
   IT_BUS.BINDING.SOAP -level LOG_INFO_HIGH -service
   {http://ws.iona.com/locator}LocatorService
```

The -propagate option can also be used when setting logging for service. For example, if you have service-specific logging enabled for IT_BUS.BINDING and IT_BUS.BINDING.SOAP, setting a service-specific log level for IT_BUS.BINDING with -propagate true also sets the service level for IT_BUS.BINDING.SOAP.

```
it_container_admin -setlogginglevel -subsystem
   IT_BUS.BINDING -level LOG_INFO_LOW -propagate true
   -service {http://ws.iona.com/locator}LocatorService
```

Inheriting a logging level

You can use the LOG_INHERIT level to cancel the current logging level and inherit from the parent subsystem instead.

For example, if the IT_BUS.CORE subsystem is set to LOG_INFO_LOW, and its parent (IT_BUS) is set to LOG_ERROR, setting IT_BUS.CORE to LOG_INHERIT results in IT_BUS.CORE logging at LOG_ERROR. This is shown in the following example:

```
it_container_admin -setlogginglevel -subsystem
   IT_BUS.CORE -level LOG_INHERIT
```

By default, all subsystems are effectively in LOG_INHERIT mode because they inherit a level from their parent subsystem.

Silent logging

You can use the LOG_SILENT level to specify that a given subsystem does not perform any logging, for example:

```
it_container_admin -setlogginglevel -subsystem
   IT_BUS.TRANSPORT.TUNNEL -level LOG_SILENT
```

Further information

For more details on using the it_container_admin command, see "Deploying Services in an Artix Container" on page 75.

For more details on subsystems, see "Logging for Subsystems and Services" on page 25.

Configuring Message Snoop

Message snoop is an ART-based message interceptor that sends input/output messages to the Artix log to enable viewing of the message content. This is a useful debugging tool when developing and testing an Artix system.

Message snoop is enabled by default. It is automatically added as the last interceptor before the binding to detect any changes that other interceptors might make to the message. By default, message_snoop logs at INFO_MED in the MESSAGE_SNOOP subsystem. You can change these settings in configuration.

Disabling message snoop

Message snoop is invoked on every message call, twice in the client and twice in the server (assuming Artix is on both sides). This means that it can have an impact on performance. More importantly, message snoop involves risks to confidentiality. You can disable message snoop using the following setting:

```
artix:interceptors:message_snoop:enabled = "false";
```

WARNING: For security reasons, it is strongly recommended that message snoop is disabled in production deployments.

Setting a message snoop log level

You can set a message snoop log level globally or for a service port. The following example sets the level globally:

```
artix:interceptors:message_snoop:log_level = "WARNING";
event_log:filters = ["*=WARNING",
   "IT_BUS=INFO_HI+WARN+ERROR", "MESSAGE_SNOOP=WARNING"];
```

The following example sets the level for a service port:

```
artix:interceptors:message_snoop:http://www.acme.com/test
   s:myService:myPort:log_level = "INFO_MED";
event_log:filters = ["*=INFO_MED", "IT_BUS=",
   "MESSAGE_SNOOP=INFO_MED"];
```

Setting a message snoop subsystem

You can set message snoop to a specific subsystem globally or for a service port. The following example sets the subsystem globally:

```
artix:interceptors:message_snoop:log_subsystem =
   "MY_SUBSYSTEM";
event_log:filters = ["*=INFO_MED", "IT_BUS=",
   "MY_SUBSYSTEM=INFO_MED"];
```

The following example sets the subsystem for a service port:

```
artix:interceptors:message_snoop:http://www.acme.com/test
   s:myService:myPort:log_subsystem = "MESSAGE_SNOOP";
event_log:filters = ["*=INFO_MED", "IT_BUS=",
   "MESSAGE_SNOOP=INFO_MED"];
```

If message snoop is disabled globally, but configured for a service/port, it is enabled for that service/port with the specified configuration only. For example:

```
artix:interceptors:message_snoop:enabled = "false";

artix:interceptors:message_snoop:http://www.acme.com/test
   s:myService:myPort:log_level = "WARNING";
artix:interceptors:message_snoop:http://www.acme.com/test
   s:myService:myPort:log_subsystem = "MY_SUBSYSTEM";

event_log:filters = ["*=WARNING",
   "IT_BUS=INFO_HI+WARN+ERROR", "MY_SUBSYSTEM=WARNING"];
```

Setting message snoop in conjunction with log filters is useful when you wish to trace only messages that are relevant to a particular service, and you do not wish to see logging for others (for example, the container, locator, and so on).

Configuring SNMP Logging

SNMP

*Simple Network Management Protocol (*SNMP) is the Internet standard protocol for managing nodes on an IP network. SNMP can be used to manage and monitor all sorts of equipment (for example, network servers, routers, bridges, and hubs).

The Artix SNMP `LogStream` plug-in uses the open source library `net-snmp` (v.5.0.7) to emit SNMP v1/v2 traps. For more information on this implementation, see http://sourceforge.net/projects/net-snmp/. To obtain a freeware SNMP Trap Receiver, visit http://www.ncomtech.com.

Artix Management Information Base (MIB)

A *Management Information Base* (MIB) file is a database of objects that can be managed using SNMP. It has a hierarchical structure, similar to a directory tree. It contains both pre-defined values and values that can be customized. The Artix MIB is shown below:

Example 5: *Artix MIB*

```
ARTIX-MIB DEFINITIONS  ::= BEGIN

 IMPORTS
       MODULE-IDENTITY, OBJECT-TYPE,
       Integer32, Counter32,
       Unsigned32,
       NOTIFICATION-TYPE              FROM   SNMPv2-SMI
       DisplayString                  FROM   RFC1213-MIB
;

-- v2 s/current/current


 Micro Focus OBJECT IDENTIFIER ::= { iso(1) org(3) dod(6) internet(1) private(4)
   enterprises(1) 5043 }

 ArtixMib MODULE-IDENTITY
 LAST-UPDATED "201503210000Z"

 ORGANIZATION "Micro FocusInternational plc"

 CONTACT-INFO
             "
                Corporate Headquarters
                The Lawn, 22-30 Old Bath Road
                Newbury
                Berkshire  RG14 1QN
                UK
                Tel: +44 (0) 1635 565200
                http://www.microfocus.com

             "
 DESCRIPTION
       "This MIB module defines the objects used and format of SNMP traps that are generated
       from the Event Log for Artix based systems from Micro Focus"

 ::= { artix 1 }
```

Example 5: *Artix MIB*

```
--                     Micro Focus (5043)

--                       |
--                     microfocusMib(1)
--                       |
--              ___________________________________
--             |                |                  |
--           orbix3(2)       IONAAdmin (3)         Artix (4)
-                                                   |
--                                          --------------------
--                                          |                  |
--                                 ArtixEventLogMibObjects(0)  ArtixEventLogMibTraps (1)
--                                     |                              |
--               ---------------------------------------       ----------------------
--                                |- eventSource (1)                 |- ArtixbaseTrapDef (1)
--                                |- eventId (2)
--                                |- eventPriority (3)
--                                |- timeStamp (4)
--                                |- eventDescription (5)



Artix                          OBJECT IDENTIFIER  ::= { ionaMib 4 }
ArtixEventLogMibObjects        OBJECT IDENTIFIER  ::= { Artix 0 }
ArtixEventLogMibTraps          OBJECT IDENTIFIER  ::= { Artix 1 }
ArtixBaseTrapDef               OBJECT IDENTIFIER  ::= { ArtixEventLogMibTraps 1 }


-- MIB variables used as varbinds
eventSource         OBJECT-TYPE
   SYNTAX      DisplayString (SIZE(0..255))
   MAX-ACCESS  not-accessible
   STATUS      current
   DESCRIPTION
       "The component or subsystem which generated the event."
    ::= { ArtixEventLogMibObjects 1 }
```

Example 5: *Artix MIB*

```
eventId        OBJECT-TYPE
   SYNTAX      INTEGER
   MAX-ACCESS  not-accessible
   STATUS      current
   DESCRIPTION
       "The event id for the subsystem which generated the event."

   ::= { ArtixEventLogMibObjects 2 }

eventPriority      OBJECT-TYPE
   SYNTAX          INTEGER
   MAX-ACCESS      not-accessible
   STATUS          current
   DESCRIPTION
       "The severity level of this event.  This maps to IT_Logging::EventPriority types.  All
        priority types map to four general types: INFO (I), WARN (W), ERROR (E), FATAL_ERROR (F)"

   ::= { ArtixEventLogMibObjects 3 }

timeStamp       OBJECT-TYPE
   SYNTAX      DisplayString (SIZE(0..255))
   MAX-ACCESS  not-accessible
   STATUS      current
   DESCRIPTION
       "The time when this event occurred."

   ::= { ArtixEventLogMibObjects 4 }

eventDescription       OBJECT-TYPE
   SYNTAX           DisplayString (SIZE(0..255))
   MAX-ACCESS       not-accessible
   STATUS           current
   DESCRIPTION
       "The component/application description data included with event."

   ::= { ArtixEventLogMibObjects 5 }

-- SNMPv1 TRAP definitions
-- ArtixEventLogBaseTraps   TRAP-TYPE
--    OBJECTS {
--       eventSource,
--       eventId,
--       eventPriority,
```

Example 5: *Artix MIB*

```
--       timestamp,
--       eventDescription
--    }

--    STATUS current
--    ENTERPRISE iona
--    VARIABLES { ArtixEventLogMibObjects }
--    DESCRIPTION  "The generic trap generated from an Artix Event Log."
--    ::= { ArtixBaseTrapDef 1 }

-- SNMPv2 Notification type

 ArtixEventLogNotif   NOTIFICATION-TYPE
    OBJECTS {
       eventSource,
       eventId,
       eventPriority,
       timestamp,
       eventDescription
    }

    STATUS current
    ENTERPRISE iona
    DESCRIPTION  "The generic trap generated from an Artix Event Log."
    ::= { ArtixBaseTrapDef 1 }

END
```

SNMP integration

Events received from various Artix components are converted into SNMP management information. This information is sent to designated hosts as SNMP traps, which can be received by any SNMP managers listening on the hosts. In this way, Artix enables SNMP managers to monitor Artix-based systems.

Artix supports SNMP version 1 and 2 traps only.

Artix provides a log stream plug-in called `snmp_log_stream`. The shared library name of the SNMP plug-in found in the `artix.cfg` file is:

```
plugins:snmp_log_stream:shlib_name = "it_snmp"
```

Configuring the SNMP plugin

The SNMP plugin has five configuration variables, whose defaults can be overridden by the user. The availability of these variables is subject to change. The variables and defaults are:

```
plugins:snmp_log_stream:community = "public";
plugins:snmp_log_stream:server    = "localhost";
plugins:snmp_log_stream:port      = "162";
plugins:snmp_log_stream:trap_type = "6";
plugins:snmp_log_stream:oid       = "your IANA number in dotted decimal notation"
```

Configuring the Enterprise Object Identifier

The last variable described, `oid`, is the Enterprise Object Identifier. This is assigned to specific enterprises by the Internet Assigned Numbers Authority (IANA). The first six numbers correspond to the prefix: `iso.org.dod.internet.private.enterprise` (`1.3.6.1.4.1`). Each enterprise is assigned a unique number, and can provide additional numbers to further specify the enterprise and product.

For example, the `oid` for Micro Focus is `5043`. The additional number `1.4.1.0` specifies Artix. Therefore the complete OID for Artix is `1.3.6.1.4.1.5043.1.4.1.0`. To find the number for your enterprise, visit the IANA website at http://www.iana.org.

The SNMP plug-in implements the `IT_Logging::LogStream` interface and therefore acts like the `local_log_stream` plug-in.

Enterprise Performance Logging

Performance logging plug-ins enable Artix to integrate effectively with third-party Enterprise Management Systems (EMS).

Enterprise Management Integration

The performance logging plug-ins enable Artix to integrate effectively with *Enterprise Management Systems* (EMS), such as IBM Tivoli™, HP OpenView™, or BMC Patrol™. The performance logging plug-ins can also be used in isolation or as part of a bespoke solution.

Enterprise Management Systems enable system administrators and production operators to monitor enterprise-critical applications from a single management console. This enables them to quickly recognize the root cause of problems that may occur, and take remedial action (for example, if a machine is running out of disk space).

Performance logging

When performance logging is configured, you can see how each Artix server is responding to load. The performance logging plug-ins log this data to file or `syslog`. Your EMS (for example, IBM Tivoli) can read the performance data from these logs, and use it to initiate appropriate actions, (for example, issue a restart to a server that has become unresponsive, or start a new replica for an overloaded cluster).

Example EMS integration

Figure 1 shows an overview of the Artix and IBM Tivoli integration at work. In this example, a restart command is issued to an unresponsive server.

In Figure 1, the performance log files indicate a problem. The Artix Tivoli Provider uses the log file interpreter to read the logs. The provider sees when a threshold is exceeded and fires an event. The event causes a task to be activated in the Tivoli Task Library. This task restarts the appropriate server.

This chapter explains how to manually configure the performance logging plug-ins. It also explains the format of the performance logging messages.

For details on how to integrate your EMS environment with Artix, see the ***Artix Management Guide, C++ Runtime***.

Configuring Performance Logging

This section explains how to manually configure performance logging. This section includes the following:

- "Performance logging plug-in".
- "Monitoring Artix requests".
- "Specifying a log file".
- "Monitoring clusters".
- "Configuring a server ID".
- "Configuring a client ID".
- "Performance Logging Message Formats".

Performance logging plug-in

The performance logging component includes the following plug-ins:

Table 6: *Performance Logging Plug-in*

Plug-in	Description
Response monitor	Monitors response times of requests as they pass through the Artix binding chains. Performs the same function for Artix as the response time logger does for Orbix.
Collector	Periodically collects data from the response monitor plug-in and logs the results.

Monitoring Artix requests

You can use performance logging to monitor Artix server and client requests.

To monitor both client and server requests, add the `bus_response_monitor` plug-in to the `orb_plugins` list in the global configuration scope. For example:

```
orb_plugins = ["xmlfile_log_stream", "soap", "at_http",
   "bus_response_monitor"];
```

To configure performance logging on the client side only, specify this setting in a client scope only.

Logging to a file or memory

You can specify whether logging is output to a file or stored in memory using `plugins:bus_response_monitor:type` variable. Specifying `file` outputs performance logging data to a file, while

specifying `memory` places the data into memory so it can be retrieved using the Artix container service. `When file` is enabled, `memory` is also enabled. For example:

```
plugins:bus_response_monitor:type = "file";
```

Specifying a log file

You can configure the collector plug-in to log data to a specific file location.

The following example configuration results in performance data being logged to `/var/log/my_app/perf_logs/treasury_app.log`:

```
plugins:it_response_time_collector:filename =
"/var/log/my_app/perf_logs/treasury_app.log";
```

Monitoring clusters

You can configure your EMS to monitor a cluster of servers. You can do this by configuring multiple servers to log to the same file. If the servers are running on different hosts, the log file location must be on an NFS mounted or shared directory.

Alternatively, you can use `syslogd` as a mechanism for monitoring a cluster. You can do this by choosing one `syslogd` to act as the central logging server for the cluster. For example, say you decide to use a host named `teddy` as your central log server. You must edit the `/etc/syslog.conf` file on each host that is running a server replica, and add a line such as the following:

```
# Substitute the name of your log server
  user.info                                          @teddy
```

Some syslog daemons will not accept log messages from other hosts by default. In this case, it may be necessary to restart the `syslogd` on `teddy` with a special flag to allow remote log messages.

You should consult the `man` pages on your system to determine if this is necessary and what flags to use.

Configuring a server ID

You can configure a server ID that will be reported in your log messages. This server ID is particularly useful in the case where the server is a replica that forms part of a cluster.

In a cluster, the server ID enables management tools to recognize log messages from different replica instances.

You can configure a server ID as follows:

```
plugins:it_response_time_collector:server-id = "Locator-1";
```

This setting is optional; and if omitted, the server ID defaults to the ORB name of the server. In a cluster, each replica must have this value set to a unique value to enable sensible analysis of the generated performance logs.

Configuring a client ID

You can also configure a client ID that will be reported in your log messages, for example:

```
plugins:it_response_time_collector:server-id = "my_client_app";
```

This setting enables management tools to recognize log messages from client applications. This setting is optional; and if omitted, it is assumed that a server is being monitored.

Configuration example

The following simple example configuration file is from the management demo supplied in your Artix installation:

```
include "../../../../../etc/domains/artix.cfg";

demos {

  management
  {
    orb_plugins = ["xmlfile_log_stream", "soap", "at_http",
                   "bus_response_monitor"];


   client {
    plugins:it_response_time_collector:server-id=
      "management-demo-client";

    plugins:it_response_time_collector:filename=
      "management_demo_client.log";
   };

   server {
     plugins:it_response_time_collector:server-id=
       "management-demo-server";

     plugins:it_response_time_collector:filename=
       "management_demo_server.log";
   };
  };
};
```

In this example, the `bus_response_monitor` plug-in is set in the global scope. This specifies settings for both the client and server applications.

Performance Logging Message Formats

This section describes the performance logging message formats used by Artix. It includes the following:

- "Artix log message format".
- "Simple life cycle message formats".

Artix log message format

Performance data is logged in a well-defined format. For Artix applications, this format is as follows:

```
YYYY-MM-DD HH:MM:SS server=ServerID [namespace=nnn service=sss
   port=ppp operation=name] count=n avg=n max=n min=n int=n oph=n
```

Table 7: *Artix log message arguments*

Argument	Description
server	The server ID of the process that is logging the message.
namespace	The Artix namespace.
service	The Artix service.
port	The Artix port.
operation	The name of the operation for CORBA invocations or the URI for requests on servlets.
count	The number of operations of invoked (IIOP). or The number of times this operation or URI was logged during the last interval (HTTP).
avg	The average response time (milliseconds) for this operation or URI during the last interval.
max	The longest response time (milliseconds) for this operation or URI during the last interval.
min	The shortest response time (milliseconds) for this operation or URI during the last interval.
int	The number of milliseconds taken to gather the statistics in this log file.
oph	Operations per hour.

The combination of namespace, service and port above denote a unique Artix endpoint.

Simple life cycle message formats

The server will also log simple life cycle messages. All servers share the following common format.

```
YYYY-MM-DD HH:MM:SS server=ServerID status=CurrentStatus
```

Table 8: *Simple life cycle message formats arguments*

Argument	Description
server	The server ID of the process that is logging the message.
status	A text string describing the last known status of the server (for example, `starting_up`, `running`, `shutting_down`).

Remote Performance Logging

The performance logging plug-ins can be configured to log data to a local file or to a remote endpoint. Depending on your specific architecture, it might not always be desirable or feasible to deploy the required management tools on a particular platform. In this case, it would not be appropriate to persist the performance logging data to a local file, because there would be no local application to consume it.

In some situations, NFS or a similar file sharing mechanism might be used to persist data across your distributed system. However, security and performance concerns often prevent the use of such protocols. In such cases, Artix provides a remote logging facility for the purposes of sending logging data to a remote endpoint where the data can be persisted and subsequently consumed by an application that is native to that remote system.

Components of a remote logging framework

The components of a remote logging framework are as follows:

- The performance logging *collector* plug-in runs within a deployed application on the source host. This is the host that sends its logging data to a remote endpoint. The collector is configured to harvest the required performance logging data and to write this data to a remote CORBA endpoint (instead of, for example, to a local file on the source host).

  Note: Remote logging is only supported in the C++ version of the performance logging collector plug-in.

- The *remote logger daemon* is an Artix application that is deployed on the remote target host. It loads the remote log receiver servant, which is accepts the performance logging data from the source applications and logs this data to a local file on the target host.

- The *EMS component* (for example, a Tivoli or BMC Patrol agent) runs on the remote target host. It consumes the data from the file and propagates the performance information to the centralized region manager.

Figure 1 shows how remote logging works in Artix.

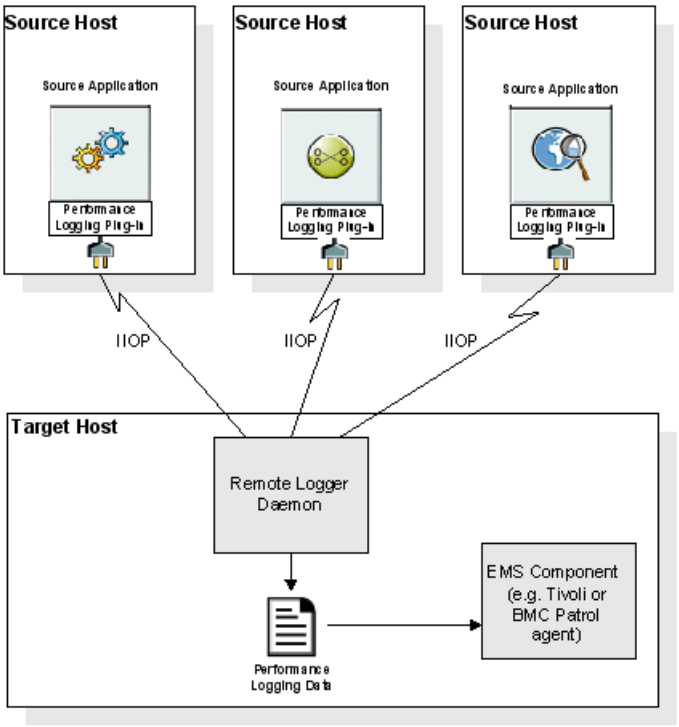


Figure 1: *Remote Logging Framework*

Deploying a remote logger daemon

As explained in "Components of a remote logging framework" on page 48, the remote logger daemon loads the remote log receiver servant, which accepts the performance logging data from the source application(s), and logs this data to a local file on the target host. You may deploy the remote logger plug-in in any Artix application. The remote logger plug-in should be deployed in a standalone container whose sole purpose is to log data from one or more source applications. The local file on the remote host can then be consumed by the EMS agent running on that host, or used as part of some custom-made solution.

Points to note

The following points should be noted:

- IIOP is used for the data communication between the collector and the remote logger daemon. This adds very low overhead to the logging payload, because it uses a binary protocol on the wire (CDR).

- To secure the message transfer, IIOP/TLS can be used for data communication between the collector and the remote logger daemon.
- The timestamps embedded in the remote logging data are localized to the specific source system on which the monitored application is running. You must ensure that the system clocks on all participating systems are synchronized to an acceptable level, as governed by your EMS or your custom-made solution.

Configuring Remote Performance Logging

This section explains how to configure remote logging, which enables you to send logging data to a remote endpoint on another host rather than to a local file.

Configuring the Remote Logger Daemon

To configure the remote logger daemon that runs on the remote target host, add the following configuration scope and settings to the Artix configuration domain:

```
…
remote_logger_daemon
{
    orb_plugins = ["local_log_stream", "remote_log_receiver"];
    event_log:filters = ["IT_MGMT_LOGGING=*"];

    plugins:remote_log_receiver:log_filename =
       "/var/logs/remote_perflogs.txt";
    plugins:remote_log_receiver:ior_filename =
       "/var/publish/logger_ref.txt";
    plugins:remote_log_receiver:iiop:addr_list = ["host:port"];
    plugins:remote_log_receiver:prerequisite_plugins =
       ["iiop_profile", "giop", "iiop"];
};
…
```

Note: You may add this configuration scope directly to your Artix configuration domain in `artix.cfg`, or you may create a separate configuration file that includes `artix.cfg`.

Remote logging configuration settings

The settings for the `remote_log_receiver` plug-in are explained as follows:

`plugins:remote_log_receiver:log_filename`	This is the local file on the remote host to which all logs are directed.
`plugins:remote_log_receiver:ior_filename`	When the remote logger daemon is started, it writes a stringified Interoperable Object Reference (IOR) to the file specified by this configuration item. This IOR may be subsequently made available to the source applications that are acting as clients of the remote logger. However, this is not required if the source applications use a corbaloc URL rather than an IOR to contact the remote logger.
`plugins:remote_log_receiver:iiop:addr_list`	This specifies the hostname or IP address of the host on which the remote logger is running, and the port that it uses to listen for logging requests.
`plugins:remote_log_receiver:prerequisite_plugins`	This must specify the IIOP plug-ins that the remote logger needs for communication with the source host(s).

TLS security

If you are using TLS security:

- Ensure that you replace the `plugins:remote_log_receiver:iiop:addr_list` configuration item with `plugins:remote_log_receiver:iiop_tls:addr_list`.
- Ensure that the `plugins:remote_log_receiver:prerequisite_plugins` configuration item lists `"iiop_tls"` rather than `"iiop"`.

Running the remote logger daemon

To run the remote logger daemon, run the Artix container as follows:

```
it_container -ORBname remote_logger_daemon
```

Note: This is assuming that the relevant configuration scope is called `remote_logger_daemon`.

Configuring a deployed application on the source host

You must also configure your deployed application to use performance logging with the remote logger capability. For the purposes of illustration, it describes the steps that are required to configure an Artix for z/OS application.

Configuration steps

To enable a deployed application (for example, on z/OS) to use performance logging with the remote logger capability:

1. Ensure that the remote logger daemon has been configured correctly and deployed on the target host, as described in "Configuring the Remote Logger Daemon" on page 50.
2. Open the configuration domain for your deployed application (by default, this is *artixhlq*.CONFIG(ARTIX) for Artix for z/OS applications).
3. Go to the appropriate configuration scope for your application.
4. Add `it_response_time_logger` to the end of the ORB plug-ins list setting. Also, ensure that IIOP is enabled for the application, for example:

```
orb_plugins = ["local_log_stream", "iiop_profile", "giop",
   "iiop", …, "it_response_time_logger"];
```

Note: Ensure that you have a management license available.

5. Add `it_response_time_logger` to the server binding list for the application. For example:

```
binding:server_binding_list =
   ["SOAP+it_response_time_logger",
    "it_response_time_logger"];
```

6. Add the following collector plug-in configuration variables:

```
# update the log every 30 seconds
plugins:it_response_time_collector:period = "30";

# the id of the server for the log output
plugins:it_response_time_collector:server-id = "server-id";

# the remote endpoint details:
plugins:it_response_time_collector:remote_logging_enabled = "true";
initial_references:IT_PerfLoggingReceiver:reference =
   "corbaloc:iiop:1.2@remote_host:1234/IT_PerfLoggingReceiver ";
```

Note: Ensure that the *server-id* value is replaced with the actual server ID for the log output (for example `cics-server-adapter-1`).

Example output

The following is example output from the performance log on the remote file system where a number of different operations have been run against the application:

