OpenEdge® Getting Started: Application and Integration Services
© 2014 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

Business Making Progress, Corticon, DataDirect (and design), DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, Easy!, Fathom, Making Software Work Together, OpenEdge, Powered by Progress, Progress, Progress Control Tower, Progress RPM, Progress Software Business Making Progress, Progress Software Developers Network, Rollbase, RulesCloud, RulesWorld, SequeLink, SpeedScript, Stylus Studio, and WebSpeed are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries. AccelEvent, AppsAlive, AppServer, BusinessEdge, DataDirect Spy, DataDirect SupportLink, , Future Proof, High Performance Integration, Modulus, OpenAccess, Pacific, ProDataSet, Progress Arcade, Progress Pacific, Progress Profiles, Progress Results, Progress RFID, Progress Progress Software, ProVision, PSE Pro, SectorAlliance, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, WebClient, and Who Makes Progress are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

Please refer to the Release Notes applicable to the particular Progress product release for any third-party acknowledgements required to be provided in the documentation associated with the Progress product.

The Release Notes can be found in the OpenEdge installation directory and online at:

For the latest documentation updates see OpenEdge Product Documentation on Progress Communities: (https://community.progress.com/technicalusers/w/openedgegeneral/1329.openedge-product-documentation-overview.aspx).

December 2014

Last updated with new content: Release 11.1.0
## Table of Contents

### Preface

- Purpose...........................................................................................................11
- Audience........................................................................................................12
- Organization.....................................................................................................12
- Using this manual..............................................................................................13
  - References to ABL compiler and run-time features........................................13
  - References to ABL data types..........................................................................13
  - Typographical conventions.............................................................................14
  - OpenEdge messages.........................................................................................15
  - Obtaining more information about OpenEdge messages...............................16

### Chapter 1: Application and Integration Services in OpenEdge

- What are application and integration services?................................................17
  - Services and service-oriented architecture.....................................................18
    - Interaction with a service............................................................................18
    - Services—the basis for SOA.........................................................................18
  - Application and integration services in OpenEdge.........................................19
    - Application service support.........................................................................19
    - Integration service support..........................................................................19
    - Services foundations in OpenEdge..............................................................20
  - What is distributed application computing?....................................................20
    - Partitioning of an application......................................................................20
    - N-tier model.................................................................................................21
  - A comparison of computing models...............................................................21
    - Traditional two-tier model.........................................................................22
    - OpenEdge distributed application computing models..................................22
      - Logical three-tier model..........................................................................23
      - Physical n-tier model..............................................................................23

### Chapter 2: OpenEdge Application Server Architecture

- OpenEdge Application Server overview...........................................................25
- Understanding the AppServer............................................................................27
  - AppServer architecture...................................................................................28
  - Further information on the AppServer............................................................29
- AppServer clients overview...............................................................................29
  - ABL clients....................................................................................................30
  - OpenEdge Open Clients.................................................................................30
  - Web service clients.......................................................................................30
Chapter 7: OpenEdge Messaging and ESB

OpenEdge Adapter for SonicMQ

ABL-JMS API

OpenEdge Adapter for SonicMQ architecture

OpenEdge Adapter for SonicMQ ClientConnect

OpenEdge Adapter for SonicMQ ServerConnect

OpenEdge Adapter for SonicMQ BrokerConnect

OpenEdge Adapter for Sonic ESB

Native Invocation methodology

Declarative vs. non-declarative approach

Annotations

.esboe files

Web Service Invocation methodology

WSM files

WSDL files

Sonic Management Console

Sonic ESB container

SonicMQ container

SonicMQ Broker

AppServer

OpenEdge services—usage and management

Chapter 8: WebSpeed

Introduction to WebSpeed

Advantages of WebSpeed

WebSpeed components

WebSpeed Messenger

NameServer

WebSpeed Transaction Server

WebSpeed broker

WebSpeed agent

WebSpeed Workshop

AppBuilder

WebTools
SpeedScript...................................................................................................................104
JavaScript .....................................................................................................................104
Progress Developer Studio for OpenEdge........................................................................105
Configuration and management utilities..........................................................................105
OpenEdge Management or OpenEdge Explorer............................................................105
Command-line utilities..................................................................................................105
SpeedStart.....................................................................................................................106
How WebSpeed handles a request....................................................................................106