```
2006-10-18 10:08:22 server=cics-server-adapter-1 status=starting_up
2006-10-18 10:08:22 server=cics-server-adapter-1 status=running
2006-10-18 10:08:52 server=cics-server-adapter-1 status=running
2006-10-18 10:09:22 server=cics-server-adapter-1 status=running
2006-10-18 10:09:22 server=cics-server-adapter-1 [ operation=test_bounded ] count=1
   avg=110 max=110 min=110
int=30001 oph=119
2006-10-18 10:09:22 server=cics-server-adapter-1 [ operation=test_unbounded ] count=1
   avg=809 max=809 min=809
int=30001 oph=119
2006-10-18 10:09:52 server=cics-server-adapter-1 status=running
2006-10-18 10:09:52 server=cics-server-adapter-1 [ operation=call_me ] count=1 avg=793
   max=793 min=793
int=29998 oph=120
2006-10-18 10:10:22 server=cics-server-adapter-1 status=running
2006-10-18 10:10:22 server=cics-server-adapter-1 [ operation=_get_currentMappings ]
   count=1 avg=0 max=0 min=0
int=30000 oph=120
2006-10-18 10:10:52 server=cics-server-adapter-1 status=running
2006-10-18 10:11:22 server=cics-server-adapter-1 status=running
2006-10-18 10:11:52 server=cics-server-adapter-1 status=running
2006-10-18 10:12:22 server=cics-server-adapter-1 status=running
2006-10-18 10:12:22 server=cics-server-adapter-1 [ operation=resolve ] count=1 avg=0
   max=0 min=0 int=29999 oph=120
2006-10-18 10:12:52 server=cics-server-adapter-1 status=running
2006-10-18 10:12:57 server=cics-server-adapter-1 status=shutdown_started
2006-10-18 10:12:57 server=cics-server-adapter-1 status=shutdown_complete
```

Using Artix with International Codesets

The Artix SOAP and CORBA bindings enable you to transmit and receive messages in a range of codesets.

Introduction to International Codesets

A *coded character set*, or *codeset* for short, is a mapping between integer values and characters that they represent. The best known codeset is ASCII (American Standard Code for Information Interchange). ASCII defines 94 graphic characters and 34 control characters using the 7-bit integer range.

European languages

The 94 characters defined by the ASCII codeset are sufficient for English, but they are not sufficient for European languages, such as French, Spanish, and German.

To remedy the situation, an 8-bit codeset, ISO 8859-1, also known as Latin-1, was invented. The lower 7-bit portion is identical to ASCII. The extra characters in the upper 8-bit range cover those languages used widely in Western Europe.

Many other codesets are defined under ISO 8859 framework. These cover languages in other regions of Europe, as well as Russian, Arabic and Hebrew. The most recent addition is ISO 8859-15, which is a revision of ISO 8859-1. This adds the Euro currency symbol and other letters while removing less used characters.

For further information about ISO-8859-x encoding, see the following web site: "The ISO 8859 Alphabet Soup" (http://czyborra.com/charsets/iso8859.html).

Ideograms

Asian countries that use ideograms in their writing systems need more characters than fit in an 8-bit integer. Therefore, they invented double-byte codesets, where a character is represented by a bit pattern of 2 bytes.

These languages also needed to mix the double-byte codeset with ASCII in a single text file. So, *character encoding schemas*, or simply *encodings*, were invented as a way to mix characters of multiple codesets.

Some of the popular encodings used in Japan include:

- Shift JIS
- Japanese EUC
- Japanese ISO 2022

Unicode

Unicode is a codeset that aims to assign a unique number, or code point, to every character that exists (and even once existed) in all languages. To accomplish this, Unicode, which began as a double-byte codeset, has been expanded into a quadruple-byte codeset.

Unicode, in pure form, can be difficult to use within existing computer architectures, because many APIs are byte-oriented and assume that the byte value `0` means the end of the string.

For this reason, Unicode Transformation Format for 8-bit channel, or UTF-8, is frequently used. When browsers list “Unicode” in its encoding selection menu, they usually mean UTF-8, rather than the pure form of Unicode.

For more information about Unicode and its variants, visit Unicode (http://www.unicode.org/).

Charset names

To address the need for computer networks to connect different types of computers that use different encodings, the Internet Assigned Number Authority, or IANA, has a registry of encodings at http://www.iana.org/assignments/character-sets.

IANA names are used by many Internet standards including MIME, HTML, and XML. Table 9 lists IANA names for some popular charsets.

Table 9: *IANA Charset Names*

IANA Name	Description
US-ASCII	7-bit ASCII for US English
ISO-8859-1	Western European languages
UTF-8	Byte oriented transformation of Unicode
UTF-16	Double-byte oriented transformation of Unicode
Shift_JIS	Japanese DOS & Windows
EUC-JP	Japanese adaptation of generic EUC scheme, used in UNIX
ISO-2022-JP	Japanese adaptation of generic ISO 2022 encoding scheme

Note: IANA names are case insensitive. For example, US-ASCII can be spelled as us-ascii or US-ascii.

CORBA names

In CORBA, codesets are identified by numerical values registered with the Open Group’s registry, OSF Codeset Registry: ftp://ftp.opengroup.org/pub/code_set_registry/code_set_registry1.2g.txt.

Java names

Java uses IANA charset names, but recent Java versions also recognize the older "historical" names used by earlier Java versions. See http://docs.oracle.com/javase/6/docs/api/java/nio/charset/Charset.html for details.

Note: Artix uses IANA charset names even for CORBA codesets.

Working with Codesets using SOAP

Because SOAP messages are XML based, they are composed primarily of character data that can be encoded using any of the existing codesets. If the applications in a system are using different codesets, they can not interpret the messages passing between them. The Artix SOAP plug-in uses the XML prologue of SOAP messages to ensure that it stays in sync with the applications that it interacts with.

Making requests

When making requests or broadcasting a message, the SOAP plug-in determines the codeset to use from its Artix configuration scope. You can set the SOAP plug-in's character encoding using the `plugins:soap:encoding` configuration variable. This takes the IANA name of the desired codeset. The default value is `UTF-8`.

For more information on this configuration variable, see the ***Artix Configuration Reference***. For general information on configuring Artix applications, see "Getting Started" on page 3.

Responding to SOAP requests

When an Artix server receives a SOAP message, it checks the XML prologue to see what encoding codeset the message uses. If the XML prologue specifies the message's codeset, Artix uses the specified codeset to read the message and to write out its response to the request. For example, an Artix server that receives a request with the XML prologue shown in Example 6 decodes the message using `UTF-16` and encodes its response using `UTF-16`.

Example 6: *XML Prologue*

```
<?xml version="1.0" encoding="UTF-16"?>
```

If an Artix server receives a SOAP message where the XML prologue does not include the `encoding` attribute, the server will use whatever default codeset is specified in its configuration to decode the message and encode the response.

Working with Codesets using CORBA

The Artix CORBA plug-in supports both wide characters and narrow characters to accommodate an array of codesets. It also supports *codeset negotiation*. Codeset negotiation is the process by which two CORBA processes which use different *native codeset*s determine which codeset to use as a *transmission codeset*. Occasionally, the process requires the selection of a *conversion codeset* to transmit data between the two processes. The algorithm is defined in section 13.10.2.6 "Code Set Negotiation" of the CORBA 2.6.1 specification.

Native codeset

A native codeset (NCS) is a codeset that a CORBA program speaks natively.

For JAX-RPC, this is UTF-8 (`0x05010001`) for `char` and `String`, and UTF-16 (`0x00010109`) for `wchar` and `wstring`.

For C and C++, this is the encoding that is set by `setlocale()`, which in turn depends on the `LANG` and `LC_xxxx` environment variables.

You can configure the Artix CORBA plug-in's native codesets using the configuration variables listed in Table 10.

Table 10: *Configuration Variables for CORBA Native Codeset*

Configuration Variable	Description
`plugins:codeset:char:ncs`	Specifies the native codeset for narrow character and string data.
`plugins:codeset:wchar:ncs`	Specifies the native codeset for wide character and string data.

Conversion codeset

A conversion codeset (CCS) is an alternative codeset that the application registers with the ORB. More than one CCS can be registered for each of the narrow and wide interfaces. CCS should be chosen so that the expected input data can be converted to and from the native codeset without data loss. For example, Windows code page 1252 (`0x100204e4`) can be a conversion codeset for ISO-8859-1 (`0x00010001`), assuming only the common characters between the two codesets are used in the data.

You can configure the Artix CORBA plug-in's list of conversion codesets using the configuration variables listed in Table 11.

Table 11: *Configuration Variables for CORBA Conversion Codesets*

Configuration Variable	Description
`plugins:codeset:char:ccs`	Specifies the list of conversion codesets for narrow character and string data.
`plugins:codeset:wchar:ccs`	Specifies the list of conversion codesets for wide character and string data.

Transmission codeset

A transmission codeset (TCS) is the codeset agreed upon after the codeset negotiation. The data on the wire uses this codeset. It is either the native codeset, one of the conversion codesets, or UTF-8 for the narrow interface and UTF-16 for the wide interface.

Negotiation algorithm

Codeset negotiation uses the following algorithm to determine which codeset to use in transferring data between client and server:

1. If the client and server are using the same native codeset, no translation is required.
2. If the client has a converter to the server's codeset, the server's native codeset is used as the transmission codeset.
3. If the client does not have an appropriate converter and the server does have a converter to the client's codeset, the client's native codeset is used as the transmission codeset.
4. If neither the client nor the server has an appropriate converter, the server ORB tries to find a conversion codeset that both server and client can convert to and from without loss of data. The selected conversion codeset is used as the transmission codeset.
5. If no conversion codeset can be found, the server ORB determines if using UTF-8 (narrow characters) or UTF-16 (wide characters) will allow communication between the client

and server without loss of data. If UTF-8 or UTF-16 is acceptable, it is used as the transmission codeset. If not, a `CODESET_INCOMPATIBLE` exception is raised.

Codeset compatibility

The final steps involve a compatibility test, but the CORBA specification does not define when a codeset is compatible with another. The compatibility test algorithm employed in Orbix is outlined below:

1. ISO 8859 Latin-*n* codesets are compatible.
2. UCS-2 (double-byte Unicode), UCS-4 (four-byte Unicode), and UTF-x are compatible.
3. All other codesets are not compatible with any other codesets.

This compatibility algorithm is subject to change without notice in future releases. Therefore, it is best to configure the codeset variables as explicitly as possible to reduce dependency on the compatibility algorithm.

Working with Codesets using Fixed Length Records

Artix fixed record length support enables Artix to interact with mainframe systems using COBOL. For example, many COBOL applications send fixed length record data over WebSphere MQ.

Artix provides a fixed binding that maps logical messages to concrete fixed record length messages. This binding enables you to specify attributes such as encoding style, justification, and padding character.

Encoding attribute

The Artix fixed binding provides an optional `encoding` attribute for both its `fixed:binding` and `fixed:body` elements. The `encoding` attribute specifies the codeset used to encode the text data. Valid values are any IANA codeset name. See http://www.iana.org/assignments/character-sets for details.

The `encoding` attribute for the `fixed:binding` element is a global setting; while the `fixed:body` attribute is per operation. Both settings are optional. If you do not set either, the default value is `UTF-8`.

For more details, see *`ArtixInstallDir`*`\schemas\fixed-binding.xsd`.

Fixed binding example

The following WSDL example shows a fixed binding with encoding attributes for `fixed:body` elements. This binding includes two operations, `echoVoid` and `echoString`.

Example 7: *Fixed Length Record Binding*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:fixed="http://schemas.iona.com/bindings/fixed"
   xmlns:http="http://schemas.iona.com/transports/http"
   xmlns:http-conf="http://schemas.iona.com/transports/http/configuration"
   xmlns:iiop="http://schemas.iona.com/transports/iiop_tunnel"
   xmlns:mq="http://schemas.iona.com/transports/mq"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.iona.com/artix/test/I18nBase/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsd1="http://www.iona.com/artix/test/I18nBase" name="I18nBaseService"
   targetNamespace="http://www.iona.com/artix/test/I18nBase/"

    <message name="echoString">
        <part name="stringParam0" type="xsd:string"/>
    </message>

    <message name="echoStringResponse">
        <part name="return" type="xsd:string"/>
    </message>

    <message name="echoVoid"/>
    <message name="echoVoidResponse"/>

    <portType name="I18nBasePortType">
        <operation name="echoString">
            <input message="tns:echoString" name="echoString"/>
            <output message="tns:echoStringResponse" name="echoStringResponse"/>
        </operation>
        <operation name="echoVoid">
            <input message="tns:echoVoid" name="echoVoid"/>
            <output message="tns:echoVoidResponse" name="echoVoidResponse"/>
        </operation>
    </portType>
```

Example 7: *Fixed Length Record Binding*

```
   <binding name="I18nFIXEDBinding" type="tns:I18nBasePortType">
       <fixed:binding/>
       <operation name="echoString">
           <fixed:operation discriminator="discriminator"/>
           <input name="echoString">
               <fixed:body encoding="ISO-8859-1">
                   <fixed:field bindingOnly="true" fixedValue="01"
  name="discriminator"/>
                   <fixed:field name="stringParam0" size="50"/>
               </fixed:body>
           </input>
           <output name="echoStringResponse">
               <fixed:body encoding="ISO-8859-1">
                   <fixed:field name="return" size="50"/>
               </fixed:body>
           </output>
       </operation>

 <operation name="echoVoid">
           <fixed:operation discriminator="discriminator"/>
           <input name="echoVoid">
               <fixed:body>
                   <fixed:field name="discriminator" fixedValue="02"
  bindingOnly="true"/>
               </fixed:body>
           </input>
           <output name="echoVoidResponse">
               <fixed:body/>
           </output>
       </operation>
    </binding>
</definitions>
```

Further information

For more details on the Artix fixed length binding, see ***Artix Bindings and Transports, C++ Runtime***.

Working with Codesets using Message Interceptors

Artix provides support for codeset conversion for transports that do not have their own concept of headers. For example, IBM WebSphere MQ, and BEA Tuxedo. This generic support is implemented using an Artix message interceptor and WSDL port extensors.

For example, an Artix C++ client could use Artix Mainframe to access a mainframe system, using a binding for fixed length record over MQ. In this scenario, an Artix message interceptor can be configured to enable codeset conversion between ASCII and EBCDIC (Extended Binary Coded Decimal Interchange Code).

You can enable this codeset conversion simply by editing your WSDL file, or by using accessor methods in your application code. This section explains how to use both of these approaches.

Note: Codeset conversion set in application code takes precedence over the same settings in a WSDL file.

Codeset conversion attributes

This generic support for codeset conversion is implemented using a message interceptor. This message interceptor manipulates the following codeset conversion attributes:

`LocalCodeSet`	Specifies the codeset used locally by a client or server application.
`OutboundCodeSet`	Specifies the codeset used by the application for outgoing messages.
`InboundCodeSet`	Specifies the codeset used by the application for incoming messages.

You can specify these attributes to convert client-side requests and server-side responses. All three attributes are optional.

Configuring codeset conversion in a WSDL file

You can configure codeset conversion by setting the codeset conversion attributes in a WSDL file. Example 8 shows the contents of the Artix internationalization schema (`i18n-context.xsd`).

Example 8: *Artix i18n Schema*

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
           targetNamespace="http://schemas.iona.com/bus/i18n/context"
           xmlns:i18n-context="http://schemas.iona.com/bus/i18n/context"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified">

  <xs:import namespace = "http://schemas.xmlsoap.org/wsdl/"
   schemaLocation="wsdl.xsd"/>

  <xs:element name="client" type="i18n-context:ClientConfiguration" />

  <xs:complexType name="ClientConfiguration">

   <xs:annotation>
       <xs:documentation> I18n Client Context Information
       </xs:documentation>
   </xs:annotation>

    <xs:complexContent>
      <xs:extension base="wsdl:tExtensibilityElement" >
        <xs:attribute name="LocalCodeSet" type="xs:string" use="optional" />
        <xs:attribute name="OutboundCodeSet" type="xs:string" use="optional" />
        <xs:attribute name="InboundCodeSet" type="xs:string" use="optional" />
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
 <xs:element name="server" type="i18n-context:ServerConfiguration"/>

  <xs:complexType name="ServerConfiguration" >
   <xs:annotation>
       <xs:documentation> I18n Server Context Information
       </xs:documentation>
   </xs:annotation>

    <xs:complexContent>
      <xs:extension base="wsdl:tExtensibilityElement" >
        <xs:attribute name="LocalCodeSet" type="xs:string" use="optional" />
        <xs:attribute name="OutboundCodeSet" type="xs:string" use="optional" />
        <xs:attribute name="InboundCodeSet" type="xs:string" use="optional" />
      </xs:extension>
    </xs:complexContent>

  </xs:complexType>

</xs:schema>
```

The Artix internationalization message interceptor uses this schema as a port extensor. This enables you to configure codeset conversion attributes in a WSDL file.

Client/server WSDL example

The following example shows codeset conversion settings for a client and a server application specified in a sample WSDL file:

Example 9: *i18n Specified in a WDSL File*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="I18nBaseService"
   targetNamespace="http://www.iona.com/artix/test/I18nBase/"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.iona.com/artix/test/I18nBase/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:mq="http://schemas.iona.com/transports/mq"
    xmlns:http="http://schemas.iona.com/transports/http"
    xmlns:http-conf="http://schemas.iona.com/transports/http/configuration"
    xmlns:fixed="http://schemas.iona.com/bindings/fixed"
    xmlns:i18n-context="http://schemas.iona.com/bus/i18n/context"
    xmlns:xsd1="http://www.iona.com/artix/test/I18nBase">

  <import namespace="http://www.iona.com/artix/test/I18nBase"
   location="./I18nServiceBindings.wsdl"/>

    <service name="I18nService">

       <port binding="tns:I18nFIXEDBinding" name="I18nFIXED_HTTPPort">
             <http:address location="http://localhost:0"/>
        <i18n-context:client LocalCodeSet="ISO-8859-1" InboundCodeSet="UTF-8"/>
        <i18n-context:server LocalCodeSet="UTF-8" OutboundCodeSet="ISO-8859-1"/>
       </port>

     <port binding="tns:I18nFIXEDBinding" name="I18nFIXED_MQPort">

       <mq:client QueueManager="MY_DEF_QM" QueueName="MY_FIRST_Q" AccessMode="send"
         ReplyQueueManager="MY_DEF_QM" ReplyQueueName="REPLY_Q"
        CorrelationStyle="messageId copy" />

       <mq:server QueueManager="MY_DEF_QM" QueueName="MY_FIRST_Q"
       ReplyQueueManager="MY_DEF_QM" ReplyQueueName="REPLY_Q" AccessMode="receive"
        CorrelationStyle="messageId copy" />
       <i18n-context:client LocalCodeSet="UTF-8" InboundCodeSet=""/>
       <i18n-context:server LocalCodeSet="ISO-8859-1"/>
     </port>

  </service>

</definitions>
```

This sample WSDL file shows a single service named `I18nService`, with two bindings and two ports named `I18nFIXED_HTTPPort` and `I18nFIXED_MQPort`. The binding in both cases is fixed length record, each with a single operation.

Enabling codeset conversion in application code

You can also enable codeset conversion attributes by calling the following methods in your C++ application code:

```
namespace IT_ContextAttributes
{
    class IT_CONTEXT_ATTRIBUTE_API ClientConfiguration
    {
      void setLocalCodeSet(const IT_Bus::String & val);
      void setOutboundCodeSet(const IT_Bus::String & val);
      void setInboundCodeSet(const IT_Bus::String & val);
    };

    class IT_CONTEXT_ATTRIBUTE_API ServerConfiguration
    {
      void setLocalCodeSet(const IT_Bus::String & val);
      void setOutboundCodeSet(const IT_Bus::String & val);
      void setInboundCodeSet(const IT_Bus::String & val);
    };
}
```

An Artix `ContextContainer` in the message interceptor, and the WSDL configuration are checked for each attribute. This is performed during the client's `intercept_invoke()` method and the server's `intercept_dispatch()` method. The client request buffer or server response buffer can be converted to another encoding as needed. This conversion can occur on the outbound or inbound intercept points.

The interceptor refers to the current context on a per-thread basis. For detailed information on Artix contexts, see Developing Artix Applications with C++.

Linking with the context library

The message interceptor uses a common type library of Artix context attributes. The application must be linked with this common library, and with any transports that use this context to set or get attributes. The generated header files for this common library are available in the following directory:

```
ArtixInstallDir\include\it_bus_pdk\context_attrs
```

You must ensure that your application links with the context library that contains the generated stub code for `i18n-context.xsd`.

Client code example

Example 10 shows an example of the code that you need to add to your C++ client application:

Example 10: *Accessing i18n in C++ Client Code*

```
void
I18nTest::echoString(
    I18nBaseClient* client, const String& instr)
{
    String outstr;
    try
    {

    // Set the i18n request context to match the fixed binding encoding setting

    IT_Bus::Bus_var bus = client->get_bus();
    ContextRegistry * reg = bus->get_context_registry();

    ContextCurrent & cur = reg->get_current();
    ContextContainer * registered_ctx = cur.request_contexts();

    AnyType & i18n_ctx_info =
    registered_ctx->get_context(IT_ContextAttributes::I18N_INTERCEPTOR_CLIENT_QNAME,
   true);
    ClientConfiguration & i18n_ctx_cfg = dynamic_cast<ClientConfiguration&>
   (i18n_ctx_info);

    // Set the Inbound codeset to match the binding encoding

    static const String LOCAL_CODE_SET = "ISO-8859-1";
    i18n_ctx_cfg.setLocalCodeSet(LOCAL_CODE_SET);

    const String & local_codeset = (*i18n_ctx_cfg.getLocalCodeSet());

    client->echoString(instr, outstr);

    // Read the i18n reply context

    registered_ctx = cur.reply_contexts();

    AnyType & i18n_ctx_reply_info =

   registered_ctx->get_context(IT_ContextAttributes::I18N_INTERCEPTOR_CLIENT_QNAME,
   true);

    const ClientConfiguration & i18n_ctx_reply_cfg =
      dynamic_cast<const ClientConfiguration&> (i18n_ctx_reply_info);
```

Example 10: *Accessing i18n in C++ Client Code*

```
    const String * local_codeset_reply = i18n_ctx_reply_cfg.getLocalCodeSet();
    const String * outbound_codeset_reply = i18n_ctx_reply_cfg.getOutboundCodeSet();
    const String * inbound_codeset_reply = i18n_ctx_reply_cfg.getInboundCodeSet();

    if(local_codeset_reply)
     cout << "client LocalCodeSet reply context:" << local_codeset_reply->c_str() <<
   endl;
    if(outbound_codeset_reply)
     cout << "client OutboundCodeSet reply context:"<< outbound_codeset_reply->c_str
   << endl;
    if(inbound_codeset_reply)
     cout << "client InboundCodeSet reply context" << inbound_codeset_reply->c_str()
   << endl;
    }

    catch (IT_Bus::ContextException& ce)
    {
    ...
    }
    catch (IT_Bus::Exception& ex)
    {
    ...
    }
    catch (...)
    {
    ...
    }
}
```

Server code example

Example 10 shows example of the code that you need to add to your C++ servant application.

Example 11: *Accessing i18n in C++ Server Code*

```
void
I18nServiceImpl::echoString(
    const String& stringParam0,
    String & var_return) IT_THROW_DECL((IT_Bus::Exception))
{

    var_return = stringParam0;

   try
   {
   // Read the i18n reply context

   ContextRegistry * reg = m_bus->get_context_registry();

   ContextCurrent & cur = reg->get_current();
   ContextContainer * registered_ctx = cur.request_contexts();
```

Example 11: *Accessing i18n in C++ Server Code*

```
   AnyType & i18n_ctx_info =
   registered_ctx->get_context(IT_ContextAttributes::I18N_INTERCEPTOR_SERVER_QNAME,
   false);
   const ServerConfiguration & i18n_ctx_cfg =
   dynamic_cast<const ServerConfiguration&> (i18n_ctx_info);

   const String * local_codeset = i18n_ctx_cfg.getLocalCodeSet();
   const String * outbound_codeset = i18n_ctx_cfg.getOutboundCodeSet();
   const String * inbound_codeset = i18n_ctx_cfg.getInboundCodeSet();

   if(local_codeset)
     cout << "server LocalCodeSet request context:" << local_codeset->c_str() << endl;
   if(outbound_codeset)
    cout << "server OutboundCodeSet request context:" << outbound_codeset->c_str() <<
   endl;
   if(inbound_codeset)
     cout << "server InboundCodeSet request context:" << inbound_codeset->c_str() <<
   endl;

   // Add code to change the reply context

   registered_ctx = cur.reply_contexts();

   AnyType & i18n_reply_ctx =
   registered_ctx->get_context(IT_ContextAttributes::I18N_INTERCEPTOR_SERVER_QNAME,
   true);

   ServerConfiguration & i18n_reply_ctx_cfg =
    dynamic_cast<ServerConfiguration&> (i18n_reply_ctx);

   // Set the local codeset to match the binding encoding

   static const String LOCAL_CODE_SET = "ISO-8859-1";
   i18n_reply_ctx_cfg.setLocalCodeSet(LOCAL_CODE_SET);

   String & set_local_context = (*i18n_reply_ctx_cfg.getLocalCodeSet());

   assert(set_local_context == LOCAL_CODE_SET);
   }
   catch (IT_Bus::ContextException& ex)
    {
     cout << "Error with server context" << ex.message() << endl;
    }
  catch (IT_Bus::Exception& ex)
    {
     cout << "Error with server context" << ex.message() << endl;
    }
  catch (...)
    {
     cout << "Unknown Error with server context" << endl;
    }
}
```

Artix configuration settings

Finally, you must also enable the i18n message interceptor in your `artix.cfg` file. Example 12 shows the required settings:

Example 12: *Artix Configuration Settings*

```
// Add to a demo/application scope.
interceptor{
    binding:artix:client_message_interceptor_list =
   "i18n-context:I18nInterceptorFactory";

    binding:artix:server_message_interceptor_list =
   "i18n-context:I18nInterceptorFactory";

    orb_plugins = ["xmlfile_log_stream", "i18n_interceptor"];

    event_log:filters = ["*=WARN+ERROR+FATAL"];
};
```

Further information

For more information details on writing Artix C++ applications and on Artix contexts, see ***Developing Artix Applications with C++***.

Routing with International Codesets

When routing between applications, Artix attempts to correctly map between different codesets. If both endpoints use bindings that support internationalization (i18n), Artix uses codeset conversion. If only one of the endpoints supports internationalization, the Artix endpoint supporting internationalization attempts to use codeset conversion on the messages.

The following bindings do not natively support internationalization:

- Tagged
- G2++
- XML

However, for these bindings you can use the Artix i18n interceptor to perform codeset conversion on the message buffer before it is placed on the wire. For more details, see ***Artix Bindings and Transports, C++ Runtime***.

Routing between internationalized endpoints

When Artix is routing between internationalized endpoints, the receiving endpoint and the sending endpoint both behave independently of each other.

For example, if one endpoint of a router receives a request in Shift_JIS and the router is configured to use ISO-8859-1, the Shift_JIS request is properly decoded by the router.

However, when the request is passed on by the router, it is passed on in ISO-8859-1. If the two codesets are not compatible, there is a good chance that data will be lost in the conversion and the request will not be properly handled.

Note: If the codesets are not compatible, and data is lost in the router, Artix does not generate a warning.

Routing from non-internationalized to internationalized bindings

When Artix is routing from a non-internationalized endpoint to an internationalized endpoint, it uses the default codeset specified in the router's configuration for writing messages to internationalized endpoints. If the Artix router is configured to encode messages using a codeset that is different from the one used by the endpoint, you will lose data.

For example, if a Tuxedo application makes a request on a Web service through a router, the router receives non-internationalized data from the Tuxedo application. And the router then writes the SOAP message using the codeset specified in its configuration. If the Web service and the router are both configured to write in us-dk, the operation proceeds without a problem. The router receives the encoded response from the server and passes it back to the Tuxedo binding.

However, if the Web service is configured to accept data using us-dk, and the router is configured to encode data using Chinese, data may be lost between the router and the Web service due to codeset incompatibility.

Routing from internationalized to non-internationalized bindings

When Artix is routing SOAP messages to a non-SOAP endpoint, such as a Tuxedo server on a mainframe using the fixed plug-in, Artix handles the message transformations so that the SOAP application receives responses in the correct codeset.

For example, a Web service client in a Chinese locale encodes its requests in eucTW and invokes on a service that is hosted on a mainframe that is behind an Artix router, as shown in Figure 2.

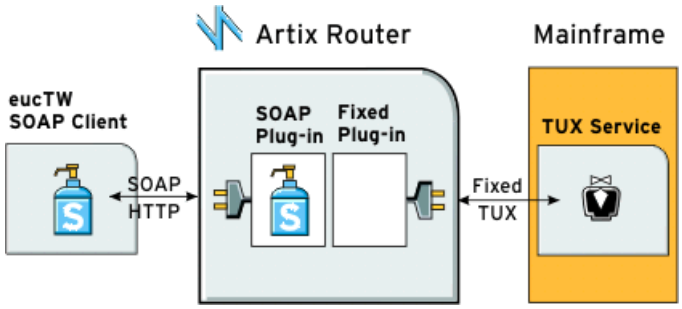


Figure 2: *Routing Internationalized Requests*

The Artix router would process the request as follows:

1. On receiving the SOAP request, the router inspects the XML prologue and decodes the message using the specified codeset (in this case, eucTW).
2. The fixed binding plug-in then writes out the message to the mainframe service.
3. When the mainframe sends its response back to the router, the fixed binding decodes the message and passes it back to the SOAP plug-in.
4. The SOAP plug-in inspects the message and determines the request to that corresponds it.
5. The SOAP plug-in then encodes the message using the codeset specified in the request (in this case, eucTW), and passes the response to the client.

Part II

Deploying Artix Services

In this part

This part contains the following chapters:

Deploying Services in an Artix Container	page 75
Deploying an Artix Transformer	page 99
Deploying a Service Chain	page 109
Deploying Artix Services for High Availability	page 115
Deploying WS-Reliable Messaging	page 129

Deploying Services in an Artix Container

The Artix container enables you to deploy and manage C++ services dynamically. For example, you can deploy a new service into a running container, or perform runtime tasks such as start, stop, and list existing services in a container.

Introduction to the Artix Container

The Artix container provides a consistent mechanism for deploying and managing Artix services. This section provides an overview the Artix container architecture and its main components. The Artix container is the recommended way to deploy Artix services. To use the container, your services should be developed as Artix plug-ins.

Artix plug-ins

You can write Artix Web service implementations as C++ plug-ins. An Artix *plug-in* is a code library that can be loaded into an Artix application at runtime.

Artix provides a platform-independent framework for loading plug-ins dynamically, based on the dynamic linking capabilities of modern operating systems (using shared libraries and DLLs).

Benefits

Writing your application as an Artix plug-in means that you need to write less code, and that you can deploy your services into an Artix container. When you deploy your service into a container, this eliminates the need to write your own C++ server mainline. Instead, you can deploy your service by simply passing the location of a generated deployment descriptor to an Artix container's administration client. This provides a powerful programming model where the code is location independent.

In addition, the Artix container retains information about the services that it deploys. This enables the container to reload services dynamically when it restarts.

Main components

The Artix container architecture includes the following main components:

- Artix container server
- Artix container service
- Artix service plug-in
- Artix deployment descriptor

- Artix container administration client
- WSDL contract

How it works

Figure 3 shows an simple overview of how the main Artix container components interact. Some user-defined service plug-ins are deployed into an Artix container server, along with an Artix container service.

When the Artix container service is running, you can then use a container administration client to communicate with it at runtime. This client enables you to deploy and manage your services dynamically.

An Artix container service can run inside any Artix bus. Because it is implemented as an Artix plug-in, it can be loaded into any application. The recommended approach is to deploy it into an Artix container server, as shown in Figure 3.

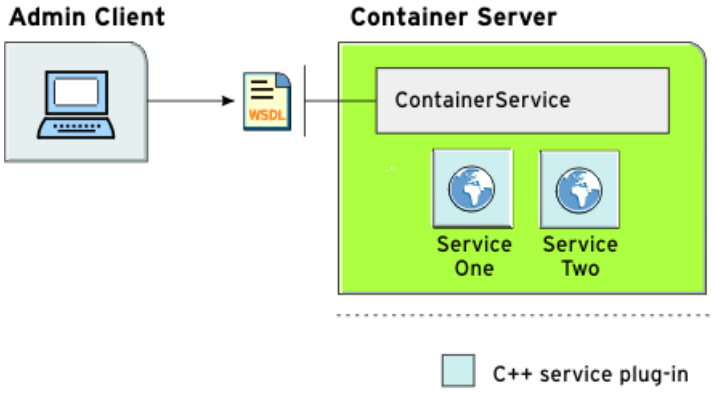


Figure 3: *Artix Container Architecture*

Artix container server

An Artix container server is a simple Artix application that hosts the container service. It consists of a server mainline that initializes a bus and loads the Artix container service, which enables you to remotely deploy and manage your services.

You can run an Artix container server using the `it_container` command. If your application requires some configuration, you can start an Artix container server with a configuration scope. For more details, see "Running an Artix Container Server" on page 82.

Artix deployment descriptor

When deploying a user-defined service into an Artix container, you must pass in a generated Artix deployment descriptor. This is a simple XML file that specifies the details such as:

- Service name.
- Plug-in that implements the service.

You can generate a C++ deployment descriptor by using Artix code generation commands. For more details, see "Generating a Plug-in and Deployment Descriptor" on page 79.

Artix container service

The Artix container service is a remote interface that supports the following operations:

- List all services in the application.
- Stop a running service.
- Start a dormant service.
- Remove a service.
- Deploy a new service.
- Get an endpoint reference for a service.
- Get the WSDL for a service.
- Get the URL to a service's WSDL.
- Shut down the container service.

When an Artix container service deploys a new service, it loads the appropriate plug-ins, sets up and activates your service.

The Artix container service assumes that the plug-ins are available in your application environment, so you must ensure that they are in the expected library path. The Artix container service supports C++ applications, provided that they are compiled into plug-ins.

The Artix container service has a WSDL-based interface and so can be used with any binding or transport.

Artix container administration client

Because the Artix container service has a WSDL-based interface with a SOAP/HTTP binding, you can communicate with it using any client. Artix provides a command-line tool that uses the Artix container stub code, and which enables you to manage the container service easily. The Artix container administration client currently supports SOAP/HTTP only.

You can run an Artix container administration client using the `it_container_admin` command. This client makes all the container service operations available through simple command-line options. For more details, see "Running an Artix Container Administration Client" on page 85.

Multiple Artix services and containers

You can deploy single or multiple Artix services in a single Artix container. How many containers you should have depends on the needs of your system. In general, it is recommended that you deploy services that need to co-exist into the same container. Otherwise, you should partition your services into different Artix containers.

Artix container demos

The following demos in your Artix installation show use of the Artix container:

- `...\samples\advanced\container\deploy_plugin`

  This shows how starting with a `.wsdl` file, you can use the `wsdltocpp` command-line tool to generate a C++ plug-in and deployment descriptor. It then shows how to deploy the plug-in into the Artix container.

- `...\samples\advanced\container\deploy_routes`

  This shows how routes are simply advanced services that happen to be implemented by the router plug-in, and whose implementation is just a proxy to a different service. It shows how you can dynamically deploy and manage routes in the Artix container.
- `...\samples\advanced\container\secure_container`

  This shows how to run a container server in a secure mode with client authentication and authorization. It shows how to restart a service in secure mode, and how to shutdown a container by requesting a user name and password from a console. For details of securing a container, see the ***Artix Security Guide***.

Several other advanced Artix demos also use the Artix container, for example:

- `...\samples\advanced\locator`
- `...\samples\advanced\session_management`
- `...\samples\routing`

Generating a Plug-in and Deployment Descriptor

Artix services are implemented by C++ plug-ins. When you want to deploy a service into an Artix container, the first step is to generate a plug-in from a WSDL contract.

This generates a dynamic library (Windows), or shared library (UNIX), and a dependencies file. An XML deployment descriptor is also generated for the service. You can generate a plug-in and deployment descriptor using any of the following commands:

- `wsdltocpp`
- `wsdd`

Using wsdltocpp

For example, to generate a C++ plug-in library and a deployment descriptor for a specified `.wsdl` file, use the following command:

```
wsdltocpp -n deploy_plugin -impl -server -m NMAKE:library
   -plugin:it_simple_service_cpp_bus_plugin -deployable
   simple_service.wsdl
```

The `-plugin` and `-deployable` options are the most important. `-plugin` generates a new plug-in, and `-deployable` generates a corresponding deployment descriptor.

The generated plug-in can have an optional name (in this case, `it_simple_service_cpp_bus_plugin`). If a name is specified, the generated plug-in library uses this name. The name is ignored if the `.wsdl` file contains more than one service definition. If no plug-in name is set or ignored, the plug-in name takes the following format: *`ServiceNamePortTypeName`*.

In this example, `-impl` generates the skeleton code for implementing the server defined by the WSDL. `-server` generates code for a server sample implementation, and `-m` generates a makefile.

Note: You specify `all` as the make target; the default target does not generate the dependencies file (`.dps`).

For full details on using the `wsdltocpp` command, see the ***Artix Command Line Reference***, or ***Developing Artix Applications in C++***.

C++ deployment descriptor

The deployment descriptor generated for the example C++ service is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<m1:deploymentDescriptor xmlns:m1="http://schemas.iona.com/deploy">
   <service xmlns:servicens

   ="http://www.iona.com/bus/tests">servicens:SimpleServiceService</ser
   vice>
   <plugin>
      <name>it_simple_service_cpp_bus_plugin</name>
      <type>Cxx</type>
   </plugin>
</m1:deploymentDescriptor>
```

The `type` element tells the Artix container that this is a C++ service.

Using wsdd

For more complex deployment descriptors, you can use the Web services deployment descriptor (`wsdd`) command as an alternative to `wsdltocpp`.

The descriptors generated by `wsdltocpp` do not include all the possible information that descriptors can have—for example, `provider_namespace` (see the `advanced/container/deploy_routes` demo).

The following example uses the `wsdd` command:

```
wsdd -service {http://www.iona.com/test}CustomService
   -pluginName testplugin -pluginType Cxx
```

The full syntax of the `wsdd` command is as follows:

```
wsdd -service QName -pluginName PluginName -pluginType Cxx
   [-pluginImpl Library/ClassName ] [-pluginURL url]
   [-wsdlurl WsdlLocation] [-provider ProviderNamespace]
   [-file OutputFile] [-d OutputDir] [-h] [-v] [-verbose]
   [-quiet]
```

The following arguments are required:

Table 12: *Required Arguments to wsdd*

`-service` *`QName`*	Specifies the name of a service to be deployed.
`-pluginName` *`PluginName`*	Specifies the name that a plug-in is registered as.
`-pluginType Cxx`	Specifies the plug-in type.

The following arguments are optional:

Table 13: *Optional Arguments to wsdd*

`-pluginImpl` *`Library/ClassName`*	Specifies a library name (`.dll`/`.so`) for a C++ plug-in.
`-pluginURL` *`url`*	Specifies the location where plug-in library/classes are located. This option, if specified, has no effect on deployment.
`-wsdlurl` *`WsdlLocation`*	Specifies a URL to a service WSDL.
`-provider` *`ProviderNamespace`*	Specifies the provider namespace. Used in the `container/deploy_routes` demo. For example, this can be used by plug-ins to provide servant implementations for more than one service.
`-file` *`OutputFile`*	Specifies the name of the generated descriptor file. The default is `deploy`*`serviceLocalName`*. For example, if `-service {http://www.iona.com/test}CustomService` is used, it is `deployCustomService.xml`
`-d` *`OutputDir`*	The location where a descriptor should be generated.
`-h[elp]`	Displays detailed help information for each option.
`-v[ersion]`	Displays the version of the tool.
`-verbose`	Displays output in verbose mode.
`-quiet`	Displays output in quiet mode.

Adding business logic

For C++ applications, you must still add your business logic code to the servant implementation class.

The supplied Artix demos include a fully implemented servant file instead of the generated file.

Artix deployment descriptors

As well as hosting user-defined services, an Artix container can be used to host Artix services such as the locator. The following is an example generated deployment descriptor for the locator service:

```
<?xml version="1.0" encoding="utf-8"?>
<m1:deploymentDescriptor xmlns:m1="http://schemas.iona.com/deploy">
    <service
   xmlns:servicens="http://ws.iona.com/2005/11/locator">servicens:LocatorService</service>
    <plugin>
        <name>it_service_locator</name>
        <type>Cxx</type>
    </plugin>
</m1:deploymentDescriptor>
```

For details on deploying a locator in the container, see the ***Artix Locator Guide***.

Running an Artix Container Server

An Artix container server is an Artix server mainline that initializes an Artix bus, and loads an Artix container service.

As well as hosting your own service plug-ins, the Artix container server can also be used to host Artix services, such as the locator, session manager, router, and so on. You can run as many instances of the Artix container server as your applications require.

This section explains how to run an Artix container server process using the `it_container` command.

it_container command

To run an Artix container server, use the `it_container` command. This has the following syntax:

```
it_container [-s[ervice] Options]  [-d[aemon]] [-p[ort]
   PortNumber] [-publish [-file Filename]] [-deploy Descriptor]
   [-deployfolder ] [-env Name=Value]  [-policy Descriptor]
   [-v[ersion]] [-h[elp]]
```

`-s[ervice]`	On Windows, runs the container server as a Windows service. Without this parameter, it runs in foreground. See “Running an Artix Container as a Windows Service” on page 93.
`-d[aemon]`	On UNIX, runs the container server as a daemon in the background. Without this parameter, it runs in the foreground.
`-p[ort] PortNumber`	Specifies the port number for the container service. There is no default port number.
`-publish [-file Filename]`	Specifies the location to export the container service URL. By default, this is `/ContainerService.url`. You can override the default using `-file`.
`-deploy Descriptor`	Deploys a service using a specified deployment descriptor (for example, at startup). This is instead of deploying with the container service (see “Using the it_container_admin command” on page 86).
`-deployfolder Path`	Specifies the location of a local folder to store deployment descriptors. This enables redeployment of existing services on restart (see “Deploying Services on Restart” on page 89).
`-env Name=Value`	Specifies arguments passed to the container server process such as environment variables (see “Specifying arguments to the container server” on page 85).
`-policy Descriptor`	Define the set of policies acting on this container service using the specified policy descriptor.
`-v[ersion]`	Prints version information and exits.
`-h[elp]`	Prints usage summary and exits.

Running the container server in the background

On UNIX, to run a container server in the background, use the `it_container -daemon` command.

If the `-daemon` option is not specified, the container server runs in the foreground of the active command window. This option does not apply on Windows.

Publishing the container service URL in a file

To publish a container service URL, use the `-publish` option, for example:

```
it_container -publish -file
   my_directory/my_container_service.url
```

The `-publish` option tells the container server to publish the container service URL in a local file. This URL can then be later retrieved by the `it_container_admin` command, which uses it to contact the container service, and initialize a container service client proxy.

By default, a `ContainerService.url` file is created in the local directory. Use the `-file` option to override this behavior.

Running the container server on a specified port

To run a container server on a specific port, specify the `-port` option, for example:

```
it_container -port 1111
it_container -port 2222
```

This port is used for the container service. This is also the port for the `wsdl_publish` plug-in. The container administrative client uses `wsdl_publish` to get contracts for the container service and for all other services hosted by the container.

This port number can then be used by a container service administration client when contacting the container server, for example:

```
it_container_admin -port 1111
```

Specifying configuration to the container server

You can run `it_container` without any configuration, and this is sufficient for many simple applications. However, if your application requires additional settings, you can start `it_container` with command-line configuration.

For simple applications, the container server loads any plug-ins that you need to instantiate your service, so you do not normally need to configure a plug-ins list, or any other configuration. However, some advanced features may involve launching `it_container` with command-line configuration.

The following example is from the ..samples\advanced\locator demo and shows running the locator service in the container server:

```
it_container -BUSname demo.locator.service
   -BUSdomain_name locator -BUSconfig_domains_dir
   ../../etc -publish -file
   ../../etc/ContainerService.url
```

In this example, the locator service picks up specific configuration from its demo.locator.service scope. For more details, see the demos for the locator, session manager, and router.

Specifying arguments to the container server

You can use the -env option to specify arguments passed to the container server process as follows:

```
it_container –env foo=bar
```

All arguments passed to the container process are set before Bus::init() is called.

For example, you can use the -env option to set environment variables as follows:

```
it_container -env PATH="c:\myApp;%PATH%"
```

You can specify the -env option multiple times to add more than one change to the environment, for example:

```
it_container –env foo=bar –env foo2=bar2 –env foo3=bar3
```

Note: Due to operating system dependent limitations, not all environment variables can be set on all platforms (for example, LD_LIBRARY_PATH on Solaris).

See also "Installing a container as a Windows service" on page 94

Running an Artix Container Administration Client

This section explains how to use the Artix container administration client to perform tasks such as deploying a generated plug-in into the Artix container server, and retrieving a service URL. It explains the full syntax of the it_container_admin command, which is used to control the Artix container administration client.

Using the it_container_admin command

The full syntax for the `it_container_admin` command is as follows:

`-deploy -file` *`dd.xml`*	Deploys a new service into the container server. This involves loading a plug-in that contains the service implementation. You must specify an Artix deployment descriptor using the `-file` option.
`-listservices`	Displays all services in the application. Shows the state of each service (for example, initialized, activated, de-activated, or shutting down).
`-startservice -service {`*`Namespace}LocalPart`*	Restarts the specified service that is visible but dormant, or that has been previously stopped.
`-stopservice -service {`*`Namespace}LocalPart`*	Stops the specified running service.
`-removeservice -service {`*`Namespace}LocalPart`*	Removes and undeploys all trace of the specified service from the application.
`-publishreference -service {`*`Namespace}LocalPart`* `[-file` *`Filename]`*	Gets an endpoint reference for the specified service. The `-file` option publishes the reference to a local file. This can then be used to initialize a client application.
`-publishwsdl -service {`*`Namespace}LocalPart`* `[-file` *`Filename]`*	Gets the WSDL for the specified service. The `-file` option publishes the WSDL to a local file. This can then be used to initialize a client application.
`-publishurl -service {`*`Namespace}LocalPart`* `[-file` *`Filename`*`]`	Gets an HTTP URL for the specified service from which you can then download the WSDL. The `-file` option publishes the URL to a local file. This can then be used to initialize a client application.
`-shutdown [-soft]`	Shuts down the entire application. The `-soft` option shuts down gracefully.
`-port` *`ContainerPort`*	Contacts the container server on the specified port. There is no default container port. See "Running the container server on a specified port" on page 84. This can be used with other options instead of `-container`.
`-host` *`ContainerHostname`*	Contacts the container server on the specified host. Defaults to localhost if unspecified. The `-host` option is for use with `-port` only.
`-container` *`File.url`*	Runs the specified container service. This can be used with other options instead of `-port` and `-host`.
`-getservicepolicy -service <{`*`Namespace`*`}`*`LocalPart`*`>`	Retrieves the set of policies applied to the specified service.

`-getlogginglevel [-subsystem SubSystem] [-service {Namespace}LocalPart]`	Gets the dynamic logging level for the specified subsystem or service. See "Dynamic Artix Logging" on page 33.
`-setlogginglevel -subsystem SubSystem -level Level [-propagate] [-service {Namespace}Localpart]`	Sets the logging level for a specified subsystem of a specified service. See "Dynamic Artix Logging" on page 33.

Note: By default, `it_container_admin` looks in the local directory for the `ContainerService.url` file. If this file is not local, use the `-container` option, or the `-port` and `-host` options, to contact the container.

Deploying the generated plug-in

To deploy a generated plug-in into the container server, use the `-deploy` option, for example:

```
it_container_admin -deploy -file
   ../plugin/deploySimpleServiceService.xml
```

The `-file` option specifies a generated deployment descriptor. This lists the service that this plug-in can provide, the plug-in name, and plug-in type. In this example, the portable C++ plug-in library name is expected to be the same as the plug-in name. The library is expected to be located in the `../plugin` directory.

When a container service loads the plug-in, it registers a servant for the service that is described in the deployment descriptor.

Getting service WSDL

To get the WSDL for a deployed service from the container, use the `-publishwsdl` option, for example:

```
it_container_admin -publishwsdl -service
   {http://www.iona.com/bus/demos}WellWisherService -file
   my_service
```

The `-publishurl` option gets the service's WSDL contract. The `-file` option publishes the WSDL to a local file. When the client runs, it reads the published WSDL from the local file, and uses it to initialize a client stub, and communicate with a deployed service.

Using the `-publishreference`, `-publishwsdl`, and `-publishurl` options means that you can write WSDL contracts without hard-coded ports, and that your clients will still be able to call against them.

Getting a service URL

To get a URL for a deployed service from the container service, use the `-publishurl` option, for example:

```
it_container_admin -publishurl -service
   {http://www.iona.com/bus/tests}SimpleServiceService -file
   my_service
```

The `-publishurl` option gets a URL to the service's WSDL contract. The `-file` option publishes the URL to a local file. When the client runs, it reads the published WSDL URL from the local file, and uses it to initialize a client stub, and then communicate with a deployed service.

Listing deployed services

To display a list of the services in your application, use the `-listservices` option, for example:

```
it_container_admin -port 2222 -listservices
{http://www.iona.com/demos/wellwisher}WellWisherService
   ACTIVATED
{http://www.iona.com/demos/greeter}GreeterService ACTIVATED
```

This example shows the output listed under the `it_container_admin -listservices` command. The `ACTIVATED` service state indicates that a service is running and accepting requests. In this example, the `-port` option is used to contact a container server that was already started on port `2222`.

Service states

The possible service states are as follows:

`NOT_INITIALIZED`	Service has not yet initialized an implementation object or work queue.
`INITIALIZED`	A transient service state. A service remaining in this state indicates that activation failed, and the service was not removed from the bus.
`ACTIVATED`	Service implementation object and work queue created, listener accepting requests.
`DEACTIVATED`	Service not accepting requests, but still in memory, and can return to `ACTIVATED` state.
`SHUTDOWN_PENDING`	Service waiting to complete any pending requests, but stopped accepting new requests.
`SHUTDOWN_COMPLETE`	Service work queue stopped, and unloaded from memory.

Stopping deployed services

To stop a currently deployed service, use the `-stopservice` option, for example:

```
it_container_admin -port 2222 -stopservice -service
   {http://www.iona.com/demos/wellwisher}WellWisherService
```

This following example shows the output from `-listservices` after the service has been stopped.

```
it_container_admin -port 2222 -listservices
{http://www.iona.com/demos/wellwisher}WellWisherService
   DEACTIVATED
{http://www.iona.com/demos/greeter}GreeterService ACTIVATED
```

The `WellWisherService` is now listed as `DEACTIVATED`.

Specifying configuration to the administration client

You can run `it_container_admin` without any configuration. This is sufficient for most simple applications. However, if your application requires additional settings, you can start `it_container_admin` with command-line configuration.

For simple applications, the container service loads any plug-ins that you need to instantiate your service, so you do not normally need to configure a plug-ins list, or any other configuration. However, some advanced features may involve launching `it_container_admin` with command-line configuration.

The following example shows shutting down the locator service using the `it_container_admin -shutdown` option:

```
it_container_admin -BUSdomain_name locator
   -BUSconfig_domains_dir ../../etc -container
   ../../etc/ContainerService.url -shutdown
```

For more details, see the demos for the locator, session manager, and router.

Deploying Services on Restart

The Artix container can be configured to retain information about the service plug-ins that it has deployed. This enables it to reload services automatically on restart. This ability to remember deployed services is known as *persistent deployment*.

To enable persistent deployment, you must configure the container to use a local folder to store deployment descriptors. These descriptors specify what the container should deploy at startup. The container ensures that this folder accurately reflects what is deployed in case of a restart.

How it works

To reload services that have been deployed by the container service before shutdown, the container persists all deployment descriptors when processing new deployment requests. The container needs to know the location of a local folder where deployment descriptor files are saved to, and where to read them from on restart.

The container finds the location of this folder from either:

- A command-line argument passed to the container.
- A configuration variable in an `artix.cfg` file.

Note: The command-line arguments take precedence over the configuration variables.

At startup, the container looks in the configured deployment folder and deploys the contents of the folder. It deploys all services that it finds in the folder where possible. If any deployment fails, the container fails to start.

Persistent deployment modes

You can configure the deployment descriptor folder for either read/write or read-only deployment.

Dynamic read/write deployment

In this case, the container adds and removes files from the deployment folder dynamically as services are deployed or removed from the container. When a call to deploy a service is made, a descriptor file is added to the folder. When a call to remove a service is made, a descriptor file is removed, and the service is not redeployed upon restart.

Read-only deployment

The deployment descriptor folder can also be used as a read-only initialization folder that predeploys the same required set of services after every restart.

When a deployment folder is read-only, the container predeploys the same set of services on restart. No deployment descriptors are removed from, or saved into, a read only deployment folder by the container.

By making a deployment folder read-only, you can share deployment descriptors between multiple container instances. In this scenario, you can enable a single container instance to modify the contents of this folder, and all container instances are affected after restart.

Enabling dynamic read/write deployment

You can enable a read/write deployment folder using the following command-line arguments:

```
it_container -deployfolder ../etc
```

Alternatively, you can set the following variable in a configuration file:

```
plugins:container:deployfolder="../etc";
```

This means that the `../etc` folder is used for predeploying services and persisting new descriptors.

Enabling read-only deployment

You can enable a read-only deployment folder using the following command-line arguments:

```
it_container -deployfolder -readonly ../etc
```

Alternatively, you can set the following variables in a configuration file:

```
plugins:container:deployfolder="../etc";
plugins:container:deployfolder:readonly="true";
```

This means that the `../etc` folder is used for predeploying services only.

Predeploying a service on startup

The `it_container` command also provides a `-deploy` argument, which can be used to predeploy a single service on startup, for example:

```
it_container -deploy deployCORBAService.xml
```

The `-deploy` and `-deployfolder` arguments can be used together, for example:

```
it_container -deploy deployMyService.xml -deployfolder
   ../etc
```

This means that `MyService` identified by `deployMyService.xml`, and all services identified by descriptors in the `../etc` folder, are deployed. The `deployMyService.xml` that is specified using the `-deploy` argument is not copied into a deployment folder. If you wish to copy a descriptor to the deployment folder, use the following command:

```
it_container_admin -deploy -file deployMyService.xml
   -deployfolder -deployfolder ../etc
```

Naming conventions

The Artix container uses the following format when persisting deployment descriptors into files:

```
deployLocalServiceName.xml
```

You should follow the same pattern when generating custom descriptors where possible. The container expects that all files in the deployment folder that have the `.xml` extension are valid deployment descriptors.

By default, deployment descriptors generated by Artix tools use the name of the service's local part. If you have two services with the same local part but different namespaces, you should use the `wsdd -file` option to avoid the name clashing. For more details, see "Using wsdd" on page 80.

Removing a service

When using a read/write deployment folder, you can remove a service by calling `it_container_admin -removeservice` on a running container. For example:

```
it_container_admin -removeservice -service
   {http://www.iona.com/bus/tests}SimpleServiceService
```

Alternatively, you can remove the deployment descriptor file from the folder. Both of these approaches ensure that the container does not reload the service at startup.

When using a read-only folder, removing a service using `-removeservice` does not prevent it from being redeployed after a restart. Only removing a descriptor file from the folder prevents it from being redeployed.

Note: Copying or removing files from the deployment folder has no impact if the container is already running. The container cannot react to these events. The contents of the folder is read once at startup. This only applies to services that are started using deployment descriptors.

Warnings and exceptions

It is possible that using different descriptors might lead to the container attempting to deploy the same service twice.

In this case, the container logs a warning message and proceeds with deploying other services. An exception is thrown if an attempt to deploy the same service is made from an administration console.

Further information

For a working example of persistent deployment, see the following Artix demo:

`.../samples/advanced/container/deploy_plugin`

Running an Artix Container as a Windows Service

On Windows, you can install instances of an Artix container server as a Windows service. By default, this means that the installed container will start up when your system restarts.

This feature also enables you to manage the container using the Windows service controls. For example, you can start or stop a container using the Windows Control Panel, or Windows `net` commands, such as `net stop` *`ServiceName`*.

Format of service names

When a container is installed as a Windows service, the container name takes the following format in the Windows registry:

```
ITArtixContainer ServiceName
```

For example, if you call your service `test_service`, the name generated by the install command that appears in the registry is:

```
ITArtixContainer test_service
```

This name is stored under the following entry in the registry:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services
```

Setting your environment variables

Before installing the Artix container as a Windows service, you must ensure that your system environment variables have been set correctly, and that your machine has rebooted. These steps can be performed either when installing Artix, or at any time prior to installing the container as a Windows service.

Your environment variables enable the container to find all the information it needs on restart. They must be set as follows:

Environment Variable	Setting
`IT_PRODUCT_DIR`	Your Artix installation directory (for example, `c:\artix`). **Note:** This is needed only if your `PATH` specifies `%IT_PRODUCT_DIR%`, instead of the full path to any Artix directories.

Environment Variable	Setting
PATH	Should include the following: • Any C++ plug-ins that will be deployed by the container. • *ArtixInstallDir*\bin.

Installing a container as a Windows service

To install a container as a Windows service, use the `it_container -service install` command:

```
it_container -service install [-BUSParamName [ParamValue]]
   -displayname Name -svcName ServiceName
```

These parameters are described as follows:

`-BUSParamName` — Represents zero or more `-BUSParamName` command-line options (for example, `-BUSlicense_file`). These specify the location of the Artix license file, domain name, configuration directory, or Artix bus name.

These values must be specified either as command-line parameters or environment variables. However, specifying on the command line allows easier deployment of multiple `it_container` instances as multiple Windows services.

`-displayname` — Specifies the name that is displayed in the Windows **Services** dialog (select **Start**|**Settings**|**Control Panel**|**Application Tools**|**Services**). The `-displayname` parameter is required.

`-svcName` — Specifies the service name that is listed in the Windows registry (select **Start**|**Run**, and type `regedit`). The `-svcName` parameter is required.

In addition to these `-service install` parameters, the following `it_container` parameters also apply:

`-port` — Specifies the port that the container will run on (see "Running the container server on a specified port" on page 84). This parameter is required.

`-deployfolder` — Specifies a local folder to store deployment descriptors. This enables redeployment on startup (see "Deploying Services on Restart" on page 89). This parameter is optional.

`-env Name=Value` Specifies arguments passed to the container process, which are also passed to the Windows service command line. For example, `-env PATH="c:\myApp;%PATH%"`. See "Specifying arguments to the container server" on page 85

Example command

The following example shows all the parameters needed to install a container instance as a Windows service:

```
it_container -service install -BUSlicense_file
   c:\InstallDir\etc\licenses.txt -BUSconfig_dir c:\InstallDir\etc
   -BUSdomain_name artix -displayName "My Test Service" -svcName
   my_test_service -port 2222 -deployfolder C:\deployed_files
```

If you do not set your license file, domain name, and configuration directory, as environment variables, you must set them as `-BUS`*`ParamName`* entries (the recommended approach). The `-BUSname` parameter is optional.

You will need administrative privileges on the machine in order to install services.

Example service

The installed Windows service is listed in the **Services** dialog, as shown in Figure 4.

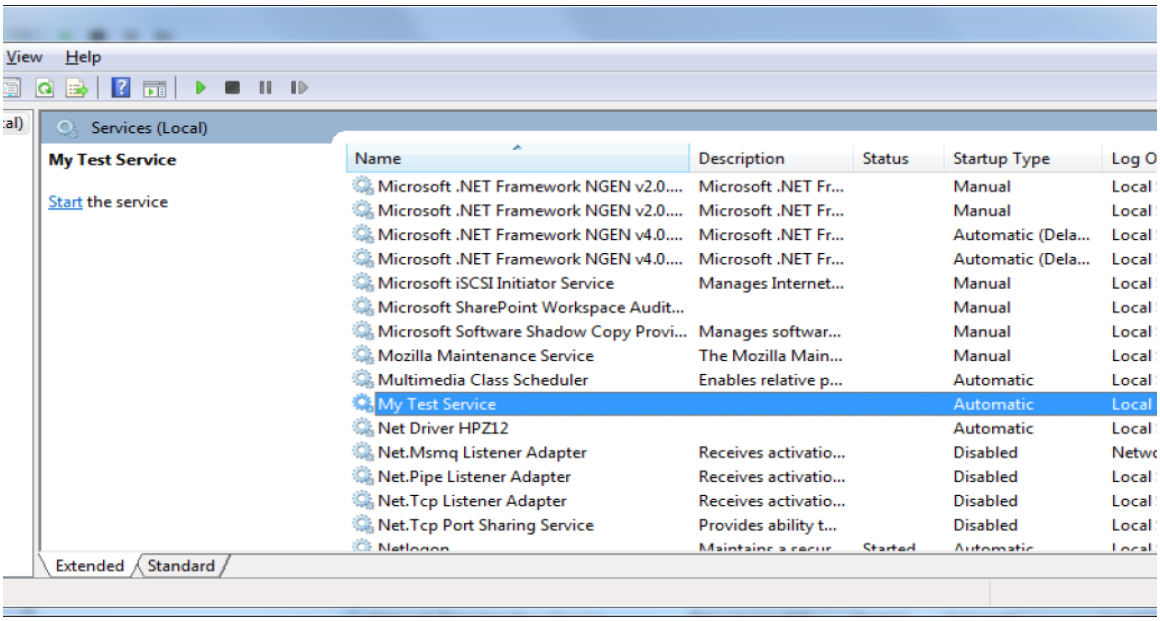


Figure 4: *Installed Windows Service*

Double-clicking on `My Test Service` displays the properties shown in Figure 5.

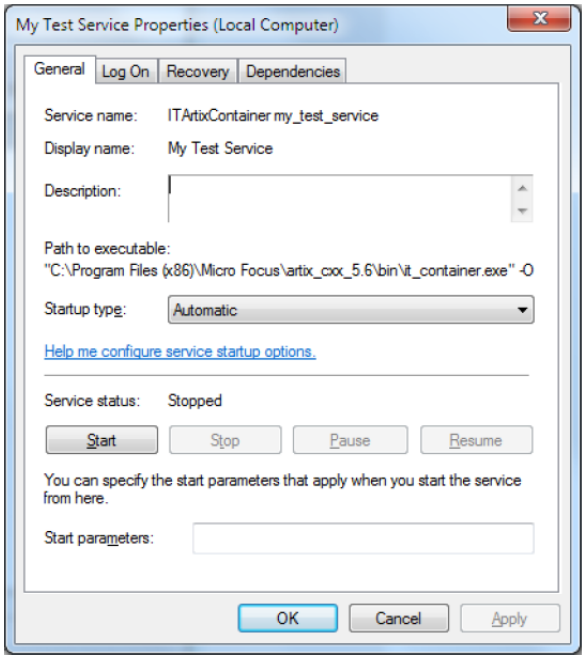


Figure 5: *Service Properties*

After running the `it_container -service install` command, you must start the services manually. However, when your computer is restarted, the installed services are configured to restart automatically.

Uninstalling a container

To uninstall a container as a Windows service, use the `it_container uninstall` command.

```
it_container -service uninstall -svcName ServiceName
```

For example:

```
it_container -service uninstall -svcName my_test_service
```

You will need administrative privileges on the machine in order to uninstall services.

Debugging Plug-ins Deployed in a Container

When developing and testing Artix plug-ins, you may need to debug your plug-in code while it runs in the Artix container. This section explains how to debug C++ plug-ins deployed in an Artix container.

Debugging Artix C++ plug-ins

The easiest option is to create an empty project in your development environment (for example, Microsoft Visual Studio), and set up a debug session. To debug an Artix C++ plug-in, perform the following steps:

1. Start your development tool from an environment that is initialized for Artix (for example, a shell that has already run the `artix_env` script).
2. When configuring the debug session, provide the same details for the executable and parameters as when starting the Artix container from command line. Figure 6 shows a Visual Studio example based on the Artix `advanced/container/deploy_plugin` sample.

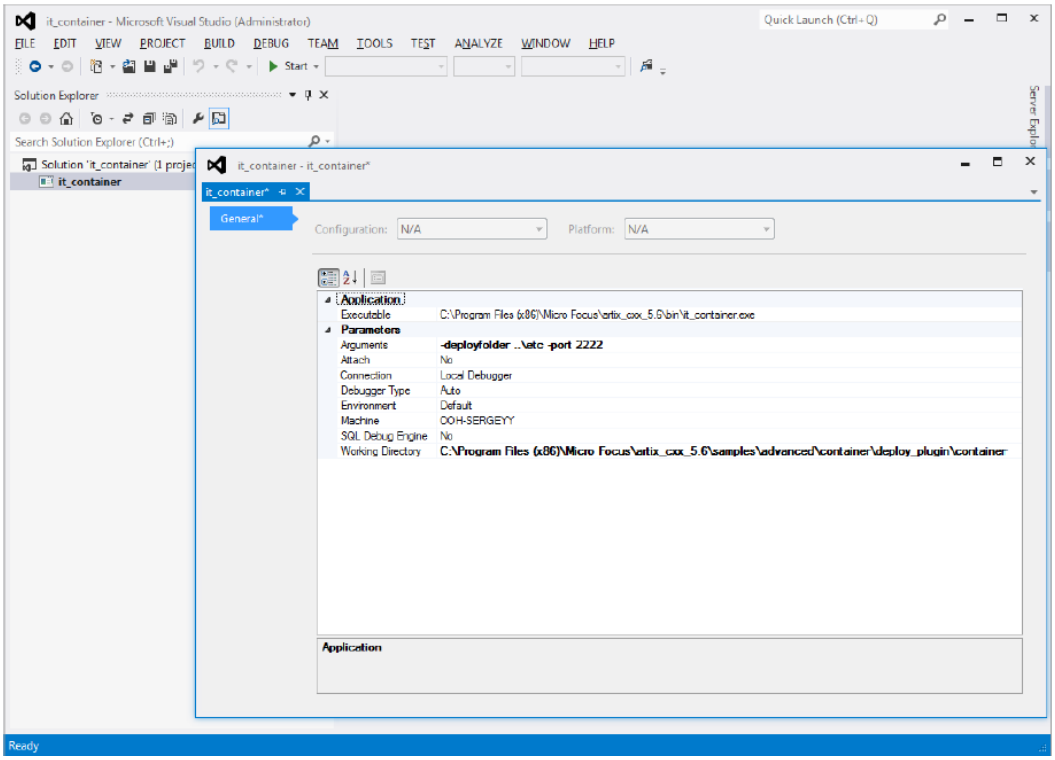


Figure 6: *Visual Studio debug settings*

3. Load the application plug-in source code into your development environment, and set the breakpoints accordingly.

Deploying an Artix Transformer

Artix provides an XSLT transformer service that can be configured to run as a servant process that replaces an Artix server.

The Artix Transformer

The Artix transformer provides a means of processing messages without writing application code. The transformer processes messages based on XSLT scripts and returns the result to the requesting application. XSLT stands for *Extensible Stylesheet Language Transformations*.

These XLST scripts can perform message transformations, such as concatenating two string fields, reordering the fields of a complex type, and truncating values to a given number of decimal places. XSLT scripts can also be used to validate data before passing it onto a Web service for processing, and a number of other applications.

Deployment Patterns

The Artix transformer is implemented as an Artix plug-in. Therefore, it can be loaded into any Artix process. This makes it extremely flexible in how it can be deployed in your environment. If the speed of calls or security is an issue, the transformer can be loaded directly into an application. If you need to spread resources across a number of machines, the transformer plug-in can be loaded in a separate process.

There are two main patterns for deploying the Artix transformer:

- Standalone deployment
- Deployment as part of a chain

Standalone deployment

The first pattern is to deploy the transformer by itself. This is useful if your application is doing basic data manipulation that can be described in an XSLT script. The transformer replaces the server process and saves you the cost of developing server application code. This style of deployment can also be useful for performing data validation before passing requests to a server for processing.

The most straightforward way to deploy the transformer is to deploy it as a separate servant process hosted by the Artix container server. When deployed in this way the transformer receives requests from a client, processes the message based on supplied XSLT scripts, and replies with the results of the script. In this configuration, shown Figure 8, the transformer becomes the server process in the Artix solution.

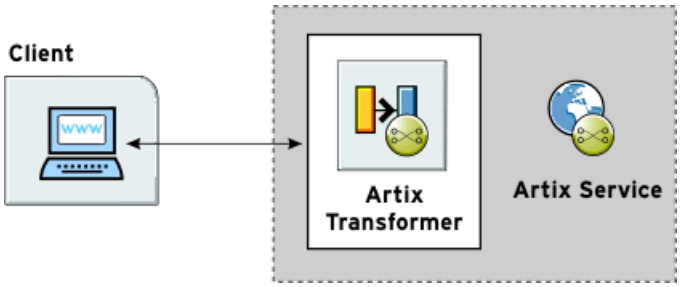


Figure 7: *Artix Transformer Deployed as a Servant*

You can modify the deployment pattern shown in Figure 8 by eliminating the Artix container server and having your client directly load the transformer's plug-in as shown in Figure 8. This saves the overhead of making calls outside of the client process to reach the transformer However, it can reduce the overall efficiency of your system if the transformer requires a large amount of resources to perform its work.

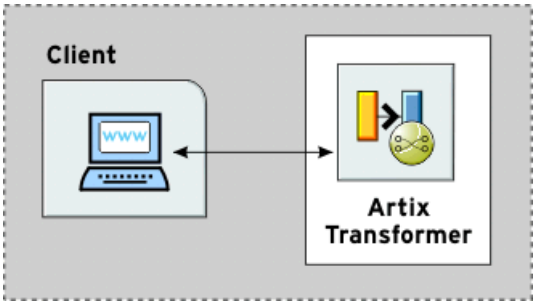


Figure 8: *Artix Transformer Loaded by a Client*

Deployment as part of a chain

The second pattern is to deploy the Artix transformer as part of a Web service chain controlled by the Web service chain builder. This deployment is useful if you need to connect legacy clients to updated servers whose interfaces may have changed or are connecting applications that have different interfaces. It can also be useful for a range of applications where data transformation is needed as part of a larger set of business logic.

Figure 9 shows an example of this type of deployment where the transformer and the chain builder are both hosted by the Artix container server. The chain builder directs the requests to the transformer which transforms messages. When the transformer returns the processed data, the chain builder then passes it onto the server. In this example, the server returns the results to the client without further processing, but the results can also be passed back through the transformer. Neither the client nor the server need to be aware of the processing.

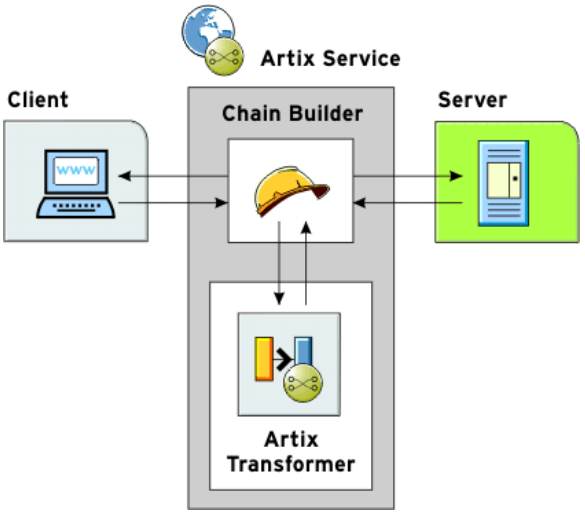


Figure 9: *Artix Transformer Deployed with the Chain Builder*

You could modify this deployment pattern in a number of ways, depending on how you allocate resources. For example, you can configure the client process to load the chain builder and the transformer. You can also load the chain builder and the transformer into separate processes.

Standalone Deployment

To deploy an instance of the Artix transformer you must first decide what process is hosting the transformer's plug-in. You must then add the following to the process configuration scope:

- The transformer plug-in, `xslt`.
- An Artix endpoint configuration to represent the transformer.
- The transformer's configuration information.

Updating the orb_plugins list

Configuring the application to load the transformer requires adding it to the application's `orb_plugins` list. The plug-in name for the transformer is `xslt`. Example 13 shows an `orb_plugins` list for a process hosting the transformer.

Example 13: *Plug-in List for Using XSLT*

```
orb_plugins={"xslt", "xml_log_stream"};
```

Adding an Artix endpoint definition

The transformer is defined as a generic Artix endpoint. To instantiate it as a servant, Artix must know the following details:

- The location of the Artix contract that defines the transformer's endpoint.
- The interface that the endpoint implements.
- The physical details of its instantiation.

This information is configured using the configuration variables in the `artix:endpoint` namespace. These variables are described in Table 14.

Table 14: *Artix Endpoint Configuration*

Variable	Function
`artix:endpoint:endpoint_list`	Specifies a list of the endpoints and their names for the current configuration scope.
`artix:endpoint:`*`endpoint_name`*`:wsdl_location`	Specifies the location of the contract describing this endpoint.

`artix:endpoint:`*`endpoint_name`*`:wsdl_port`	Specifies the port that this endpoint can be contacted on. Use the following syntax:
	`[{`*`service_qname`*`}]`*`service_name`*`[/`*`port_name`*`]`
	For example:
	`{http://www.mycorp.com}my_service/my_port`

Configuring the transformer

Configuring the transformer involves two steps that enable it to instantiate itself as a servant process and perform its work.

- Configuring the list of servants.
- Configuring the list of scripts.

Configuring the list of servants

The name of the endpoints that will be brought up as transformer servants is specified in `plugins:xslt:servant_list`. The endpoint identifier is one of the endpoints defined in

`artix:endpoint:endpoint_list` entry. The transformer uses the endpoint's configuration information to instantiate the appropriate servants

Note: `artix:endpoint:endpoint_list` must be specified in the same configuration scope.

Configuring the list of scripts

The list of the XSLT scripts that each servant uses to process requests is specified in `plugins:xslt:`*`endpoint_name`*`:operation_map`. Each endpoint specified in the servant list has a corresponding operation map entry. The operation map is specified as a list using the syntax shown in Example 14.

Example 14: *Operation Map Syntax*

```
plugins:xslt:endpoint_name:operantion_map =
   ["wsdlOp1@filename1" , "wsdlOp2@filename2", ...,
   "wsdlOpN@filenameN"];
```

Each entry in the map specifies a logical operation that is defined in the service's contract by an `operation` element, and the XSLT script to run when a request is made on the operation. You must specify an XSLT script for every operation defined for the endpoint. If you do not, the transformer raises an exception when the unmapped operation is invoked.

Configuration example

Example 15 shows the configuration scope of an Artix application, `transformer`, that loads the Artix Transformer to process messages. The transformer is configured as an Artix endpoint named `hannibal` and the transformer uses the endpoint information to instantiate a servant to handle requests.

Example 15: *Configuration for Using the Artix Transformer*

```
transformer
{
orb_plugins = ["local_log_stream","xslt"];

artix:endpoint:endpoint_list = ["hannibal"];

artix:endpoint:hannibal:wsdl_location = "transformer.wsdl";
artix:endpoint:hannibal:wsdl_port = "{http://transformer.com/xslt}WhiteHat/WhitePort";

plugins:xslt:servant_list=["hannibal"]
plugins:xslt:hannibal:operation_map = ["op1@../script/op1.xsl",
   "op2@../script/op2.xsl", "op3@../script/op3.xsl"]
}
```

Deployment as Part of a Chain

Deploying the Artix Transformer as part of Web service chain allows you to use it as part of an integration solution without needing to necessarily modify your applications. The Artix Web service chain builder facilitates the placement of the transformer into a series of Web service calls managed by Artix.

The plug-in architecture of the transformer and the chain builder allow for you to deploy this type of solution in a variety of ways depending on what is the best fit for your particular solution. The most straightforward way to deploy this type of solution is to deploy both the transformer and the chain builder into the same process. This is the deployment that will be used to outline the steps for configuring the transformer to be deployed as part of a Web service chain. In general, you will need to complete all of the same steps regardless of how you choose to deploy your solution.

Procedure

To deploy the transformer as part of a Web service chain you need to complete the following steps:

1. Modify your process's configuration scope to load the transformer and the chain builder.
2. Configure Artix endpoints for each of the applications that will be part of the chain.
3. Configure an Artix endpoint to represent the transformer.
4. Configure the transformer.
5. Configure the service chain to include the transformer at the appropriate place in the chain.

Updating the orb_plugins list

Configuring the application to load the transformer plug-in and the chain builder plug-in requires adding them to the process's `orb_plugins` list. The plug-in name for the transformer is `xslt` and the plug-in name for the chain builder is `ws_chain`. Example 16 shows an `orb_plugins` list for a process hosting the transformer and the chain builder.

Example 16: *Loading the Artix Transformer as Part of a Chain*

```
orb_plugins={"xslt", "ws_chain", "xml_log_stream"};
```

Configuring the endpoints in the chain

The Artix Web service chain builder uses generic Artix endpoints to represent all of the applications in a chain, including the transformer. Table 14 on page 102 shows the configuration variables used to configure a generic Artix endpoint.

Configuring the transformer

The transformer requires the same configuration information regardless of how it is deployed. You must provide it with the name of the endpoints it will instantiate from the list of endpoints and provide each instantiation with an operation map. For more information about providing this information see "Configuring the transformer" on page 102.

Placing the transformer in the chain

The chain builder instantiates a servant for each endpoint specified in its servant list. Each servant can have a multiple operations. For each operation that will be involved in a Web service chain, you need to specify a list of endpoints and their operations that make up the chain. This list is specified using `plugins:chain:`*`endpoint_name`*`:`*`operation_name`*`:service_chain`.

To include the transformer in one of the chains, you add the appropriate operation and endpoint names for the transformer at the appropriate place in the service chain.

For more information on configuring the chain builder see "Deploying a Service Chain" on page 109.

Configuration example

Example 17 shows a configuration scope that contains configuration information for deploying the transformer as part of a Web service chain.

Example 17: *Configuring the Artix Transformer in a Web Service Chain*

```
transformer
{
  orb_plugins = ["ws_chain", "xslt"];

  event_log:filters = ["*=FATAL+ERROR+WARNING", "IT_XSLT=*"];

  bus:qname_alias:oldClient = "{http://bank.com}ATM";
  bus:initial_contract:url:oldClient = "bank.wsdl";

  bus:qname_alias:newServer = "{http://bank.com}newATM";
  bus:initial_contract:url:newServer = "bank.wsdl";

  artix:endpoint:endpoint_list = ["transformer"];

  artix:endpoint:transformer:wsdl_location = "bank.wsdl";
  artix:endpoint:transformer:wsdl_port =
   "{http://bank.com}transformer/transformer_port";

  plugins:xslt:servant_list = ["transformer"];
  plugins:xslt:transformer:operation_map =
   ["transform@transformer.xsl"];
```

Example 17: *Configuring the Artix Transformer in a Web Service Chain*

```
  plugins:chain:servant_list = ["oldClient"];
  plugins:chain:oldClient:client_operation:service_chain =
   ["transform@transformer", "withdraw@newServer"];
};
```

Note: Even though a list of servants can be specified, only one servant is currently supported in a process.

Optional Configuration

Overview

You can also use the following optional configuration settings:

- "Specifying an XSLT trace filter"
- "Specifying message part element names"

Specifying an XSLT trace filter

You can use the `plugins:xslt:`*`endpoint_name`*`:trace_filter` variable to trace and debug the output of the XSLT engine. For example:

```
plugins:xslt:endpoint_name:trace_filter =
   "INPUT+TEMPLATE+ELEMENT+GENERATE+SELECT";
```

These settings are described as follows:

`INPUT`	Traces the XML input passed to the XSLT engine.
`TEMPLATE`	Traces template matches in the XSLT script.
`ELEMENT`	Traces element generation.
`GENERATE`	Traces generation of text and attributes.
`SELECT`	Traces node selections in the XSLT script.

Specifying message part element names

You can use the `plugins:xslt:`*`endpoint_name`*`:use_element_name` variable to specify whether to use the message part element name or message part name when performing transformations. The default value is `false`, which means to use the message part name.

Using the message part element name matches the behavior of Artix content-based routing. To use the message part element name, specify the following setting:

```
plugins:xslt:endpoint_name:use_element_name = "true";
```

The following WSDL file extract shows an example message part element name and part name:

```
<message name="client_request_message">
  <part element="tns:client_request_type" name="client_request"/>
</message>
```

The following XSL file extract shows the example part element name when this variable is set to `true`:

```
<xsl:template match="client_request_type">
  <xsl:value-of select="first_name"/>
  <xsl:text> </xsl:text>
  <xsl:value-of select="last_name"/>
</xsl:template>
```

If this variable is set to `false`, the part name is used instead (in this case, **`client_request`**).

Deploying a Service Chain

Artix provides a chain builder that enables you to create a series of services to invoke as part of a larger process.

The Artix Chain Builder

The Artix chain builder enables you to link together a series of services into a multi-part process. This is useful if you have processes that require a set order of steps to complete, or if you wish to link together a number of smaller service modules into a complex service.

Chaining services together

For example, you may have four services that you wish to combine to service requests from a single client. You can deploy a service chain like the one shown in Figure 10.

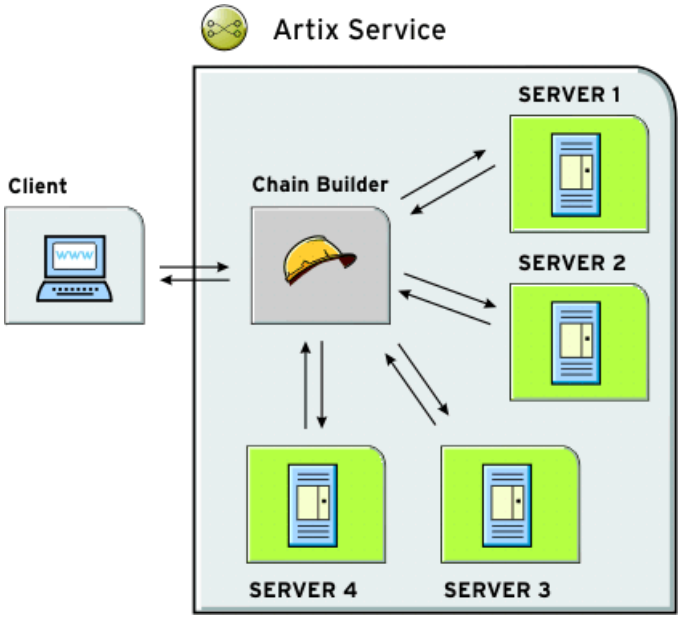


Figure 10: *Chaining Four Servers to Form a Single Service*

In this scenario, the client makes a single request and the chain builder dispatches the request along the chain starting at `Server1`. The chain builder takes the response from `Server1` and passes that to the next endpoint in the chain, `Server2`. This continues until the end of the chain is reached at `Server4`. The chain builder then returns the finished response to the client.

The chain builder is implemented as an Artix plug-in so it can be deployed into any Artix process. The decision about which process that you deploy it in depends on the complexity of your system, and also how you choose to allocate resources for your system.

Assumptions

To make the discussion of deploying the chain builder as straightforward as possible, this chapter assumes that you are deploying it into an instance of the Artix container server. However, the configuration steps for configuring and deploying a chain builder are the same no matter which process you choose to deploy it in.

Configuring the Artix Chain Builder

To configure the Artix chain builder, complete the following steps:

1. Add the chain builder's plug-in to the `orb_plugins` list.
2. Configure all the services that are a part of the chain.
3. Configure the chain so that it knows what servants to instantiate and the service chain for each operation implemented by the servant.

Adding the chain builder in the orb_plugins list

Configuring the application to load the chain builder's plug-in requires adding it to the application's `orb_plugins` list. The plug-in name for the chain builder is `ws_chain`. Example 18 shows an `orb_plugins` list for a process hosting the chain builder.

Example 18: *Plug-in List for Using a Web Service Chain*

```
orb_plugins={"ws_chain", "xml_log_stream"};
```

Configuring the services in the chain

Each service that is a part of the chain, and the client that makes requests through the chain service, must be configured in the chain builder's configuration scope. For example, you must supply the service name and the location of its contract.

This provides the chain builder with the necessary information to instantiate a servant that the client can make requests against. It also supplies the information needed to make calls to the services that make up the chain.

To configure the services in the chain, use the configuration variables in Table 15.

Table 15: *Artix Service Configuration*

Variable	Function
bus:qname_alias:*service*	Specifies a service name using the following syntax: {*service_qname*}*service_name* For example: {http://www.mycorp.com}my_service
bus:initial_contract:url:*service*	Specifies the location of the contract describing this service. The default is the current working directory.

Configuring the service chains

The chain builder requires you to provide the following details

- A list of services that are clients to the chain builder.
- A list of operations that each client can invoke.
- Service chains for each operation that the clients can invoke.

Specifying the servant list

The first configuration setting tells the chain builder how many servants to instantiate, the interfaces that the servants must support, and the physical details of how the servants are contacted. You specify this using the plugins:chain:servant_list variable. This takes a list of service names from the list of Artix services that you defined earlier in the configuration scope.

Specifying the operation list

The second part of the chain builder's configuration is a list of the operations that each client to the chain builder can invoke. You specify this using plugins:chain:*endpoint*:operation_list where *endpoint* refers to one of the endpoints in the chain's service list.

plugins:chain:*endpoint*:operation_list takes a list of the operations that are defined in <operation> tags in the endpoint's contract. You must list all of the operations for the endpoint or an exception will be thrown at runtime. You must also be sure to enter a list of operations for each endpoint specified in the chain's service list.

Specifying the service chain

The third piece of the chain builder's configuration is to specify a service chain for every operation defined in the endpoints listed in plugins:chain:servant_list. This is specified using the plugins:chain:*endpoint*:*operation*:service_chain configuration variable. The syntax for entering the service chains is shown in Example 19.

Example 19: *Entering a Service Chain*

```
plugins:chain:endpoint:operation:service_chain=["op1@endpt1", "op2@endpt2", ...,
   "opN@endptN"];
```

For each entry, the syntax is as follows:

endpoint	Specifies the name of an endpoint from the chain builder's servant list
operation	Specifies one of the operations defined by an operation entry in the endpoints contract. The entries in the list refer to operations implemented by other endpoints defined in the configuration.
opN	Specifies one of the operations defined by an operation entry in the contract defining the service specified by *endptN*. The operations in the service chain are invoked in the order specified. The final result is returned back to the chain builder which then responds to the client.

Instantiating proxy services

The chain invokes on other services, and for this reason, it instantiates proxy services. It can instantiate proxies when the chain servant starts (the default), or later, when a call is made. The following configuration variable specifies to instantiate proxy services when a call is made:

```
plugins:chain:init_on_first_call ="true";
```

This defaults to `false`, which means that proxies are instantiated when the chain servant starts. However, you might not be able to instantiate proxies when the chain servant is started because the servant to call has not started. For example, this applies when using the Artix locator or UDDI.

Configuration example

Example 16 shows the contents of a configuration scope for a process that hosts the chain builder.

Table 16: *Configuration for Hosting the Artix Chain Builder*

```
colaboration {
  orb_plugins = ["ws_chain"];

  bus:qname_alias:customer= "{http://needs.com}POC";
  bus:initial_contract:url:customer = "order.wsdl";

  bus:qname_alias:pm = "{http://ORBSrUs.com}prioritize";
  bus:initial_contract:url:pm = "manager.wsdl";

  bus:qname_alias:designer = "{http://ORBSrUs.com}design";
  bus:initial_contract:url:designer = "designer.wsdl";

  bus:qname_alias:builder = "{http://ORBSrUs.com}produce";
  bus:initial_contract:url:builder = "engineer.wsdl";
```

Table 16: *Configuration for Hosting the Artix Chain Builder*

```
  plugins:chain:servant_list = ["customer"];

  plugins:chain:customer:requestSolution:service_chain =
    ["estimatePriority@pm", "makeSpecification@designer",
     "buildORB@builder"];
};
```

Configuration guidelines

When Web services are chained, the following rules must be obeyed:

- The input type of the chain service (in this example, `customer`) must match the input of the first service in the chain (`pm`).
- The output type of a previous service in the chain must match the input type of the next service in the chain.
- The output type of the last service in the chain must match the output of the chain service.
- One configuration entry must exist for each operation in the `portType` of the chain service (for example, `customer`). This simple example shows only one entry, and the `portType` for the customer endpoint has only one operation (`requestSolution`).
- The chain service can invoke only on services that have one port.
- Finally, not all operations must be configured in the chain, only those that are invoked upon. This means that no check is made when all operations are mapped to a chain. If a client invokes on an unmapped operation, the chain service throws a `FaultException`.

Deploying Artix Services for High Availability

Artix uses Berkeley DB high availability to provide support for replicated services. This chapter explains how to configure and deploy high availability in Artix.

Introduction

Scalable and reliable Artix applications require high availability to avoid any single point of failure in a distributed system. You can protect your system from single points of failure using *replicated services*.

A replicated service is comprised of multiple instances, or *replicas*, of the same service; and together, these act as a single logical service. Clients invoke requests on the replicated service, and Artix routes the requests to one of the member replicas. The routing to a replica is transparent to the client.

How it works

Artix high availability support is built on Berkeley DB, and uses its replication features. Berkeley DB has a master-slave replica model where a single replica is designated the master, and can process both read and write operations from clients. All other replicas are slaves and can only process read operations. Slaves automatically forward write requests to masters, and masters push all updates out to slaves, as shown in Figure 11.

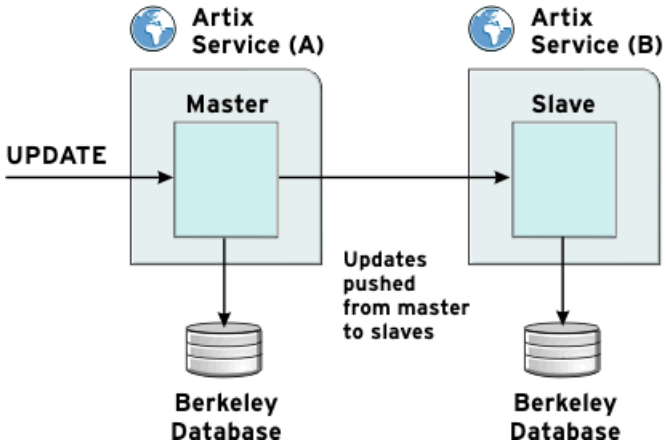


Figure 11: *Artix Master Slave Replication*

Electing a master

Using Artix high availability, when members of a replicated cluster start up, they all start up as slaves. When the cluster members start talking to each other, they hold an election to select a master.

Election protocol

The protocol for selecting a master is as follows:

1. For an election to succeed, a majority of votes must be cast. This means that for a group of three replicas, two replicas must cast votes. For a group of four, three replicas must cast votes; for a group of five, three must cast votes, and so on.
2. If a slave exists with a more up-to-date database than the other slaves, it wins the election.
3. If all the slaves have equivalent databases, the election result is based on the configured priority for each slave. The slave with the highest priority wins.

Note: Because voting is done by majority, it is recommended that high availability clusters have an odd number of members. The recommended minimum number of replicas is three.

After the election

When a master is selected, elections stop. However, if the slaves lose contact with the master, the remaining slaves hold a new election for master. If a slave can not get a majority of votes, nobody is promoted.

At this point, the database remains as a slave, and keeps holding elections until a master can be found. If this is the first time for the database to start up, it blocks until the first election succeeds, and it can create a database environment on disk.

If this is not the first time that the database has started up, it starts as a slave (using the database files already on disk from its previous run), and continues holding elections in the background anyway.

Auto-demotion

In the event of a network partition, by default, the master replica is configured to automatically demote itself to a slave when it loses contact with the replica cluster. This prevents the creation of duplicate masters.

Request forwarding

Slave replicas automatically forward write requests to the master replica in a cluster. Because slaves have read-only access to the underlying Berkeley DB infrastructure, only the master can make updates to the database. This feature works as follows:

1. When a replicated server starts up, it loads the `request_forwarder` plug-in.
2. When the client invokes on the server, the `request_forwarder` plug-in checks if it should forward the operation, and where to forward it to. The server programmer indicates which operations are write operations using an API.
3. If the server is running as a slave, it tries to forward any write operations to the master. If no master is available, an exception is thrown to the client, indicating that the operation cannot be processed.

Because the forwarding works as an interceptor within a plug-in, there is minimal code impact to the user. No servant code is impacted. For details on how to configure request forwarding, see "Specifying your orb_plugins list" on page 119.

Setting up high availability

You can configure all the necessary settings in an `artix.cfg` file (see "Configuring Persistent Services for High Availability" on page 118).

Replication is supported for C++ service development, and by the Artix locator (see "Configuring Locator High Availability" on page 121).

Master and replicas must share same architecture

Master and slave replicas must share the same architecture. For example, if you configure your master service to run on a 32-bit Oracle SPARC server, all replicas must also run on a 32-bit Oracle SPARC server.

Setting up a Persistent Database

To enable a service to take advantage of high availability, it needs to work with a persistent database. This is created using a C++ API. There are no configuration steps required. The Artix configuration variables for persistent databases are set with default values that should not need to be changed.

Using the Persistence API

Artix provides set of C++ APIs for manipulating persistent data. For example, the C++ API uses the `PersistentMap` template class. This class stores data as name value pairs. This API is defined in `it_bus_pdk\persistent_map.h`.

This API enables you to perform tasks such as the following:

- Create a `PersistentMap` database.
- Insert data into a `PersistentMap`.
- Get data from a `PersistentMap`.
- Remove data from a `PersistentMap`.

For more details, see ***Developing Artix Applications in C++***.

Further information

For detailed information on the Berkeley DB database environment, see http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html

Artix ships Berkeley DB 4.2.52. If required, you can download and build Berkeley DB to obtain additional administration tools (for example, `db_dump`, `db_verify`, `db_recover`, `db_stat`).

Configuring Persistent Services for High Availability

For a service to participate in a high availability cluster, it must first be designed to use persistent maps ("Setting up a Persistent Database" on page 117). However, services that use persistent maps are not replicated automatically; you must configure your service to be replicated.

Configuring a service for replication

To replicate a service, you must add a replication list to your configuration, and then add configuration scopes for each replicated instance of your service. Typically, you would create a scope for your replica cluster, and then create sub-scopes for each replica. This avoids duplicating configuration settings that are common to all replicas, and separates the cluster from any other services configured in your domain.

Specifying a replication list

To specify a cluster of replicas, use this configuration variable:

```
plugins:artix:db:replicas
```

This takes a list of replicas specified using the following syntax:

```
ReplicaName=HostName:PortNum
```

For example, the following entry configures a cluster of three replicas spread across machines named `jimi`, `noel`, and `mitch`.

```
plugins:artix:db:replicas=["rep1=jimi:2000", "rep2=mitch:3000",
   "rep3=noel:4000"];
```

Note: It is recommended that you set *ReplicaName* to the same value as the replica's sub-scope (see "Configuration example").

Specifying your orb_plugins list

Because IIOP is used for communication between replicas, you must include the following plug-ins in your replica's `orb_plugins` list:

- `iiop_profile`
- `giop`
- `iiop`

In addition, to enable automatic forwarding of write requests from slave to master replicas, include the `request_forwarder` plug-in. You must also specify this plug-in as a server request interceptor. The following example shows the required configuration:

```
orb_plugins = ["xmlfile_log_stream", "local_log_stream",
   "request_forwarder", "iiop_profile", "giop", "iiop"];

binding:artix:server_request_interceptor_list=
   "request_forwarder";
```

This configuration is loaded when the replica service starts up.

Note: To enable forwarding of write requests, programmers must have already specified in the server code which operations can write to the database. For details, see "Forwarding write requests".

Specifying replica priorities

In each of the sub-scopes for the replicas, you must give each replica a priority, and configure the IIOP connection used by the replicas to conduct elections. This involves the following configuration variables:

`plugins:artix:db:priority`

Specifies the replica priority. The higher the priority the more likely the replica is to be elected as master. You should set this variable if you are using replication.

There is no guarantee that the replica with the highest priority is elected master. The first consideration for electing a master is who has the most current database.

Note: Setting a replica priority to `0` means that the replica is never elected master.

`plugins:artix:db:iiop:port`	Specifies the IIOP port the replica starts on. This entry must match the corresponding entry in the replica list.

Configuration example

The following example shows a simple example in an `artix.cfg` file:

```
ha_cluster{

  plugins:artix:db:replicas = [“rep1=jimi:2000”,
   “rep2=mitch:3000”, “rep3=noel:4000”];

  rep1{
    plugins:artix:db:priority = 80;
    plugins:artix:db:iiop:port = 2000;
  };
  rep2{
    plugins:artix:db:priority = 20;
    plugins:artix:db:iiop:port = 3000;
  };
  rep3{
    plugins:artix:db:priority = 0;
    plugins:artix:db:iiop:port = 4000;
  };
};
```

Configuration guidelines

You should keep the following in mind:

- By default, the DB home directory defaults to *`ReplicaConfigScope`*`_db` (for example, `rep1_db`), where *`ReplicaConfigScope`* is the inner-most replica configuration scope. If this directory does not already exist, it will be created in the current working directory.
- All replicas must be represented by separate WSDL ports in the same WSDL service contract. By default, you should specify the inner-most replica scope as the WSDL port name (for example, `rep1`).

Configuring a minority master

It is recommended that high availability clusters have an odd number of members, and the recommended minimum number is three. However, it is possible to use a cluster with two members if you specify the following configuration:

```
plugins:artix:db:allow_minority_master=true;
```

This allows a lone slave to promote itself if it sees that the master is unavailable. This is only allowed when the replica cluster has two members. This variable defaults to `false` (which means it is

not allowed by default). If it is set to `true`, a slave that cannot reach its partner replica will promote itself to master, even though it only has fifty per cent of the votes (one out of two).

WARNING: This variable must be used with caution. If it is set to `true`, and the two replicas in the cluster become separated due to a network partition, they both end up as master. This can be very problematic because both replicas could make database updates, and resolving those updates later could be very difficult, if not impossible.

Configuring request forward logging

You can also specify to output logging from the `request_forwarder` plug-in.

To do this, specify the following logging subsystem in your event log filter:

```
event_log:filters =
   ["IT_BUS.SERVICE.REQUEST_FORWARDER=INFO_LOW+WARN+ERROR+FATAL
   "];
```

Configuring Locator High Availability

Replicating the locator involves specifying the same configuration that you would use for other Artix services, as described in "Configuring Persistent Services for High Availability" on page 118. However, there are some additional configuration variables that also apply to the locator.

Note: All locator service replicas must be running on the same operating system.

Setting locator persistence

To enable persistence in the locator, set the following variable:

```
plugins:locator:persist_data="true";
```

This specifies whether the locator uses a persistent database to store references. This defaults to `false`, which means that the locator uses an in-memory map to store references.

When replicating the locator, you must set `persist_data` to `true`. If you do not, replication is not enabled.

Setting load balancing

When `persist_data` is set to `true`, the load balancing behavior of the locator changes. By default, the locator uses a round robin method to hand out references to services that are registered with multiple endpoints. Setting `persist_data` to true causes the locator to switch from round robin to random load balancing.

You can change the default behavior of the locator to always use `random` load balancing by setting the following configuration variable:

```
plugins:locator:selection_method = “random”;
```

Configuration example

The following example shows the configuration required for a cluster of three locator replicas.

Example 20: *Settings for Locator High Availability*

```
service {
...
bus:initial_contract:url:locator = "../../../etc/locator.wsdl";

orb_plugins = ["local_log_stream", "wsdl_publish",
   "request_forwarder", "service_locator", "iiop_profile",
   "giop", "iiop"];

binding:artix:server_request_interceptor_list=
   "request_forwarder";

plugins:locator:persist_data = "true";

plugins:artix:db:replicas = ["Locator1=localhost:7876",
   "Locator2=localhost:7877", "Locator3=localhost:7878"];

Locator1{
   plugins:artix:db:priority = "100";
   plugins:artix:db:iiop:port = "7876";
};
Locator2{
   plugins:artix:db:priority = "75";
   plugins:artix:db:iiop:port = "7877";
};
Locator3{
   plugins:artix:db:priority = "0";
   plugins:artix:db:iiop:port = "7878";
};
```

Using multiple locator replica groups

A highly available locator consists of a group of locators, one of which is designated the master. It can process both service lookups from clients and endpoint registrations from servers. All other replica locators are slaves and can only process service lookups.

The locator group is represented by a locator WSDL file that contains multiple endpoints—one for each locator. When the `ha_conf` plug-in is loaded by Artix clients, it uses this WSDL file to resolve and connect to a locator. It tries the first endpoint, and if this does not yield a valid connection, it tries the second endpoint, and so on.

Using the ha_conf plug-in, Artix client applications can failover between locators in the same replica group. However, if you are using two separate replica locator groups, you want your clients to try one group first, and then the other. In this case, you can use one of the following approaches to failover between two separate replica locator groups:

Combine the two groups

You can combine two groups by taking the locator endpoints from the second replica group's WSDL file, and adding them to the list of endpoints in the first replica group's WSDL file. You now have a single WSDL file that contains all the locator endpoints. The ha_conf plug-in will try to contact locators in the order specified in this WSDL file.

Change the configured contract

First, set your Artix configuration so that group1.wsdl is the first replica group's WSDL file, for example:

```
bus:initial_contract:url:locator = "group1.wsdl";
```

Then if a connection cannot be made to any endpoint from this file, change the configured WSDL file to group2.wsdl, re-initialize the bus, and try again.

In this way, by using an extra try/catch statement in the client, you can achieve failover between two replica locator groups.

Further information

For a working example of Artix locator high availability, see the ...advanced/high_availability_locator demo.

Configuring Client-Side High Availability

When you have implemented a highly available service using a group of replica servers, a suitably configured client can talk to the master replica. In the event that the master replica fails, one of the other replicas takes over as master, and the client fails over to one of the other replicas.

As far as the client application logic is concerned, there is no discernible interruption to the service. This section shows how to configure the client to use high availability features. It also explains the impact on the server.

Configuration steps

In most cases, configuring high availability on the client side consists of two steps:

- Create a service contract that specifies the replica group.
- Configure the client to use the high availability service.

Specifying the replica group in your contract

Before your client can contact the replicas in a replica group, you must tell the client how to contact each replica in the group. You can do this by writing the WSDL contract for your service in a particular way.

Example 21 shows the `hello_world.wsdl` contract from the `...\advanced\high_availability_persistent_servers` demo.

Example 21: *Specifying a Replica Group in a Contract*

```
?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="HelloWorld"
   targetNamespace="http://www.iona.com/hello_world_soap_http"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:http-conf="http://schemas.iona.com/transports/http/configuration"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.iona.com/hello_world_soap_http"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <wsdl:types>
        <schema targetNamespace="http://www.iona.com/hello_world_soap_http"
            xmlns="http://www.w3.org/2001/XMLSchema">
            <element name="responseType" type="xsd:boolean"/>
            <element name="requestType" type="xsd:string"/>
            <element name="overwrite_if_needed" type="xsd:boolean"/>
       </schema>
    </wsdl:types>
   ...
   <wsdl:service name="SOAPService">
        <wsdl:port binding="tns:Greeter_SOAPBinding" name="Server1">
            <soap:address location="http://localhost:9551/SOAPService/Server1"/>
        </wsdl:port>
        <wsdl:port binding="tns:Greeter_SOAPBinding" name="Server2">
            <soap:address location="http://localhost:9552/SOAPService/Server2"/>
        </wsdl:port>
        <wsdl:port binding="tns:Greeter_SOAPBinding" name="Server3">
            <soap:address location="http://localhost:9553/SOAPService/Server3"/>
        </wsdl:port>
    </wsdl:service>

</wsdl:definitions>
```

In Example 21, the `SOAPService` service contains three ports, all of the same port type. The contract specifies fixed port numbers for the endpoints. By convention, you should ensure that the first port specified by the service corresponds to the master server.

Configuring the client to use high availability

To configure your client for high availability, perform the following steps:

1. In your client scope, add the high availability plug-in (`ha_conf`) to the `orb_plugins` list. For example:

```
client {
  orb_plugins = [...,"ha_conf"];
};
```

2. Configure the client so that the Artix bus can resolve the service contract. You can do this by specifying the following configuration in the client scope:

```
client {
  bus:qname_alias:soap_service =
   "{http://www.iona.com/hello_world_soap_http}SOAPService";
  bus:initial_contract:url:soap_service = "../../etc/hello_world.wsdl";
};
```

Alternatively, you can also do this using the `-BUSservice_contract` command line parameter as follows:

```
myclient -BUSservice_contract ../../etc/hello_world.wsdl
```

For more details on configuring initial contracts, see "Accessing Contracts and References".

Impact on the server

In Example 21, the contract specifies three separate ports in the same service named `SOAPService`. The implication is that each port is implemented by a different process, and if one of these processes fails, the client switches to one of the others.

Because the servers use the same contract, the server-side code must be written so that the server can be instructed to instantiate a particular port.

Example 22 shows some relevant code. Depending on which argument the server is started with (1, 2, or 3), it instantiates either Server1, Server2 or Server3.

Example 22: *Server Code Chooses which Port to Instantiate*

```
//C++
String cfg_scope = "demos.high_availability_persistent_servers.server.";
String wsdl_url = "../../etc/hello_world.wsdl";
String server_number = argv[1];
String service_name = "SOAPService";
String port_name = "Server";

if (server_number == "1")
{
    cfg_scope += "one";
    port_name += "1";
}
else if (server_number == "2")
{
    cfg_scope += "two";
    port_name += "2";
}
else if (server_number == "3")
{
    cfg_scope += "three";
    port_name += "3";
}

else
{
    cerr << "Error: you must pass 1, 2 or 3 as a command line argument" << endl;
    return -1;
}

    IT_Bus::Bus_var bus = IT_Bus::init(argc, argv, cfg_scope.c_str());

    IT_Bus::QName service_qname(
       "",
       service_name,
      "http://www.iona.com/hello_world_soap_http"
    );
```

Example 22: *Server Code Chooses which Port to Instantiate*

```
GreeterImpl servant(bus, service_qname, port_name, wsdl_url);

    bus->register_servant(
    servant,
    wsdl_url,
    service_qname,
    port_name
);

    cout << "Server Ready" << endl;
    IT_Bus::run();
}
catch (const IT_Bus::Exception& e)
{
   cerr << "Error occurred: " << e.message() << endl;
   return -1;
}
catch (...)
{
cerr << "Unknown exception!" << endl;
return -1;
}
return 0;
```

Server-side state

Client-side failover can be used with both stateful and stateless servers. If your servers are stateful, server-side high availability must be enabled for the servers. This has no impact on the client configuration.

If your servers are stateless, no server-side configuration is necessary. However, your servers can share state using some other mechanism (for example, a shared database). In this case, client-side failover can still be used.

Forwarding write requests

When a client sends a write request to a slave replica, the slave must forward the write request to the master replica. The server programmer must use the `mark_as_write_operations()` method specify which WSDL operations can write to the database.

The C++ function is as follows:

```
// C++
void
mark_as_write_operations(
   const IT_Vector<IT_Bus::String> operations,
   const IT_Bus::QName& service,
   const IT_Bus::String& port,
   const IT_Bus::String& wsdl_url
) IT_THROW_DECL((DBException));
```

For a detailed example, see ***Developing Artix Applications in C++***.

Random endpoint selection for clients

The client-side ha_conf plug-in supports random endpoint selection. This can be very useful if you want your client applications to pick a random server each time they connect.

The random behavior can be applied all the time, so that the client always picks a random server. This approach should be used if you want your clients to be uniformly load-balanced across different servers. To use this approach, set the following configuration:

```
plugins:ha_conf:strategy="random";
plugins:ha_conf:random:selection="always";
```

Alternatively, the random behavior can be applied only after the client loses connectivity with the first server in the list. This approach should be used to make your clients favour a particular server for their initial connectivity. To use this approach, set the following configuration:

```
plugins:ha_conf:strategy="random";
plugins:ha_conf:random:selection="subsequent";
```

Further information

For working examples of high availability in Artix, see the following demos:

- ...advanced/high_availability_persistent_servers
- ...advanced/high_availability_locator

For full details of all database environment and high availability configuration settings, see the ***Artix Configuration Reference, C++ Runtime***.

Deploying WS-Reliable Messaging

Artix supports Web Services Reliable Messaging (WS-RM) for C++ applications. This chapter explains how to deploy WS-RM in an Artix runtime environment.

Introduction

Web Services Reliable Messaging (WS-RM) is a standard protocol that ensures the reliable delivery of messages in a distributed environment. It enables messages to be delivered reliably between distributed applications in the presence of software, system, or network failures.

For example, WS-RM can be used to ensure that the correct messages have been delivered across a network exactly once, and in the correct order. Web Services Reliable Messaging is also known as WS-ReliableMessaging.

How it works

WS-RM ensures the reliable delivery of messages between a source and destination endpoint. The source is the initial sender of the message and the destination is the ultimate receiver, as shown in Figure 12.

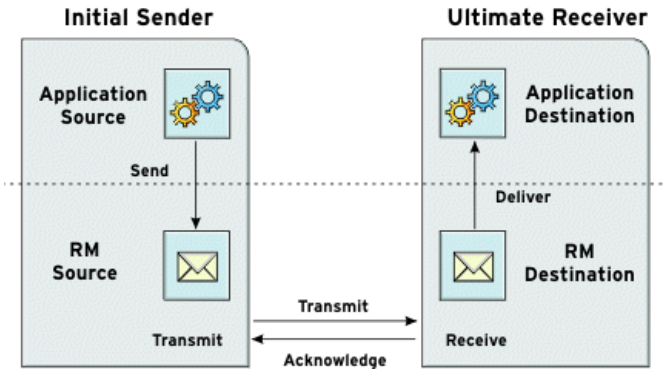


Figure 12: *Web Services Reliable Messaging*

The flow of WS-RM messages can be described as follows:

1. The RM source sends a `CreateSequence` protocol message to the RM destination. This contains a reference for the source endpoint that receives acknowledgements (`wsrm:AcksTo` endpoint).
2. The RM destination sends a `CreateSequenceResponse` protocol message back to the RM source. This contains the sequence ID for the RM sequence session.

3. The RM source adds an RM `Sequence` header to each message sent by the application source. This contains the sequence ID, and a unique message ID.
4. The RM source transmits each message to the RM destination.
5. The RM destination acknowledges the receipt of the message from the RM source by sending messages that contain the RM `SequenceAcknowledgement` header.
6. The RM destination delivers the message to the application destination in an exactly-once-in-order fashion.
7. The RM source retransmits a message for which it has not yet received an acknowledgement.

   The first retransmission attempt is made after a base retransmission interval. Successive retransmission attempts are made after a linear interval, or an exponential backoff interval (the default behavior). For more details, see "Configuring WS-RM Attributes" on page 132.

WS-RM delivery assurances

WS-RM guarantees reliable message delivery in a distributed environment, regardless of the transport protocol used. The source or destination endpoint raises an error if reliable delivery can not be assured.

The default Artix WS-RM delivery assurance policy is `ExactlyOnceInOrder`. This means that every message that is sent is delivered without duplication. If not, an error is raised on at least one endpoint. In addition, messages are delivered in the same order that they are sent.

Artix also supports the `ExactlyOnceConcurrent` and `ExactlyOnceReceivedOrder` delivery assurance policies. For more details, see "Message delivery assurance policies" on page 138.

Supported specifications

Artix supports the 2005/02 version of the WS-ReliableMessaging specification, which is based on the WS-Addressing 2004/08 specification.

Artix supports both the WS-Addressing 2004/08 specification and the WS-Addressing 2005/03 specification. However, WS-Addressing 2004/08 must be used with WS-ReliableMessaging.

For more information on WS-Addressing, see "Configuring a WS-A Message Exchange Pattern" on page 147.

Further information

For detailed information on WS-RM, see the specification at: http://specs.xmlsoap.org/ws/2005/02/rm/ws-reliablemessaging.pdf

Enabling WS-RM

This section describes the steps required to enable WS-RM in the Artix runtime. All the necessary settings are specified in an `artix.cfg` file.

Prerequisites

When you enable WS-RM, this automatically enables a WS-Addressing 2004 Message Exchange Pattern, which is required for WS-RM. For full details on how to manually configure WS-Addressing, see "Configuring a non-anonymous reply-to endpoint".

Note:A WS-Addressing 2004 MEP must be used with WS-RM. You can not use a WS-Addressing 2005 MEP with WS-RM.

In addition, if you wish to make a two-way invocation, you must configure a WS-RM-enabled WSDL port with a non-anonymous reply-to endpoint. For full details, see "Configuring a non-anonymous reply-to endpoint" on page 148.

Setting your orb_plugins list

To use Artix WS-RM, you must first specify the `wsrm` plug-in on the `orb_plugins` lists for your client and server. For example:

```
orb_plugins = ["xmlfile_log_stream", "iiop_profile", "giop",
   "iiop", "wsrm"];
```

Configuring WS-RM

WS-RM can be enabled in an `artix.cfg` file either at the bus-level or a specific WSDL port level. Port-specific configuration overrides bus-specific configuration.

Bus configuration

To enable WS-RM for a specific bus, use the following setting:

```
plugins:messaging_port:wsrm_enabled = "true";
```

WSDL port configuration

To enable WS-RM for a specific WSDL port, specify the WSDL service QName and the WSDL port name, for example:

```
plugins:messaging_port:wsrm_enabled:http://www.iona.com/bus
   /tests:SOAPHTTPService:SOAPHTTPPort="true";
```

Configuring WS-RM Attributes

You can specify various Artix WS-RM attributes in an `artix.cfg` file at the bus-level or WSDL port level. Port-specific configuration overrides bus-specific configuration.

The configurable WS-RM attributes are as follows:

- "WS-RM acknowledgement endpoint URI"
- "Use replyTo endpoint for acknowledgement"
- "Use server endpoint for acknowledgement"
- "Base retransmission interval"
- "Exponential backoff for retransmission"
- "Maximum unacknowledged messages threshold"
- "Max retransmission attempts threshold"
- "Acknowledgement interval"
- "Number of messages in an RM sequence"
- "Message delivery assurance policies"
- "Per-thread RM session"

You can also set these attributes in your client code (see "Configuring attributes in WS-RM contexts" on page 139).

WS-RM acknowledgement endpoint URI

This attribute specifies the endpoint at which the WS-RM source receives acknowledgements. This is also known as the `wsrm:AcksTo` endpoint.

The default value is the WS-A anonymous URI:

```
http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous
```

Bus configuration

The following example shows how to configure the acknowledgement endpoint for a specific bus:

```
plugins:wsrm:acknowledgement_uri =
   "http://localhost:0/WSASource/DemoAcksTo/";
```

WSDL port configuration

The following example shows how to configure the acknowledgement endpoint for a specific WSDL port:

```
plugins:wsrm:acknowledgement_uri:http://www.iona.com/bus/tests:SOAPHTTPService:SOAPHTTPPort = "http://localhost:0/WSASource/DemoAcksTo/";
```

Use replyTo endpoint for acknowledgement

If a proxy is used to make two-way invocations, you can configure the proxy so that its decoupled reply-to endpoint (`wsa:replyTo`), which receives the application response, also receives acknowledgements for application requests. In this way, the `wsa:replyTo` endpoint acts as a `wsrm:AcksTo` endpoint.

Bus configuration

The following example shows how to configure this for a specific Artix bus:

```
plugins:wsrm:use_wsa_replyto_endpoint_for_wsrm_acknowledg
   ement = "true";
```

WSDL port configuration

The following example shows how to configure this for a specific WSDL port:

```
plugins:wsrm:use_wsa_replyto_endpoint_for_wsrm_acknowled
   gement:http://www.iona.com/bus/tests:SOAPHTTPService:S
   OAPHTTPPort = "true";
```

Use server endpoint for acknowledgement

If a service is used to make two-way invocations, you can configure the service so that the server endpoint, which receives the application request, also receives acknowledgements for the application response. In other words, the server acts as a `wsrm:AcksTo` endpoint for the reverse WS-RM channel.

Bus configuration

The following example shows how to configure for a specific Artix bus:

```
plugins:wsrm:use_server_endpoint_for_wsrm_acknowledgement =
   "true";
```

WSDL port configuration

The following example shows how to configure for a specific WSDL port:

```
plugins:wsrm:use_server_endpoint_for_wsrm_acknowledgement:http:
   //www.iona.com/bus/tests:SOAPHTTPService:SOAPHTTPPort =
   "true";
```

Order of preference for acknowledgement endpoints

The order of preference in which a `wsrm:AcksTo` endpoint is chosen for a RM source endpoint is as follows:

1. If the RM source endpoint is explicitly configured (in a configuration file or code) to use a non-anonymous `wsrm:AcksTo` endpoint, it is chosen.
2. On the client-side, if the RM source endpoint is configured to use the `wsa:replyTo` endpoint as `wsrm:AcksTo`, it is chosen for the application request.

   On the server-side, if the RM source endpoint is configured to use the server endpoint as `wsrm:AcksTo`, it is chosen for the application response.
3. If neither 1 or 2 is specified, the anonymous `wsrm:AcksTo` endpoint is chosen.

Base retransmission interval

This attribute specifies the interval at which a WS-RM source retransmits a message that has not yet been acknowledged. The default value is `2000` milliseconds.

Bus configuration

The following example shows how to set the base retransmission interval for a specific bus:

```
plugins:wsrm:base_retransmission_interval = "3000";
```

WSDL port configuration

The following example shows how to set the base retransmission interval for a specific WSDL port:

```
plugins:wsrm:base_retransmission_interval:http://www.iona.
   com/bus/tests:SOAPHTTPService:SOAPHTTPPort = "3000";
```

Exponential backoff for retransmission

This attribute determines if successive retransmission attempts for an unacknowledged message are performed at exponential intervals. The default value is `false`, which means that they are attempted at exponential intervals.

If the value is `true` (exponential backoff disabled), the retransmission of unacknowledged messages is performed at the base retransmission interval.

Bus configuration

The following example shows how to set the exponential backoff for retransmission for a specific bus:

```
plugins:wsrm:disable_exponential_backoff_retransmission_i
   nterval = "true";
```

WSDL port configuration

The following example shows how to set the exponential backoff for retransmission for a specific WSDL port:

```
plugins:wsrm:disable_exponential_backoff_retransmission_i
   nterval:http://www.iona.com/bus/tests:SOAPHTTPService:
   SOAPHTTPPort = "true";
```

Maximum unacknowledged messages threshold

This attribute specifies the maximum permissible number of unacknowledged messages at the WS-RM source. When the WS-RM source reaches this limit, it sends the last message with a `wsrm:AckRequested` header indicating that a WS-RM acknowledgement should be sent by the WS-RM destination as soon as possible.

In addition, when the WS-RM source has reached this limit, it does not accept further messages from the application source. This means that the caller thread (making the invocation on the proxy) is blocked until the number of unacknowledged messages drops below the threshold.

The default value is `-1` (no limit on number of unacknowledged messages).

Bus configuration

The following example shows how to set the maximum unacknowledged messages threshold for a specific bus:

```
plugins:wsrm:max_unacknowledged_messages_threshold =
   "50";
```

WSDL port configuration

The following example shows how to set the maximum unacknowledged messages threshold for a specific WSDL port:

```
plugins:wsrm:max_unacknowledged_messages_threshold:http://w
   ww.iona.com/bus/tests:SOAPHTTPService:SOAPHTTPPort =
   "50";
```

Max retransmission attempts threshold

This attribute specifies the maximum number of retransmission attempts that the RM source session makes for an unacknowledged message. If the number of retransmission attempts reaches this threshold, the RM source session sends a `wsrm:SequenceTerminated` fault to the peer RM destination session, and closes the session. Any subsequent attempt to send a message on this session results in an `IT_Bus::Exception`. The default value is `-1` (no limit on the number of retransmission attempts).

Bus configuration

The following example shows how to set the maximum number of retransmission attempts for a specific bus:

```
plugins:wsrm:max_retransmission_attempts = "8";
```

WSDL port configuration

The following example shows how to set the maximum number of retransmission attempts for a specific WSDL port:

```
plugins:wsrm:max_retransmission_attempts:http://www.iona.
   com/bus/tests:SOAPHTTPService:SOAPHTTPPort = "8";
```

Acknowledgement interval

This attribute specifies the interval at which the WS-RM destination sends asynchronous acknowledgements. These are in addition to the synchronous acknowledgements that it sends upon receipt of an incoming message. The default asynchronous acknowledgement interval is `3000` milliseconds.

Asynchronous acknowledgements are sent by the RM destination only if both of the following conditions are met:

1. The RM destination is using a non-anonymous `wsrm:AcksTo` endpoint.
2. The RM destination is waiting for some messages to be received from the RM source.

For example, the RM destination receives five messages with message IDs of `1`, `2`, `3`, `4`, and `5`. This means that it has received all messages up to the highest received message (`5`). There are no missing messages in this case, so the RM destination will not send an asynchronous acknowledgement.

However, take the case where the RM destination receives 5 messages with message IDs of 1, 2, 4, 5, and 7. This means that messages 3 and 6 are missing, and the RM destination is still waiting to receive them. This is the case where the RM destination sends asynchronous acknowledgements.

Note: The RM destination still sends synchronous acknowledgements upon receipt of a message from the RM source.

Bus configuration

The following example shows how to set the acknowledgement interval for a specific bus:

```
plugins:wsrm:acknowledgement_interval = "2500";
```

WSDL port configuration

The following example shows how to set the acknowledgement interval for a specific WSDL port:

```
plugins:wsrm:acknowledgement_interval:http://www.iona.com/b
   us/tests:SOAPHTTPService:SOAPHTTPPort = "2500";
```

Number of messages in an RM sequence

This attribute specifies the maximum number of user messages that are permitted in a WS-RM sequence. The default is unlimited; and this is sufficient is for most cases.

When this attribute is set, the RM endpoint creates a new RM sequence when the limit is reached and after receiving all the acknowledgements for the messages previously sent. The new message is then sent using the new sequence.

Bus configuration

The following example shows how to set the maximum number of messages for a specific bus

```
plugins:wsrm:max_messages_per_sequence = "1";
```

WSDL port configuration

The following example shows how to set the maximum number of messages for a specific WSDL port:

```
plugins:wsrm:max_messages_per_sequence:http://www.iona.com/b
   us/tests:SOAPHTTPService:SOAPHTTPPort = "1";
```

Message delivery assurance policies

You can configure the RM destination to use the following delivery assurance policies:

ExactlyOnceInOrder: The RM destination delivers the messages to the application destination exactly once, and in increasing order of RM message ID. The calls to the application destination are therefore serialized. This is the default.

ExactlyOnceConcurrent: The RM destination delivers the messages to the application destination exactly once. But instead of a serialized message delivery (as in `ExactlyOnceInOrder`), messages are delivered concurrently, so they may not be delivered in order. However, for a message with ID n that is being delivered, all the messages in the range of 1 to n are received and acknowledged by the RM destination.

ExactlyOnceReceivedOrder: The RM destination delivers the messages to the application destination exactly once, and as soon as it is received from the underlying transport. The RM destination makes no attempt to ensure that either the messages are delivered in the order of message ID, or all the previous messages have been received/acknowledged. The benefit of this policy is that it avoids a context-switch during dispatch in the RM layer, and messages are not stored in the in-memory undelivered messages map.

Bus configuration

The default delivery assurance policy is `ExactlyOnceInOrder`. You can specify a different policy at bus level using the following variable:

```
plugins:wsrm:delivery_assurance_policy =
   "ExactlyOnceConcurrent";
```

WSDL port configuration

The following example shows how to set this policy at the WSDL port level:

```
plugins:wsrm:delivery_assurance_policy:http://www.iona.com/bu
   s/tests:SOAPHTTPService:SOAPHTTPPort =
   "ExactlyOnceConcurrent";
```

Per-thread RM session

When an RM source endpoint is concurrently invoked, by default, the RM session is shared by all threads. However, with the per-thread RM session attribute enabled, the RM source endpoint transparently creates a different RM sequence session for each invoking thread.

Enabling this setting eliminates the possibility of indeterminate message ID allocation. All messages sent by a particular thread are allocated a message ID in increasing order. When the RM source endpoint is closed, it closes all the open RM sequence sessions. The default value is `false` (disabled).

Bus configuration

The following example shows how to enable a per-thread RM session for a specific bus:

```
plugins:wsrm:enable_per_thread_sequence_scope = "true";
```

WSDL port configuration

The following example shows how to enable a per-thread RM session for a specific WSDL port:

```
plugins:wsrm:enable_per_thread_sequence_scope:http://www.i
   ona.com/bus/tests:SOAPHTTPService:SOAPHTTPPort =
   "true";
```

Configuring attributes in WS-RM contexts

For C++ applications, you can also specify Artix WS-RM attributes programmatically using a configuration context. Using this approach, the context is specific to the current proxy only, and can not be used by another proxy created subsequently. For full details and examples, see ***Developing Artix Applications with C++***.

The order of precedence for setting WS-RM attributes is as follows:

1. Configuration context (programmatic).
2. WSDL port (configuration file).
3. Artix bus (configuration file).

Configuring WS-RM Threading

The Artix WS-RM layer maintains a bus-specific internal thread pool. It uses this work queue to borrow execution resources for various asynchronous tasks. For example, these tasks include:

- Retransmission scheduling at the RM source.
- Retransmissions at the RM source.
- Asynchronous acknowledgement scheduling at the RM destination.
- Asynchronous acknowledgement at the RM destination.
- Concurrent message dispatches to the application destination.

Configuring a WS-RM thread pool

You can configure the WS-RM thread pool using the following variables:

initial_threads specifies the number of initial threads in the WS-RM thread pool. The default is:

```
plugins:wsrm:thread_pool:initial_threads="5";
```

high water mark specifies the maximum number of threads allowed in the WS-RM thread pool. The default is:

```
plugins:wsrm:thread_pool:high_water_mark="-1";
```

low water mark specifies the minimum number of threads allowed in the WS-RM thread pool. The default is:

```
plugins:wsrm:thread_pool:low_water_mark="-1";
```

max queue size specifies the maximum number of request items that can be queued on the WS-RM thread work queue. The default is:

```
plugins:wsrm:thread_pool:max_queue_size="-1";
```

stack size specifies the stack size for each thread. The stack size is specified in bytes. The default is:

```
plugins:wsrm:thread_pool:stack_size="OS-specificDefault
   ThreadStackSize";
```

Configuring WS-RM Persistence

The Artix WS-RM features already described in this chapter provide reliability for cases such as network failures. Enabling WS-RM persistence improves the Quality of Service by providing reliability across other types of failures such as an RM source or destination crash.

WS-RM persistence involves storing the state of the various RM endpoints in persistent storage. This enables the endpoints when reincarnated to continue sending and receiving messages as before the crash.

Artix enables WS-RM persistence for at bus level in a configuration file, or in code using an Artix context. The WS-RM persistence store implementation uses a Berkeley DB, and is available as a separate plug-in. In addition, the persistent store is also exposed using a C++ API. If you wish to implement your own persistence mechanism, you can implement this API with your preferred DB (see ***Developing Artix Applications with C++***).

Note: WS-RM persistence is supported for oneway calls only. It is disabled by default.

How it works

Artix WS-RM persistence works as follows:

- At the RM source endpoint, an outgoing message is persisted before transmission. It is evicted from the persistent store after the acknowledgement is received.
- After a recovery from crash, it recovers the persisted messages and retransmits until all the messages have been acknowledged. At that point, the RM sequence is closed.
- At the RM destination endpoint, an incoming message is persisted, and upon a successful store, the acknowledgement is sent. When a message is successfully dispatched, it is evicted from the persistent store.
- After a recovery from crash, it recovers the persisted messages and dispatches them. It also brings the RM sequence to a state where new messages are accepted, acknowledged, and delivered.

Enabling WS-RM persistence

To enable WS-RM persistence for a specific Artix bus, perform the following steps:

1. Add the `wsrm_db` plug-in to the `orb_plugins` list. For example:

```
orb_plugins = ["xmlfile_log_stream", "iiop_profile",
   "giop", "iiop", "wsrm_db"];
```

   The `wsrm_db` plug-in is the plug-in that implements the RM persistent store API. The `wsrm` plug-in is loaded automatically when `wsrm_db` is specified in the `orb_plugins` list.
2. Configure the Berkeley DB store used by the `wsrm_db` plug-in as follows:

```
plugins:artix:db:home = "db_directory";
```

   The default value is the current directory (`.`).

Further details

For working examples of reliable messaging in Artix, see the `.../advanced/wsrm` demo.

Part III

Accessing Artix Services

In this part

This part contains the following chapters:

Configuring WS-Addressing	page 145
Publishing WSDL Contracts	page 151
Accessing Contracts and References	page 159
Accessing Services with UDDI	page 173
Embedding Artix in a BEA Tuxedo Container	page 175

Configuring WS-Addressing

Artix supports WS-Addressing for C++ applications. This chapter explains how to configure WS-Addressing Message Exchange Patterns in an Artix runtime environment.

Introduction

Web Services Addressing (WS-A) provides a mechanism to identify and locate Web services and messages, which is independent of the transports used. This section explains the WS-Addressing Message Exchange Patterns (MEPs) used by Artix.

WS-Addressing Message Exchange Patterns

Artix supports WS-Addressing 2004 and 2005 Message Exchange Patterns in SOAP message headers. These enable Artix to send a request to an endpoint specified by a `wsa:To` header, and to receive a reply at an endpoint specified by a `wsa:ReplyTo` header.

Anonymous URI

If a `wsa:ReplyTo` header is not specified, by default, Artix uses the anonymous URI to synchronously receive the reply. For example, the WS-Addressing 2004 anonymous URI is:

```
http://schemas.xmlsoap.org/ws/2004/08/addressing/role/ano
   nymous
```

While the WS-Addressing 2005 anonymous URI is

```
http://www.w3.org/2005/08/addressing/anonymous
```

Non-anonymous address

When a non-anonymous `wsa:ReplyTo` header is used, the reply is received asynchronously at the reply-to endpoint. The reply is matched with the request using `wsa:MessageId` and `wsa:RelatesTo` message headers. From the user's perspective, this is still a two-way synchronous call, and the asynchronicity is handled by Artix. For oneway calls, the reply-to endpoint is not needed.

How it works

Artix WS-A MEPs follow a typical request-response pattern. At the HTTP connection level, when an anonymous `wsa:ReplyTo` header is used, the response is returned on the same HTTP connection.

However, when a non-anonymous `wsa:ReplyTo` is used, the response is returned on a separate connection. This also means that an Artix client listens on the endpoint denoted by the `wsa:ReplyTo` header. The following steps show this decoupled request-response MEP in more detail:

1. The Artix client creates an HTTP listener specified by the `wsa:ReplyTo` header. If the listener can not be created, it throws an `IT_Bus::Exception`.
2. The Artix client sends an HTTP request containing the application request to the service.
3. The Artix runtime treats the application request as one-way. This means the Artix HTTP stack expects to receive an HTTP response with status code `202` (`Accepted`).
4. The Artix client receives the application response from the service as a HTTP request on a decoupled HTTP connection.
5. The Artix runtime treats the application response as one-way and sends back a HTTP response with status code `202` (`Accepted`) on the decoupled HTTP connection.

This decoupled mechanism means there can be multiple outstanding application requests at any time. The request and response are correlated using `wsa:MessageId` and `wsa:RelatesTo` headers.

In addition, the requesting client thread blocks and creates a listener before sending the request. This is important in the event of firewalls, port conflicts, and so on.

WS-Addressing and security

In a decoupled interaction, when a non-anonymous `wsa:ReplyTo` is used, the security configuration for the request is the same as a normal Artix client-server security scenario.

However, the roles are reversed for the response. The client creates an HTTP listener corresponding to the `wsa:replyTo` endpoint, and the server creates an HTTP connection to send back the response. Therefore, the security role is reversed in this scenario. The client should be configured for server-side security, and the server should be configured for client-side security.

For full details of how to configure Artix client-server security, see the ***Artix Security Guide***.

WS-Addressing and WS-RM

When WS-Reliable Messaging is enabled in the Artix runtime, this automatically enables a WS-Addressing 2004 MEP.

Note:A WS-Addressing 2004 MEP must be used with WS-RM. You can not use a WS-Addressing 2005 MEP with WS-RM.

For information on how to configure WS-Reliable Messaging, see "Deploying WS-Reliable Messaging".

Supported specifications

Artix supports both the WS-Addressing 2004/08 specification and the WS-Addressing 2005/03 specification. However, WS-Addressing 2004/08 must be used with WS-Reliable Messaging (WS-RM).

For details of how to configure a MEP, see "Configuring a WS-A Message Exchange Pattern" on page 147.

Further information

For detailed information, see the WS-Addressing WSDL Binding specification at:

http://www.w3.org/TR/2006/WD-ws-addr-wsdl-20060216/

Configuring a WS-A Message Exchange Pattern

This section explains how to configure a WS-Addressing Message Exchange Pattern in the Artix runtime.

Enabling a WS-Addressing 2004 MEP

You can enable a WS-Addressing 2004 MEP in an Artix `.cfg` file either at the Artix bus-level or a specific WSDL port level. Port-specific configuration overrides bus-specific configuration. When WS-RM is enabled, a WS-Addressing 2004 MEP is enabled automatically (see "WS-Addressing and WS-RM" on page 146).

Bus configuration

To enable a WS-Addressing MEP at bus level, use the following setting:

```
plugins:messaging_port:supports_wsa_mep = "true";
```

WSDL port configuration

To enable WS-A at a specific WSDL port level, specify the WSDL service QName and the WSDL port name, for example:

```
plugins:messaging_port:supports_wsa_mep:http://www.iona.com/b
   us/tests:SOAPHTTPService:SOAPHTTPPort="true";
```

Enabling a WS-Addressing 2005 MEP

Similarly, you can enable a WS-Addressing 2005 MEP in an Artix `.cfg` file either at the Artix bus-level or a specific WSDL port level. Port-specific configuration overrides bus-specific configuration.

Bus configuration

To enable a WS-Addressing MEP at bus level, use the following setting:

```
plugins:messaging_port:supports_wsa_2005_mep = "true";
```

WSDL port configuration

To enable WS-A at a specific WSDL port level, specify the WSDL service QName and the WSDL port name, for example:

```
plugins:messaging_port:supports_wsa_2005_mep:http://www.iona.com/bus/tests:SOAPHTTPService:SOAPHTTPPort="true";
```

Note: Either WS-A 2004 or WS-A 2005 should be enabled. If both are enabled, Artix enables WS-A 2005, and ignores WS-A 2004, and logs a `MessagingPort` warning message.

Configuring a non-anonymous reply-to endpoint

The WS-A reply-to endpoint specifies a URI for receiving acknowledgement messages from the destination. The scope of a reply-to endpoint is at the proxy level. In Artix, two proxies can not share the same endpoint. This means that each proxy has its own reply-to endpoint.

There are two ways of configuring a reply-to endpoint:

- "Setting a reply-to endpoint in configuration"
- "Setting a reply-to endpoint in a context"

Setting a reply-to endpoint in configuration

The WS-A reply-to endpoint can be set in an Artix `.cfg` file, at the Artix bus level or WSDL port level.

Because reply-to endpoints must have a unique URI per-proxy, a base URI is specified in configuration. For example, if the base URI is specified as:

```
plugins:messaging_port:base_replyto_url=
   "http://localhost:0/WSATestClient/BaseReplyTo/";
```

And if two proxies are instantiated, the first proxy has a reply-to endpoint whose URI is as follows:

```
"http://localhost:2356/WSATestClient/BaseReplyTo/ReplyTo0001";
```

Similarly, the second proxy has a reply-to endpoint whose URI is as follows:

```
"http://localhost:2356/WSATestClient/BaseReplyTo/ReplyTo0002";
```

Setting a reply-to endpoint in a context

For C++ applications, you can also set a WS-A reply-to endpoint programmatically using a configuration context. Using this approach, the context is specific to the current proxy only, and can not be used by a proxy created subsequently. You must also ensure that it is deleted after use. For full details and examples, see ***Developing Artix Applications with C++***.

Further details

For detailed information, see the WS-Addressing WSDL Binding specification at:

http://www.w3.org/TR/2006/WD-ws-addr-wsdl-20060216/

Publishing WSDL Contracts

This chapter describes how to publish WSDL files that correspond to specific Web services. This enables clients to access the WSDL file and invoke on the service.

Artix WSDL Publishing Service

The Artix WSDL publishing service enables Artix processes to publish WSDL files for specific Web services. Published WSDL files can be downloaded by other Artix processes (for example, especially clients), or viewed in a Web browser.

The WSDL publishing service enables Artix applications to be used in various deployment models, without the need to specify file system locations. It is the recommended way to publish WSDL for Artix services.

The WSDL publishing service is implemented by the `wsdl_publish` plug-in. This plug-in can be loaded by any Artix process that hosts a Web service endpoint. This includes server applications, Artix routing applications, and applications that expose a callback object.

Use with endpoint references

It is recommended that you use the WSDL publishing service for any applications that generate and export references. To use references, the client must have access to the WSDL contract referred to by the reference. The simplest way to accomplish this is to use the WSDL publishing service.

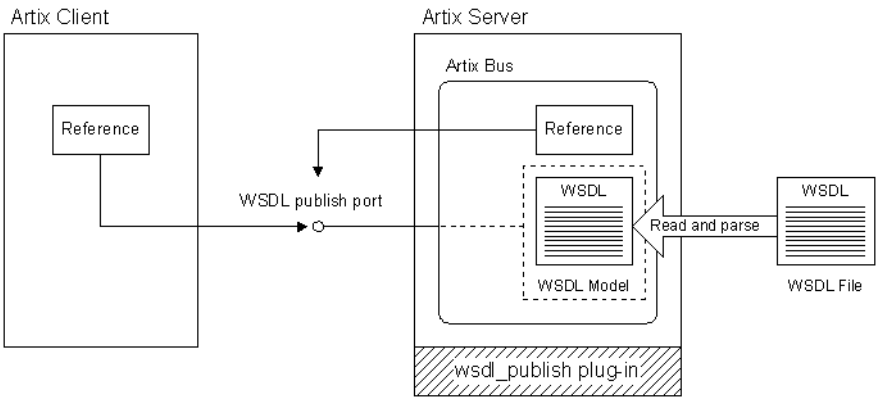
Figure 13 shows an example of creating references with the WSDL publishing service. The `wsdl_publish` plug-in automatically opens a port, from which clients can download a copy of the server's dynamically updated WSDL file. Generated references have their WSDL location set to the following URL:

```
http://Hostname:WSDLPublishPort/QueryString
```

`Hostname` is the server host, *`WSDLPublishPort`* is a TCP port used to serve up WSDL contracts, and *`QueryString`* is a string that requests a particular WSDL contract (see "Querying the WSDL Publishing Service" on page 155). If a client accesses the WSDL location URL, the server converts the WSDL model to XML on the fly and returns the WSDL contract in a HTTP message.

For more details on references, see ***Developing Artix Applications in C++***.

Figure 13: *Creating References with the WSDL Publishing Service*



Multiple transports

The WSDL publishing service makes the WSDL file available through an HTTP URL. However, the Web service described in the WSDL file can use a transport other than HTTP.

For example, when the `wsdl_publish` plug-in is loaded into an Artix server process that hosts a Web service using IIOP, it publishes the service's WSDL file at an HTTP URL.

Configuring the WSDL Publishing Service

This section describes how to load the `wsdl_publish` plug-in, and configure it to suit your needs.

Note: In a production environment, it is strongly recommended that you set a `wsdl_publish` port and hostname format.

Loading the wsdl_publish plug-in

To load the `wsdl_publish` plug-in, add the `wsdl_publish` string to your `orb_plugins` setting, in the process configuration scope. For example, if your configuration scope is `samples.server`, you might use the following `orb_plugins` list:

```
# artix.cfg

demos{
    server
    {
        orb_plugins = ["xmlfile_log_stream", "wsdl_publish"];
        ...
    };
};
```

When the process starts, the WSDL file is available at an HTTP URL that uses a TCP port assigned by the operating system. This URL is embedded in the WSDL `location` value in an endpoint reference. Processes receiving the reference can download the WSDL file from this URL. However, there is no easy way to determine the port assigned by the operating system. This makes it difficult to view the WSDL file in a web browser, or to open this port through a firewall. You can solve this problem by configuring a port for publishing WSDL.

Specifying a port for publishing WSDL

To enable viewing of WSDL files in a web browser, configure the `wsdl_publish` plug-in to use a specified port instead of a one assigned by the operating system. The `plugins:wsdl_publish:publish_port` configuration variable specifies the TCP port that WSDL files are published on. For example,

```
plugins:wsdl_publish:publish_port="2222";
```

When specifying a `publish_port`, you must confirm that the specified port is not already in use.

Viewing the WSDL file in a web browser

If you know either the `wsdl_publish` plug-in or the TCP port used by the service, you can view or download the WSDL file in a web browser.

In the browser address box, enter one of the following URLs, where *`WSDLPublishPort`* is the TCP port used by the `wsdl_publish` plug-in:

```
http://HostNameOrIP:WSDLPublishPort/get_wsdl?
http://HostNameOrIP:WSDLPublishPort
```

The Artix process returns a web page that lists all of its services. Click on an entry to retrieve the corresponding WSDL file.

Alternatively, you can enter one of the following URLs, where *`ServicePort`* is the TCP port used by the Web service:

```
http://HostNameOrIP:ServicePort/service?wsdl
http://HostNameOrIP:ServicePort/service
```

The Artix process returns the WSDL file for the service. The `http://`*`HostNameOrIP`*`:`*`ServicePort`*`/service?wsdl` format is used in the JAX-WS specification.

Specifying a hostname format

The `plugins:wsdl_publish:hostname` variable specifies how the hostname is constructed in the `wsdl_publish` URL. This is the URL that the `wsdl_publish` plug-in uses to retrieve WSDL contracts.

This variable has the following possible values:

`canonical`	The fully qualified hostname (for example, `http://myhost.mydomain.com`).
`unqualified`	The unqualified local hostname (for example, `http://myhost`).
`ipaddress`	The host's IP address (for example, `http://10.1.2.3`).
`SecondaryHostName`	For multi-homed machines, the specified literal string of a secondary hostname. Specify a logical name or a virtual IP address (for example, `http://myhost.mydomain.com` or `http://10.1.2.3`). Any leading or trailing white spaces are stripped out.

By default, the unqualified primary hostname is used.

Note: This variable should not be confused with the following:

- `policies:soap:server_address_mode_policy:publish_hostname`
- `policies:at_http:server_address_mode_policy:publish_hostname`

These specify how endpoint URLs are published in WSDL contracts.

`plugins:wsdl_publish:hostname` specifies only how to construct the URL used by the `wsdl_publish` plug-in to access the WSDL.

Whereas, `policies:soap:server_address_mode_policy:publish_hostname` and `policies:at_http:server_address_mode_policy:publish_hostname` specify how to construct the URL in the published WSDL contract.

You must be aware of both sets of configuration entries when using the `wsdl_publish` plug-in (for example, to avoid publishing a WSDL file that does not contain a complete URL).

Specifying WSDL preprocessing

You can use the `plugins:wsdl_publish:processor` variable to specify the kind of preprocessing done before publishing a WSDL contract.

Because published contracts are intended for client consumption, by default, all server-side WSDL artifacts are removed from the published contract. You can also specify to remove all Artix-specific extensors. Preprocessing can also be disabled; the only modification is updating the `location` and `schemaLocation` attributes to HTTP based URLs.

This variable has the following possible values:

`artix`	Remove server-side artifacts. This is the default setting.

standard	Remove server-side artifacts and Artix proprietary extensors.
none	Disable preprocessing.

For example:

```
plugins:wsdl_publish:processor="standard";
```

Querying the WSDL Publishing Service

If you know the TCP port used by either the `wsdl_publish` plug-in or the Web service, you can view or download the WSDL file in a web browser.

This section shows examples of querying the WSDL Publishing service. It also describes its HTML menu and WSIL support.

Example query syntax

Assume you configured `wsdl_publish` using the following values on a system with an IP address of `10.1.2.3`:

```
test.scope {
  plugins:wsdl_publish:publish_port = 1234;
  plugins:wsdl_publish:hostname = "ipaddress";
};
```

The `wsdl_publish` base URL is `http://10.1.2.3:1234`. And requests on the following types of URLs are serviced:

- `http://10.1.2.3:1234/get_wsdl`, `http://10.1.2.3:1234/get_wsdl/`, `http://10.1.2.3:1234/get_wsdl?`, or `http://10.1.2.3:1234/get_wsdl/?` returns the HTML Menu (see "Using the HTML menu" on page 156).
- `http://10.1.2.3:1234/get_wsdl?service=`*`name`*`&scope=`*`EncodedUrl`* returns the contract for the service specified in the query string.
- `http://10.1.2.3:1234/get_wsdl?stub=`*`EncodedUrl`* returns the contract for IONA specific services.
- `http://10.1.2.3:1234/inspection.wsil` returns a WSIL document containing information about active Web services (see "WSIL support" on page 157).
- `http://10.1.2.3:1234/get_wsdl/`*`context/filename`*`.wsdl` returns the specified WSDL contract. The value of *`context`* is generated at runtime.
- `http://10.1.2.3:2000/`*`service`* or `http://10.1.2.3:2000/`*`service`*`?wsdl` returns the contract for the specified service. The value of the URL is the same as the one specified in the WSDL as the `soap:address` of the service.

If an invalid URL is provided, `wsdl_publish` returns an HTTP 404 (File Not Found) Error.

For more details, see "Viewing the WSDL file in a web browser".

Querying CORBA services

Use the following `wsdl_publish` URL format when using CORBA-only services:

```
WSDLPublishURL/get_wsdl?service=Name&scope=NS
```

For example, a client could use the following setting:

```
bus:initial_contract:url:greeter =
"http://localhost:9005/get_wsdl?service=GreeterService&scope=http://www.iona.com/demo";
```

For more details, see ***Artix for CORBA***.

Using the HTML menu

The WSDL publishing service provides an HTML menu page that contains links to the contracts of activated services. This page shows all services activated on the current bus associated with a specified `wsdl_publish` instance.

Note: A process might have more than one active bus, and so more Web services might be activated in that process. Contracts for other Web services can be obtained from the `wsdl_publish` instance associated with their buses.

For example, an `it_container` instance is started on port `2000`, and the `wsdl_publish` port is configured as `1234`. The HTML menu available at `http://10.1.2.3:1234/get_wsdl` is as follows:

WSDL Services available

ContainerService(http://ws.iona.com/container)

ContainerService(http://ws.iona.com/container)

The HTML source is as follows:

```
<html>
 <body>
   <h1>WSDL Services available</h1>
   <a href=
   "http://10.1.2.3:2000/get_wsdl/WPabcd/container.wsdl">Contai
   nerService(http://ws.iona.com/container)</a>
   <br>
   <a href=
   "http://10.1.2.3:2000/services/container/ContainerService?ws
   dl">ContainerService(http://ws.iona.com/container)</a>
   <br>
 </body>
</html>
```

The first entry downloads the WSDL from the `wsdl_publish` port, while the second downloads the WSDL from the service's port.

The hostname format assigned to `plugins:wsdl_publish:hostname` affects the syntax of the first entry's URL, while the `server_address_mode_policy` variables affect the syntax of the second entry's URL. For more details, see “Specifying a hostname

format" on page 154.

WSIL support

The Web Services Inspection Language (WSIL) specification, at http://www.ibm.com/developerworks/webservices/library/ws-wsilover, provides a standard way of inspecting a Web service, and getting the contracts of active Web services.

For example, the WSIL document available from `http://10.1.2.3:1234/inspection.wsil` has the following content:

```
<?xml version="1.0"?>
<inspection
   targetNamespace="http://schemas.xmlsoap.org/ws/2001/10/inspection/"
            xmlns="http://schemas.xmlsoap.org/ws/2001/10/inspection/"

   xmlns:wsilwsdl="http://schemas.xmlsoap.org/ws/2001/10/inspection/wsdl/">
  <service>
    <description referencedNamespace="http://schemas.xmlsoap.org/wsdl/"
              location="http://10.1.2.3:1234/get_wsdl/WPabcd/container.wsdl">
      <wsilwsdl:reference>
        <wsilwsdl:referencedService xmlns:ns1="http://ws.iona.com/container">
          ns1:ContainerService
        </wsilwsdl:referencedService>
      </wsilwsdl:reference>
    </description>
  </service>
  <service>
    <description referencedNamespace="http://schemas.xmlsoap.org/wsdl/"

   location="http://10.1.2.3:2000/services/container/ContainerService?wsdl">
      <wsilwsdl:reference>
        <wsilwsdl:referencedService xmlns:ns1="http://ws.iona.com/container">
          ns1:ContainerService
        </wsilwsdl:referencedService>
      </wsilwsdl:reference>
    </description>
  </service>
</inspection>
```

HTTP transport

For an Artix process that exposes a Web service over HTTP, the WSDL Publishing service provides an alternative way to view or download the WSDL file.

Artix distinguishes between HTTP POST and HTTP GET methods. HTTP POST methods are used to invoke on the target Web service. HTTP GET methods return the WSDL file.

In the following WSDL file, the `port` element specifies the HTTP transport and makes the Web service available at a specified HTTP URL.

```
<definitions name="HelloWorld"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
...>
. . .
<service name="SOAPService">
<port binding="tns:Greeter_SOAPBinding" name="SoapPort">
<soap:address location="http://hostname:9000/test"/>
</port>
</service>
</definitions>
```

If the Artix server hosting this service loads the `wsdl_publish` plug-in, the WSDL file may be viewed or downloaded using a web browser.

In the browser's address box, enter:

```
http://hostname:9000/test
```

For this approach to work, the service's HTTP URL must include a unique context (in this example case, `/test`).

Servant registration

When the WSDL Publishing service publishes a WSDL file for a service using a statically registered servant, the published file contains valid connection details. This is true even if the WSDL file originally specified dynamic port assignment (for example, an HTTP transport with a location URL of the form `http://`*`HostName`*`:0`, or an IIOP transport with a location entry of the form `ior:`).

The HTTP URL is revised to `http://`*`HostName`*`:`*`ServicePort`*, where *`ServicePort`* is a TCP port assigned by the operating system. The IIOP location entry is revised to `IOR:...`, where `...` is the string representation of the CORBA object reference.

However, when the `wsdl_publish` plug-in publishes a WSDL file for a service using a transiently registered servant, the published file does not contain valid connection details. Valid connection details can only be obtained from the endpoint reference corresponding to the service.

For more details on servant registration, see ***Developing Artix Applications in C++***.

Accessing Contracts and References

Artix enables you to decouple the location of WSDL contracts and endpoint references from your server and client. This avoids hard-coding the location of WSDL files in your applications. This chapter explains the benefits, and shows how to use the different ways of accessing WSDL contracts and endpoint references.

Introduction

Artix enables client and server applications to access WSDL service contracts and endpoint references in a variety of ways (for example, by specifying their location on the command line, or in a configuration file). This section explains the benefits of using these features.

Hard coding WSDL in servers

Hard coding WSDL in servers limits the portability of your application, and can make it more difficult to develop and deploy.

For example, you have developed a Web service application that includes a client and a service implemented in a server process. When you first write the application, you have a local copy of the WSDL, and you have hard coded the WSDL location into your application.

Example C++ server

```
// C++
QName service_qname("", "SOAPService",
   http://www.iona.com/hello_world_soap_http);

HelloWorldImpl servant(bus);
  bus->register_servant(
      "../../etc/hello.wsdl",
     service_qname
  );
```

Hard coding WSDL in clients

Similarly, you have also hard-coded your client with the location of your local WSDL:

Example C++ client

```
// C++
HelloWorldClient proxy("../../etc/hello.wsdl");
proxy.sayHello();
```

Deploying your application

However, when your application is no longer a demo, and you want to deploy it in multiple locations, your hard-coded application may make this difficult. For example, if your client is no longer run from the same directory or machine as the server.

To solve this problem, Artix enables you to write code that is location independent, and therefore easy to distribute and deploy.

Note: These features are designed for WSDL-based services. They do not provide mechanisms for resolving local objects. For details of how to do this, see ***Developing Artix Applications with C++***.

Enabling Server and Client Applications

Artix addresses two typical use case scenarios:

- Enabling server applications to access WSDL contracts.
- Enabling client applications to access endpoint references.

Artix supports both of these use cases for C++ applications.

Enabling servers to access WSDL

When you want to activate your service in a mainline or a plug-in, you should not hard code the WSDL location. Instead, you can use Artix APIs to decouple the WSDL location from your application logic.

Example

The C++ `get_service_contract()` function takes the QName of the desired service as a parameter, and returns a pointer to the specified service. When you change your old hard-coded application to use this method, your C++ server becomes:

```
// C++
IT_Bus::QName service_qname(
  "", "SOAPService",
   "http://www.iona.com/hello_world_soap_http"
);
// Find the WSDL contract.
IT_WSDL::WSDLService* wsdl_service = bus->get_service_contract(
    service_qname
);

// Register the servant
bus->register_servant(
    servant,
    *wsdl_service
);
```

For simplicity, this example does not show any error handling. For details, see ***Developing Artix Applications with C++***.

Associating your server with a specific WSDL contract is not addressed in your application code. This is specified at runtime instead. The available options are explained in "Accessing WSDL Contracts" on page 162.

Enabling clients to access endpoint references

When you want to initialize your client proxies in your applications, you should no longer depend on local WSDL files or static stub code information to properly instantiate a proxy. Instead, you can use Artix APIs to decouple the location of client references from your application logic.

Note: The Artix 3.0 APIs for resolving initial references have been deprecated from Artix 4.0. These APIs are supported for backwards compatibility, however, it is recommended that you update your applications to use the WS-Addressing APIs available since Artix 4.0.

Example

The C++ `resolve_initial_reference()` function takes the QName of the desired service as a parameter, and returns the endpoint reference for the specified service.

You can change your old hard-coded client application as follows:

```
// C++
IT_Bus::QName service_qname(
  "", "SOAPService",
   "http://www.iona.com/hello_world_soap_http"
);

WS_Addressing::EndpointReferenceType ref;

// Find the initial reference.
bus->resolve_initial_reference(
    service_qname,
    ref
);
// Create a proxy and use it
GreeterClient proxy(ref);
proxy.sayHi();
```

The association of your client with a specific endpoint reference is not addressed in your application code. This is specified at runtime instead. The available options are explained in "Accessing Endpoint References" on page 166.

Accessing WSDL and references for clients or servers

These APIs can be used by both clients and servers. For example, typically, clients use the method and servers use the method. However, both application types can use either of these methods.

Accessing WSDL Contracts

When your application calls the Artix bus to access a WSDL contract for a service, the Artix bus uses several available options to access the requested WSDL. Artix tries each resolver mechanism in turn until it finds an appropriate contract, and returns the first result. If one of these is configured with a bad contract URL, no others are called.

Accessing WSDL is a two-step process:

1. You must first use the C++ API to resolve the WSDL (see "Enabling servers to access WSDL" on page 160).
2. You must then use one of the resolvers to configure the WSDL at runtime. These are explained in this section.

Accessing WSDL at runtime

The possible ways of accessing WSDL at runtime are as follows:

1. Command line.
2. Configuration file (`artix.cfg`).
3. Well-known directory.
4. Stub WSDL shared library.

These resolver mechanisms are listed in order of priority, which means that if you configure more than one, those higher up in the list override those lower down. See "Order of precedence for accessing WSDL" on page 165.

Configuring WSDL on the command line

You can configure WSDL by passing URLs as parameters to your application at startup. WSDL URLs passed at application startup take precedence over settings in a configuration file. The syntax for passing in WSDL to any Artix application is:

```
-BUSservice_contract url
```

For example, assuming your application is using the `get_service_contract()` method, you can avoid configuration files by starting your application as follows:

```
./server -BUSservice_contract ../../etc/hello.wsdl
```

This means that the Artix bus parses the URLs that you pass into it on startup. It finds any services that are in this WSDL, and caches them for any users that want WSDL for any of those services.

Parsing WSDL on demand

If you do not want the Artix bus to parse the document until it is needed, you can specify what services are contained in the WSDL, which results in the URL being parsed only on demand. The syntax for this is:

```
-BUSservice_contract {namespace}localpart@url
```

For example, the application would be started as follows:

```
./server -BUSservice_contract
   {http://www.iona.com/demos}HelloWorldService@../../etc/hel
   lo.wsdl
```

Specifying the WSDL URL on startup enables the Artix bus to avoid parsing the WSDL until it is requested.

Configuring WSDL in a configuration file

You can also configure the location of your WSDL in an `artix.cfg` file, using the following syntax.

```
bus:qname_alias:service-name = "{namespace}localpart";
bus:initial_contract:url:service-name = "url";
```

These configuration variables are described as follows:

- `bus:qname_alias:`*`service-name`* enables you to assign an alias or shorthand version of a service QName. You can then use the short version of the service name in other configuration variables. The syntax for the service Qname is "{*`namespace`*}*`localpart`*".
- `bus:initial_contract:url:`*`service-name`* uses the alias defined using `bus:qname_alias` to configure the location of the WSDL contract. The WSDL location syntax is "*`url`*". This can be any valid URL, it does not need to be a local file.

The following example configures a service named `SimpleService`, defined in the `http://www.iona.com/bus/tests` namespace:

```
bus:qname_alias:simple_service =
   "{http://www.iona.com/bus/tests}SimpleService";
bus:initial_contract:url:simple_service =
   "../../etc/simple_service.wsdl";
```

Configuring WSDL in a well-known directory

You can also configure an Artix application to search in a well-known directory when it needs to access WSDL. This enables you to configure multiple documents without explicitly configuring every document on the command line, or in configuration. If you specify a well-known directory, you only need to copy the WSDL documents into this directory before the application uses them.

You can configure the directory location in a configuration file or by passing a command-line parameters to your C++ application.

Configuring a WSDL directory in a configuration file

To set the directory in configuration, use the following variable:

```
bus:initial_contract_dir=[""];
```

The value "." means use the directory from where the application was started. The specified value is a list of directories, which enables you to specify multiple directories.

Configuring a WSDL directory using command-line parameters

If you do not wish to use a configuration file, you can configure the WSDL directory using command line parameters. The command line overrides any settings in a file. The syntax is as follows:

```
-BUSservice_contract_dir directory
```

For example, to configure Artix to look in the current directory, and in the "../../etc" directory, use the following command:

```
server -BUSservice_contract_dir . -BUSservice_contract_dir
   ../../etc/
```

Configuring multiple WSDL directories

You can configure multiple well-known directories for your application to search. However, it is not recommended that you put too many files in the directory.

The more files you put in the directory, the longer it may take to find the contract that you are looking for. The directory search is optimized to first do a quick file scan to see if any of the files potentially contain the target service requested. The documents are not parsed unless a match has been found.

If you use multiple directories, the ordering makes a difference if both directories contain the same service definitions. The WSDL resolvers search the directories in the order that they are configured in.

You can add WSDL documents to the well-known directories after the application has started. The file must only be present in the directory before the application requests it.

Configuring a stub WSDL shared library

It is also possible to encode a WSDL document inside a C++ shared library. Just like in Java, where resources are added to a .jar file, Artix can embed a WSDL document inside a shared library. This enables you to resolve WSDL contracts for Artix services without using a file system or any remote calls.

When a WSDL document is encoded inside a shared library, this is called a *stub WSDL shared library*. Artix provides stub WSDL shared libraries for the following Artix services:

- locator
- session manager
- peer manager
- container

This means that you can deploy these services into environments without using any other resources like WSDL documents. Artix does not provide APIs to enable you to encode your own documents into stub libraries.

Stub WSDL shared libraries are the last resolver mechanisms to be called. If you configure any others, the stub WSDL shared library is not used.

All the Artix stub WSDL libraries contain WSDL endpoints with SOAP HTTP port addresses of `0`. This means that if these versions are used to activate a service, the endpoint is instantiated on a dynamic port. This is the recommended approach for internal services like the container and peer manager.

Order of precedence for accessing WSDL

Because there are several available options for accessing WSDL, Artix searches each resolver in turn for a suitable document. It returns the first successful result to the user.

The order of precedence for accessing WSDL is as follows:

1. Contract passed on the command line.
2. Contract specified in a configuration file.
3. Well-known directory passed on the command line.
4. Well-known directory specified in a configuration file.
5. Stub WSDL shared library.

Example

You have four WSDL contracts that contain a definition for a service named `SimpleService`:

```
one/simple.wsdl
two/simple.wsdl
three/simple.wsdl
four/simple.wsdl
```

1. Configure the following in your configuration file:

```
bus:qname_alias:simple_service =
   "{http://www.iona.com/bus/tests}SimpleService";
bus:initial_contract:url:simple_service =
   "two/simple.wsdl";
bus:initial_contract_dir=["four"];
```

2. Start your server as follows:

```
server -BUSservice_contract_dir three -BUSservice_contract
   one/simple.wsdl
```

The contract in `one/simple.wsdl` is returned to the application because WSDL configured using `-BUSservice_contract` takes precedence over all other sources.

If you start your server as follows:

```
server
```

The contract in `two/simple.wsdl` is returned to the application because the order that the resolvers are called means that the contract specified in a configuration file is the first successful one.

Accessing standard Artix services

For details of accessing WSDL for standard Artix services such as the locator or session manager, see "Accessing Artix Services" on page 170.

Accessing Endpoint References

An *endpoint reference* is an object that encapsulates the endpoint and contract information for a particular WSDL service. A serialized reference is an XML document that refers to a running service instance, and contains a URL pointer to where the service WSDL can be retrieved. You can serialize a reference to any service by deploying it into the Artix container and calling `it_container_admin -publishreference`. Alternatively, you can use APIs to publish an endpoint reference directly.

For example, when your client application uses the Artix bus to look up a endpoint reference using the service QName, it calls the method resolve_initial_reference(). Accessing endpoint references works the same way as accessing WSDL, and you have several options for configuring the reference that the client uses. Like with WSDL contracts, Artix tries each resolver in turn until it gets a successful result or an error. If any of these return null, the core tries the next one. If you have a badly configured reference, the resolver returns an error or exception.

Accessing endpoint references is a two-step process:

1. You must first use the C++ API to resolve the reference (see "Enabling clients to access endpoint references" on page 161).
2. You must then use one of the resolvers to configure the reference at runtime. This is explained in this section.

For details of how to use the Artix container to publish endpoint references for a client, see "Deploying Services in an Artix Container".

Endpoint reference resolver mechanisms

The possible ways of configuring endpoint references at runtime are as follows:

1. Colocated service.
2. C++ programmatic configuration.
3. Command line
4. Configuration file.
5. WDSL contract.

These are listed in order of precedence, so if you configure more than one, those higher up in the list override those lower down. Artix searches each in turn for a suitable match and returns the first successful result.

Using a colocated service

The most convenient place to find a endpoint reference to a service that a client has requested is in the local Artix bus. When the activated service is colocated (available locally in the same process), the client can easily find a local reference to invoke. In this case, the client's `resolve_initial_reference()` method returns a reference to the colocated service.

This is the first resolver that the runtime checks. You can expect resolution to always succeed for services that are activated locally.

Specifying endpoint references

In C++ Artix code, you can register an initial reference programmatically using the Artix bus. You can register a reference in one C++ plug-in that would enable another plug-in to resolve that reference using the bus API.

Artix checks the bus for local services, so it would be unusual for an application to require the programmatic configuration unless it uses multiple buses. You can not programmatically configure a reference in one bus and have it resolved in another.

In addition, you can not activate a service in one bus, and have it resolved in another. If you wish a client in one bus to use a reference from an active service in another bus you should programmatically register the reference from one bus to the next.

For example:

```
\\ C++
QName service_qname("", "SOAPService",
   http://www.iona.com/hello_world_soap_http);

// Activate the service on bus one
HelloWorldImpl servant(bus_one);

WSDLService* contract =
   bus_one->get_service_contract(service_qname);

bus_one->register_servant(
    *contract,
     servant
);

Service_var service = bus_one->get_service(service_qname);

// Register the service reference on bus two
bus_two->register_initial_reference(service->get_endpoint_refer
   ence());
```

Specifying endpoint references on the command line

You can also pass in reference URLs as parameters to the application on startup. Endpoint reference URLs passed to the application on startup take precedence over settings in an `artix.cfg` file. The syntax for passing in a reference to any Artix application is:

```
-BUSinitial_reference url
```

For example, assuming your application is using `resolve_initial_reference()`, you could avoid configuration files by starting your application as follows:

```
./client -BUSinitial_reference ../../etc/hello.xml
```

This means that the Artix bus parses the URLs passed into it on startup. It caches them for any users that request references of this type at runtime.

Parsing endpoint references on demand

If you do not want to parse the reference XML until it is needed, you can specify the service name that the reference maps to. This means that the XML is not parsed until it is first requested. The syntax for this is

```
-BUSinitial_reference {namespace}localpart@url
```

For example, the application is started as follows:

```
./client -BUSinitial_reference
   {http://www.iona.com/demos}HelloWorldService@../../etc/hello
   .xml
```

Specifying endpoint references in a configuration file

You can also specify an endpoint reference in a configuration file. The reference must be serialized in an XML format (for example, output to a file using `itcontainer -publishreference`).

You can use configuration variable syntax to configure a URL or the contents of a serialized reference.

Specifying serialized reference URLs

You can configure the location of your WSDL in an `artix.cfg` file, using the following configuration variable syntax.

```
bus:qname_alias:service-name = "{namespace}localpart";
bus:initial_references:url:service-name = "url";
```

These variables are described as follows:

- bus:qname_alias:*service-name* enables you to assign an alias or shorthand version of a service QName. You can then use the short version of the service name in other configuration variables. The syntax for the service Qname is "{*namespace*}*localpart*".
- bus:initial_contract:url:*service-name* uses the alias defined using bus:qname_alias to configure the location of the endpoint reference. The XML location syntax is "*url*". The URL value can be any valid URL, it does not have to be a local file, but under most circumstances the endpoint reference is local.

The following example configures a service named SimpleService, defined in the http://www.iona.com/bus/tests namespace:

```
bus:qname_alias:simple_service =
   "{http://www.iona.com/bus/tests}SimpleService";
bus:initial_contract:url:simple_service =
   "../../etc/simple_service.xml";
```

Specifying inline references

Instead of configuring a URL, you can also inline the endpoint reference XML in a configuration file. This is similar to configuring CORBA initial references in Orbix, and it effectively hard codes the addressing. This should only be used for static services where you do not expect anything to change (for example, details such as the endpoint address and transport information).

The following is an example inline endpoint reference:

```
bus:qname_alias:simple_service = "{http://www.iona.com/bus/tests}SimpleService";
bus:initial_references:inline:simple_service = "<?xml version='1.0'
   encoding='utf-8'?> ....";
```

The endpoint reference appears on one line in an XML document.

Specifying endpoint references using WSDL

How Artix finds endpoint references is built on how it finds WSDL. When configuring a reference, you can use all the options available for configuring WSDL. When you locate a WSDL document that contains the wsdl:service you are looking for, you can convert it to a reference and return it to the client.

If Artix fails to find a suitable reference using the reference resolver mechanisms, it falls back to those used for WSDL. This is useful in certain scenarios. For example, when you only want to configure well-known Artix services (such as the locator). If you configure the WSDL, both the service and the client can benefit from a single configuration source.

Implications of resolving references using WSDL

When no references are found, Artix calls the WSDL resolver mechanisms. This means that you can rely on WSDL to configure client references.

However, the default WSDL contracts for well-known Artix services have SOAP/HTTP endpoints with a port of zero. For example:

```
<service name="LocatorService">
  <port binding="ls:LocatorServiceBinding" name="LocatorServicePort">
   <soap:address
   location="http://localhost:0/services/locator/LocatorService"/>
 </port>
</service>
```

If you resolve a reference with a port of zero, you get an error when you try to invoke the proxy created from the reference. The exception says that the address is invalid.

These contracts with ports of zero are intended for use by servers rather than clients, and enable servers to run on a dynamic port. Therefore, in general, your client should not rely these contracts. If the server is using this type of contract, you should publish the activated form of the contract, which contains the port assigned dynamically at startup. Your client can then access this activated version of the contract instead.

Further information

For more detailed information on endpoint references, see ***Developing Artix Applications in C++***.

Accessing Artix Services

Artix includes WSDL contracts for all of the services that it ships (for example, the locator and session manager). This section shows the default configuration provided for these services.

Pre-configured WSDL

Artix provides pre-configured aliases and WSDL locations for all of its services. By default, the `artix.cfg` file includes the following entries:

```
# Well known Services QName aliases
bus:qname_alias:container = "{http://ws.iona.com/container}ContainerService";
bus:qname_alias:locator = "{http://ws.iona.com/locator}LocatorService";
bus:qname_alias:peermanager =
   "{http://ws.iona.com/peer_manager}PeerManagerService";
bus:qname_alias:sessionmanager =
   "{http://ws.iona.com/sessionmanager}SessionManagerService";
bus:qname_alias:sessionendpointmanager =
   "{http://ws.iona.com/sessionmanager}SessionEndpointManagerService";
bus:qname_alias:uddi_inquire =
   "{http://www.iona.com/uddi_over_artix}UDDI_InquireService";
bus:qname_alias:uddi_publish =
   "{http://www.iona.com/uddi_over_artix}UDDI_PublishService";
bus:qname_alias:login_service = "{http://ws.iona.com/login_service}LoginService";

bus:initial_contract:url:container =
   "install_root/artix/Version/wsdl/container.wsdl";
bus:initial_contract:url:locator = "install_root/artix/Version/wsdl/locator.wsdl";
bus:initial_contract:url:peermanager =
   "install_root/artix/Version/wsdl/peer-manager.wsdl";
bus:initial_contract:url:sessionmanager =
   "install_root/artix/Version/wsdl/session-manager.wsdl";
bus:initial_contract:url:sessionendpointmanager =
   "install_root/artix/Version/wsdl/session-manager.wsdl";
bus:initial_contract:url:uddi_inquire =
   "install_root/artix/Version/wsdl/uddi/uddi_v2.wsdl";
bus:initial_contract:url:uddi_publish =
   "install_root/artix/Version/wsdl/uddi/uddi_v2.wsdl";
bus:initial_contract:url:login_service =
   "install_root/artix/Version/wsdl/login_service.wsdl";
```

In your application, if you resolve the WSDL or an endpoint reference for any of these services, by default, the WSDL from these values is used. Most of these services are configured to use a port of zero. If you do not want to use the default WSDL for any of these services, you must override the default.

Further information

For more details on the configuration variables for accessing WSDL contracts and endpoint references, see the ***Artix Configuration Reference, C++ Runtime***.

For more examples of accessing WSDL and references in Artix applications, see the following demos:

- `..samples\basic\bootstrap`
- `..samples\advanced\container\deploy_plugin`
- `..samples\advanced\container\deploy_routes`
- `..samples\advanced\locator`
- `..samples\advanced\locator_query`

Accessing Services with UDDI

Artix provides support for Universal Description, Discovery and Integration (UDDI). This chapter explains the basics, and shows how to configure UDDI proxy support in Artix applications. It also shows how to configure jUDDI repository settings.

Introduction to UDDI

A Universal Description, Discovery and Integration (UDDI) registry is a form of database that enables you to store and retrieve Web services endpoints. It is particularly useful as a means of making Web services available on the Internet.

Instead of making your WSDL contract available to clients in the form of a file, you can publish the WSDL contract to a UDDI registry. Clients can then query the UDDI registry and retrieve the WSDL contract at runtime.

Publishing WSDL to UDDI

You can publish your WSDL contract either to a local UDDI registry or to a public UDDI registry.

To publish your WSDL contract, navigate to one of the public UDDI Web sites and follow the instructions there.

Artix UDDI URL format

Artix uses UDDI query strings that take the form of a URL. The syntax for a UDDI URL is as follows:

uddi:*UDDIRegistryEndpointURL*?*QueryString*

The UDDI URL is built from the following components:

- *UDDIRegistryEndpointURL*—the endpoint address of a UDDI registry. This could either be a local UDDI registry (for example, `http://localhost:9000/services/uddi/inquiry`) or a public UDDI registry on the Internet (for example, `http://uddi.ibm.com/ubr/inquiryapi` for IBM's UDDI registry).
- *QueryString*—a combination of attributes used to query the UDDI database for the Web service endpoint data. Currently, Artix only supports the `tmodelname` attribute. An example of a query string is:

  ```
  tmodelname=helloworld
  ```

  Within a query component, the characters `;`, `/`, `?`, `:`, `@`, `&`, `=`, `+`, `,`, and `$` are reserved.

Examples of valid UDDI URLs

```
uddi:http://localhost:9000/services/uddi/inquiry?tmodelname=he
   lloworld
uddi:http://uddi.ibm.com/ubr/inquiryapi?tmodelname=helloworld
```

Initializing a client proxy with UDDI

To initialize a client proxy with UDDI, simply pass a valid UDDI URL string to the proxy constructor.

For example, if you have a local UDDI registry, `http://localhost:9000/services/uddi/inquiry`, where you have registered the WSDL contract from the `HelloWorld` demonstration, you can initialize the `GreeterClient` proxy as follows:

```
// C++
...
IT_Bus::Bus_var bus = IT_Bus::init(argc, argv);

// Instantiate an instance of the proxy
GreeterClient hw("uddi:http://localhost:9000/services/uddi/inquiry?tmodelname=helloworld");

String string_out;

// Invoke sayHi operation
hw.sayHi(string_out);
```

Configuring UDDI Proxy

Artix UDDI proxy service can be used by applications to query endpoint information from a UDDI repository. This section explains how to configure UDDI proxy support for client applications.

C++ configuration

To configure an Artix C++ application for UDDI proxy support, add `uddi_proxy` to the application's `orb_plugins` list. For example:

```
# artix.cfg

my_application_scope {
    orb_plugins = [ ..., "uddi_proxy"];
    ...
};
```

Further information

For more details, see: http://ws.apache.org/juddi/.

Embedding Artix in a BEA Tuxedo Container

Artix can be run and managed by BEA Tuxedo like a native Tuxedo application.

Embedding an Artix Process in a Tuxedo Container

To enable Artix to interact with native BEA Tuxedo applications, you must embed Artix in the Tuxedo container.

At a minimum, this involves adding information about Artix in your Tuxedo configuration file, and registering your Artix processes with the Tuxedo bulletin board.

In addition, you can also enable Tuxedo to bring up your Artix process as a Tuxedo server when running `tmboot`.

This section explains these steps in detail.

Note: A Tuxedo administrator is required to complete a Tuxedo distributed architecture. When deploying Artix in a distributed architecture with other middleware, please also see the documentation for those middleware products.

Procedure

To embed an Artix process in a Tuxedo container, complete the following steps:

1. Ensure that your environment is correctly configured for Tuxedo.
2. You can add the Tuxedo plug-in, `tuxedo`, to your Artix process's `orb_plugins` list.

   ```
   orb_plugins=[... "tuxedo"];
   ```

   However, the tuxedo plug-in is loaded transparently when the process parses the WSDL file.
3. Set `plugins:tuxedo:server` to `true` in your Artix configuration scope.
4. Ensure that the executable for your Artix process is placed in the directory specified in the `APPDIR` entry of your Tuxedo configuration.
5. Edit your Tuxedo configuration's `SERVERS` section to include an entry for your Artix process.

   For example, if the executable of your Artix process is `ringo`, add the following entry in the `SERVERS` section:

   ```
   ringo SVRGRP=BEATLES SVRID=1
   ```

   This associates `ringo` with the Tuxedo group called `BEATLES` in your configuration and assigns `ringo` a server ID of 1. You can modify the server's properties as needed.

6. Edit your Tuxedo configuration's SERVICES section to include an entry for your Artix process.

   While standard Tuxedo servers only require a SERVICES entry if you are setting optional runtime properties, Artix servers in the Tuxedo container require an entry, even if no optional runtime properties are being set. The name entered for the Artix process is the name specified in the serviceName attribute of the Tuxedo port defined in the Artix contract for the process.

   For example, given the port definition shown in Example 23, the SERVICES entry would be personalInfoService.

Example 23: *Sample Service Entry*

```
<service name="personalInfoService">
   <port name="tuxInfoPort"
   binding="tns:personalInfoBinding">
       <tuxedo:server>
           <tuxedo:service name="personalInfoService"/>
       </tuxedo:server>
  </port>
</service>
```

7. If you made the Tuxedo configuration changes in the ASCII version of the configuration, UBBCONFIG, reload the TUXCONFIG with tmload.

When you have configured Tuxedo, it manages your Artix process as if it were a regular Tuxedo server.

Index

A

acknowledgement endpoint URI 132
acknowledgement interval 136
ACTIVATED 88
Adaptive Runtime architecture 9
anonymous URI 132, 145
ANSI C strftime() function 23
application source 130
arbitrary symbols 14
ART 9
Artix 151
artix:endpoint 102
artix:endpoint:endpoint_list 102
artix:endpoint:endpoint_name:wsdl_location 102
artix:endpoint:endpoint_name:wsdl_port 102
artix:interceptors:message_snoop:enabled 36
artix:interceptors:message_snoop:log_level 36
artix.cfg 9, 13, 70
Artix bus pre-filter 31
Artix chain builder 109
Artix container 75
artix_env 97
artix_env script 3
Artix high availability 115
Artix IDL compiler 6
Artix transformer 99
Artix WSDL publishing service 151
ASCII 55
asynchronous acknowledgements 136
auto-demotion of masters 116
avg 47

B

base retransmission interval 134
Berkeley DB 115
binding:artix:client_message_interceptor_list 70
binding:artix:server_message_interceptor_list 70
binding:artix:server_request_interceptor_list 119
binding:server_binding_list 52
-bits 4
browser 153, 155
bus:initial_contract:url:service 111
bus:initial_contract:url:service-name 163
bus:initial_contract_dir 163
bus:initial_references:url:service-name 168
bus:qname_alias:service 111
bus:qname_alias:service-name 163, 168
-BUSCONFIG_ 16, 17
-BUSconfig_dir 6, 95
-BUSconfig_domains_dir 6, 17
-BUSdomain_name 6, 95
-BUSinitial_reference 17, 168
-BUSlicense_file 5, 95
BusLogger 32
-BUSname 11, 95
-BUSname parameter 11
-BUSproduct_dir 5
bus_response_monitor 44
-BUSservice_contract 17, 162
-BUSservice_contract_dir 17, 164

C

C++ compilers 4
C++ debugging 97
canonical 154
chain builder 100, 104, 109
character encoding schema 55
class IT_CONTEXT_ATTRIBUTE_API ClientConfiguration 66
class IT_CONTEXT_ATTRIBUTE_API ServerConfiguration 66
client ID, configuring 46
cluster 116
codeset 55
CODESET_INCOMPATIBLE 60
codeset negotiation 58, 59
Collector 44
collector 48
colocated service 167
command line configuration 16
-compiler 4
configuration
  command line 16
  data type 12
  domain 9
  files 13
  namespace 11
  scope 9
  symbols 14
  variables 12
configuration context 139, 148
constructed types 12
-container 86
container 75, 164
  administration client 78
  persistent deployment 89
  server 77
  service 77

Windows service 93
ContainerService.url 83, 84
content-based routing 106
context 139, 148
ContextContainer 66
contracts 159
Conversion codeset 59
count 47
CreateSequence 129
CreateSequenceResponse 129

D

-d 81
-daemon 83
date format, rolling log file 23
db_dump 118
db_recover 118
db_stat 118
db_verify 118
DEACTIVATED 88
debugging 97
delivery assurance policies 138
delivery assurances 130
dependencies file 79, 80
-deploy 83, 86, 87
-deployfolder 83, 91, 94
deployment descriptor 77, 79
destination 129
-displayname 94
documentation
.pdf format x
updates on the web x
double-byte Unicode 60
dynamic logging 33, 87
dynamic read/write deployment 90

E

EBCDIC 62
echoString 61
echoVoid 61
election protocol 116
EMS, definition 43
encodings 55
endpoint references 151, 159, 160, 166
Enterprise Management Systems 43
Enterprise Object Identifier 42
-env 83, 95
environment variables 93
ERROR 21
EUC-JP 56
event_log:filters 19, 24, 70, 121
event_log:filters:artix:pre_filter 31
event_log:filter_sensitive_info 21
event_log:log_service_names:active 32
event_log:log_service_names:services 32
ExactlyOnceConcurrent 130, 138
ExactlyOnceInOrder 130, 138
ExactlyOnceReceivedOrder 130, 138
exponential backoff for retransmission 134
exponential backoff interval 130
Extended Binary Coded Decimal Interchange Code 62
Extensible Stylesheet Language Transformations 99

F

FATAL_ERROR 21
-file 81, 86
filters 25
fixed:binding 60
fixed:body 61
four-byte Unicode 60

G

get_logging_config() 33
-getlogginglevel 33, 87
get_service_contract() 160, 162

H

ha_conf 122, 125
hard coded WSDL 159
-help 81, 83
high availability 115
clients 123
locator 121
high water mark 140
-host 86
hostname format 154
HTML menu 156
HTTP GET 157
HTTP POST 157
HTTP trace logging 24
HTTP transport 157

I

i18n-context.xsd 64, 66
i18n_interceptor 70
IANA 42, 56
IBM Tivoli integration 43
IBM WebSphere MQ, internationalization 62
ideograms 55
idl.cfg 6
IDL configuration file 6
InboundCodeSet 63
include statement 13
INFO_ALL 20
INFO_HIGH 20
INFO_LOW 20
INFO_MEDIUM 20
INITIALIZED 88
initial sender 129
initial_threads 140
inline references 169
int 47
intercept_dispatch() 66
intercept_invoke() 66
internationalization
CORBA 58
MQ 62
SOAP 57

Internet Assigned Number Authority 56
Internet Assigned Numbers Authority 42
Interoperable Object Reference 51
IOR 51
ipaddress 154
ISO-2022-JP 56
ISO 8859 55
ISO-8859-1 56
ITArtixContainer 93
IT_ARTIXENV 7
IT_ARTIX_ENV_SET 7
IT_ATLI2_IOP 29
IT_ATLI2_IP 29
IT_ATLI2_IP_TUNNEL 29
IT_ATLI_TLS 29
IT_BUS 25
IT_Bus::Exception 135
IT_Bus::init() 11, 16, 22
IT_BUS.BINDING 25
IT_BUS.BINDING.COLOC 26
IT_BUS.BINDING.CORBA 26
IT_BUS.BINDING.CORBA.CONTEXT 26
IT_BUS.BINDING.FIXED 26
IT_BUS.BINDING.HTTP 26
IT_BUS.BINDING.SOAP 26
IT_BUS.BINDING.SOAP_COMMON 26
IT_BUS.BINDING.TAGGED 26
IT_BUS.CORE 26
IT_BUS.CORE.CONFIG 26
IT_BUS.CORE.CONTEXT 26
IT_BUS.CORE.INITIAL_REFERENCE 26
IT_BUS.CORE.PLUGIN 26
IT_BUS.CORE.RESOURCE_RESOLVER 26
IT_BUS.FOUNDATION.AFC 26
IT_BUS.FOUNDATION.CONTEXT_LIBRARY 26
IT_BUS.I18N.INTERCEPTOR 26
IT_BUS.INTEGRATION.AP_NANO_AGENT 26
IT_BUS.INTEGRATION.CA_WSDM_OBSERVER 26
IT_BUS.JNI.GENERIC_PLUGIN 26
IT_BUS.JNI.JBUS 26
IT_BUS.JNI.JBUS.TRANSACTION 26
IT_BUS.JNI.JNI_UTIL 26
IT_BUS.JNI.TRANSACTION 26
IT_BUS.JVM_MANAGER 26
IT_BUS.LOGGING 26
IT_BUS.LOGGING.LOG4J 26
IT_BUS.LOGGING.RESPONSE_TIME 26
IT_BUS.LOGGING.SNMP 26
IT_BUS.MANAGEMENT 27
IT_BUS.MESSAGING_PORT 27
IT_BUS.SERVICE 27
IT_BUS.SERVICE.ACTIVATOR.REGISTRY 27
IT_BUS.SERVICE.CHAIN 27
IT_BUS.SERVICE.CONTAINER 27
IT_BUS.SERVICE.DB 27
IT_BUS.SERVICE.DB.ENV 27
IT_BUS.SERVICE.DB.REPLICA.IMPL 27
IT_BUS.SERVICE.DB.REPLICA.MGR 27
IT_BUS.SERVICE.DB.REPLICA.MONITOR 27
IT_BUS.SERVICE.DB.REPLICA.SYNC 27
IT_BUS.SERVICE.LOCATOR 27
IT_BUS.SERVICE.PEER_MGR 27
IT_BUS.SERVICE.ROUTING 27
IT_BUS.SERVICE.SECURITY 27
IT_BUS.SERVICE.SECURITY.CERT_VALIDATOR 27
IT_BUS.SERVICE.SECURITY.LOGIN_SERVICE.CLIENT 27
IT_BUS.SERVICE.SECURITY.LOGIN_SERVICE.SERVICE 27
IT_BUS.SERVICE.SECURITY.SECURITY_INTERCEPTOR 27
IT_BUS.SERVICE.SECURITY.WSS 27
IT_BUS.SERVICE.SESSION_MGR 27
IT_BUS.SERVICE.WSDL_PUBLISH 27
IT_BUS.SERVICE.XSLT 27
IT_BUS.TRANSACTIONS.OTS 27
IT_BUS.TRANSACTIONS.WSAT 27
IT_BUS.TRANSACTIONS.XA 27
IT_BUS.TRANSPORT.HTTP 27
IT_BUS.TRANSPORT.MQ 28
IT_BUS.TRANSPORT.STUB_TRANSPORT 28
IT_BUS.TRANSPORT.TUNNELL 28
IT_BUS.TRANSPORT.TUXEDO 28
IT_BUS.VERSION 28
IT_BUS.WSRM 28
IT_BUS.WSRM_DB 28
IT_BUS.XA_SWITC 28
IT_COBOL_PLI 29
IT_CODESET 29
IT_CONFIG_DIR 6
IT_CONFIG_DOMAINS_DIR 6
IT_CONNECTION_FILTER 29
it_container 77, 82
it_container_admin 33, 78, 85, 166
IT_CORE 29
IT_CSI 29
IT_DOMAIN_NAME 6
IT_GenericSecurityToolkit 29
IT_GIOP 29
IT_GSP 29
IT_HTTP 29
IT_HTTPS 29
IT_IDL_CONFIG_FILE 6
IT_IIOP 29
IT_IIOP_TLS 29
IT_INIT_BUS_LOGGER_MEM 32
IT_LICENSE_FILE 5
IT_LICENSING 29
IT_Logging::LogStream 42
IT_MESSAGING 29
IT_MGMT_LOGGING 29
IT_OBJECT_KEY_REPLACER 29
IT_OTS 29
IT_OTS_LITE 29
IT_POA 29
IT_POA_LOCATOR 30
IT_PRODUCT_DIR 5, 93

IT_REQUEST_LOGGER 30
it_response_time_logger 52
IT_SCHANNEL 30
IT_SECURITY 30
IT_TLS 30
IT_WORKQUEUE 30
IT_WSDLGEN_CONFIG_FILE 6
IT_WSRM 28
IT_XA 30

J

Japanese EUC 55
Japanese ISO 2022 55
Java API for XML-Based Web Services vii
Java configuration 45
JAVA_HOME 5
JAX-WS vii

L

Latin-1 55
life cycle message formats 48
-listservices 86, 88, 89
LocalCodeSet 63
local_log_stream 19
locator 164
locator, load balancing 121
log date format 23
log file, rolling 23
log file interpreter 43
logging 121
  API 33
  inheritance 35
  message severity levels 20
  passwords 21
  per bus 32
  precision 25
  service-based 32
  set filters for subsystems 25
  silent 35
logging collector 48
LoggingConfig 33
logging levels
  getting 33, 87
  setting 19, 33, 34, 87
logging message formats 47
LOG_INHERIT 35
LOG_SILENT 35
low water mark 140

M

mark_as_write_operations() 127
master-slave replication 115
max 47
maximum messages in RM sequence 137
maximum unacknowledged messages threshold 135
max queue size 140
MEP 146
Message Exchange Pattern 145, 147
message part element 106
MESSAGE_SNOOP 30
message snoop 35
MIB, definition 38
Microsoft Visual C++ 97
min 47
minority master 120
MQ, internationalization 62
multi-homed 154

N

namespace 47
namespace IT_ContextAttributes 66
naming conventions 92
native codeset 58
NCS 58
NOT_INITIALIZED 88

O

oneway calls 140
operation 47
oph 47
orb_plugins 44, 102, 104, 110, 141
OSF CodeSet Registry 56
OutboundCodeSet 63

P

part element 106
passwords
  logging 21
PATH 94
peer manager 164
performance logging 43
performance logging collector 48
persistence 140
persistent database 117
persistent deployment 89
PersistentMap 117
per-thread RM session 138
-pluginImpl 81
-pluginName 81
plugins:artix:db:allow_minority_master 120
plugins:artix:db:home 141
plugins:artix:db:iiop:port 120
plugins:artix:db:priority 119
plugins:artix:db:replicas 118
plugins:bus_response_monitor:type 45
plugins:chain:endpoint:operation:service_chain 111
plugins:chain:endpoint:operation_list 111
plugins:chain:endpoint_name:operation_name:service_chain 105
plugins:chain:init_on_first_call 112
plugins:chain:servant_list 111
plugins:codeset:char:ccs 59
plugins:codeset:char:ncs 58
plugins:codeset:wchar:ccs 59
plugins:codeset:wchar:ncs 58
plugins:container:deployfolder 91
plugins:container:deployfolder:readonly 91
plugins:ha_conf:random:selection 128

plugins:ha_conf:strategy 128
plugins:it_response_time_collector:filename 45
plugins:it_response_time_collector:server-id 45
plugins:local_log_stream:buffer_file 24
plugins:local_log_stream:filename_date_format 23
plugins:local_log_stream:log_thread_id 25
plugins:local_log_stream:precision_logging 25
plugins:local_log_stream:rolling_file 23
plugins:locator:persist_data 121
plugins:locator:selection_method 122
plugins:messaging_port:base_replyto_url 148
plugins:messaging_port:supports_wsa_2005_mep 147
plugins:messaging_port:supports_wsa_mep 147
plugins:messaging_port:wsrm_enabled 131
plugins:remote_log_receiver:iiop addr_list 51
plugins:remote_log_receiver:ior_filename 51
plugins:remote_log_receiver:log_filename 51
plugins:remote_log_receiver:prerequisite_plugins 51
plugins:snmp_log_stream:community 41
plugins:snmp_log_stream:oid 41
plugins:snmp_log_stream:port 41
plugins:snmp_log_stream:server 41
plugins:snmp_log_stream:trap_type 41
plugins:soap:encoding 57
plugins:wsdl_publish:hostname 154
plugins:wsdl_publish:processor 154
plugins:wsdl_publish:publish_port 153
plugins:wsrm:acknowledgement_interval 137
plugins:wsrm:acknowledgement_uri 132
plugins:wsrm:base_retransmission_interval 134
plugins:wsrm:delivery_assurance_policy 138
plugins:wsrm:disable_exponential_backoff_retransmission_interval 134
plugins:wsrm:enable_per_thread_sequence_scope 139
plugins:wsrm:max_messages_per_sequence 137
plugins:wsrm:max_retransmission_attempts 136
plugins:wsrm:max_unacknowledged_messages_threshold 135
plugins:wsrm:thread_pool:high_water_mark 140
plugins:wsrm:thread_pool:initial_threads 140
plugins:wsrm:thread_pool:low_water_mark 140
plugins:wsrm:thread_pool:max_queue_size 140
plugins:wsrm:thread_pool:stack_size 140
plugins:xmlfile_log_stream:buffer_file 24
plugins:xmlfile_log_stream:filename 22
plugins:xmlfile_log_stream:filename_date_format 23
plugins:xmlfile_log_stream:rolling_file 23
plugins:xmlfile_log_stream:use_pid 22
plugins:xslt:endpoint_name:operation_map 103
plugins:xslt:endpoint_name:trace_filter 106
plugins:xslt:endpoint_name:use_element_name 106
plugins:xslt:servant_list 102
-pluginType 81
-pluginURL 81
policies:at_http:server_address_mode_policy:publish_hostname 154
policies:http:trace_requests:enabled 24
policies:https:trace_requests:enabled 24
policies:soap:server_address_mode_policy:publish_hostname 154
-port 83, 86, 94
port 47
precedence, finding references 167
precedence, finding WSDL 165
precision logging 25
pre-filter 31
preprocessing 154
preserve 4
primitive types 12
programmatic configuration 167
-propagate 34
-provider 81
proxy 148
-publish 83
-publishreference 86, 87, 168
-publishurl 86, 87, 88
-publishwsdl 86, 87

Q

QName 160
QueryString 173
-quiet 81

R

random endpoint selection 128
read-only deployment 90
references 151, 159
remote logger daemon 48
remote logging 48
remote_log_receiver 51
-removeservice 86, 92
replica group 124
replica priorities 119
replicas, minimum number 116, 120
replicated services 115

reply-to endpoint 148
request_forwarder 117
resolve_initial_reference() 161, 167
Response monitor 44
retransmission 134
rolling log file 23
running 48

S

secondary hostname 154
SequenceAcknowledgement 130
serialized reference 168
servant registration 155
server ID 47, 48
server-id 46
server ID, configuring 45
-service 81, 86
service 47
-service install 94
Services dialog 95
service states 88
-service uninstall 96
session manager 164
setInboundCodeSet 66
setLocalCodeSet 66
setlocale() 58
-setloggginglevel 33, 87
setOutboundCodeSet 66
Shift JIS 55
Shift_JIS 56
-shutdown 86, 89
SHUTDOWN_COMPLETE 88
SHUTDOWN_PENDING 88
shutting_down 48
SNMP
  definition 37
  Management Information Base 38
snmp_log_stream 41
source 129
stack size 140
starting_up 48
-startservice 86
stateless servers 127
status 48
-stopservice 86, 89
strftime() 23
stub WSDL shared library 164
-svcName 94
symbols 14

T

TCS 59
thread pool 139
Tivoli integration 43
Tivoli Task Library 43
tmodelname 173
trace logging 24
transformer 99
transmission codeset 58, 59

U

UCS-2 60
UCS-4 60
UDDI 173
uddi_proxy 174
UDDIRegistryEndpointURL 173
ultimate receiver 129
unacknowledged messages 135
Unicode 56
unqualified 154
US-ASCII 56
UTF-16 56, 57
UTF-8 56

V

-verbose 4, 81
-version 81, 83

W

WARNING 20
web browser 153, 155
Web service chain builder 100, 104, 109
Web Services Inspection Language 157
Web Services Reliable Messaging 129
WebSphere MQ, internationalization 62
Windows service 93
work queue 88
wsa:MessageId 145
wsa:RelatesTo 145
wsa:ReplyTo 145
wsa:replyTo 134
wsa:To 145
WS-Addressing 145
WS-Addressing Message Exchange Pattern 146
ws_chain 110
wsdd 80
WSDL contracts 159, 160
wsdlgen.cfg 6
WSDL preprocessing 154
wsdl_publish 151
WSDL publishing service 151
wsdltocpp 80
-wsdlurl 81
WSIL 157
WS-ReliableMessaging 129
WS-RM 129
wsrm 131, 141
wsrm:AckRequested 135
wsrm:AcksTo 129, 132, 134, 136
wsrm:SequenceTerminated 135
WS-RM acknowledgement endpoint URI 132
wsrm_db 141
WS-RM persistence 140
WS-RM threading 139
WS-S 21

X

xmlfile_log_stream 19
XSLT service 99