Index................................................................................................................................109
For details, see the following topics:

- Purpose
- Audience
- Organization
- Using this manual
- Typographical conventions
- OpenEdge messages

**Purpose**

This manual is the starting point to learn about OpenEdge® support for service-oriented architectures (SOA). It provides a comprehensive overview of the many OpenEdge product components and features that support application development and deployment in an SOA environment. These products and features include:

- The OpenEdge AppServer™
- The Open Client Toolkit
- The AppServer Internet Adapter
- Web Services in OpenEdge
- Messaging and ESB in OpenEdge
- WebSpeed®
All of these product components and features share one of the following functions:

• They comprise part of the OpenEdge Application Server product set, which provides the core support for application and integration services.

• They play a unique role within OpenEdge application and integration services to help you develop and deploy applications as part of an SOA.

This overview includes an introduction to services, services-oriented architectures, and distributed computing, a broad overview of the OpenEdge Application Server architecture, and a detailed introduction and architecture for each OpenEdge component and feature to help you decide if and how to use it. It also provides references to further documentation on how to use these components and features for application development, deployment, and administration.

Audience

This manual is for anyone who needs to identify and compare the options for working with application and integration services in OpenEdge. It is also for application developers and administrators who are getting started with developing or deploying AppServer application services and developing AppServer clients, including development and deployment of non- ABL clients, Web services and Web service clients, and exposing ABL procedures to Sonic ESB®, developing WebSpeed application services for access by Web browser clients; and implementing Java™ Message Service (JMS) messaging in OpenEdge applications using the SonicMQ® JMS backbone.

Organization

Application and Integration Services in OpenEdge on page 17
Defines what services and service-oriented architectures mean and provides an overview of distributed application computing in OpenEdge.

OpenEdge Application Server Architecture on page 25
Provides an overview of the OpenEdge Application Server architecture and components, focussing on its two core application server components, the AppServer and WebSpeed.

AppServer for OpenEdge Applications on page 37
Describes basic AppServer operation and its fundamental features and components.

OpenEdge Open Clients on page 55
Provides an overview of the Open Client architecture and the tools for developing Open Clients in OpenEdge, allowing non-ABL clients to access the AppServer.

AppServer Internet Adapter on page 59
Provides an overview of the AppServer Internet Adapter (AIA) and the requirements for using it to allow clients to access the AppServer over the Internet.

Web Services—Architecture and Tools on page 67
Provides an overview of OpenEdge support for Web services, both for developing Web services using ABL and for accessing industry Web services using ABL.

OpenEdge Messaging and ESB on page 89
Provides an overview of the OpenEdge Adapter for Sonic MQ and the OpenEdge Adapter for Sonic ESB. The adapters provide support for JMS messaging in ABL and support for ABL procedures to be incorporated into ESB processes in Sonic ESB.

WebSpeed on page 101

Describes basic WebSpeed operation and its fundamental features and components.

Using this manual

OpenEdge provides a special purpose programming language for building business applications. In the documentation, the formal name for this language is ABL (Advanced Business Language). With few exceptions, all keywords of the language appear in all UPPERCASE, using a font that is appropriate to the context. All other alphabetic language content appears in mixed case.

For the latest documentation updates see the OpenEdge Product Documentation Overview page on Progress Communities:


References to ABL compiler and run-time features

ABL is both a compiled and an interpreted language that executes in a run-time engine. The documentation refers to this run-time engine as the ABL Virtual Machine (AVM). When the documentation refers to ABL source code compilation, it specifies ABL or the compiler as the actor that manages compile-time features of the language. When the documentation refers to run-time behavior in an executing ABL program, it specifies the AVM as the actor that manages the specified run-time behavior in the program.

For example, these sentences refer to the ABL compiler's allowance for parameter passing and the AVM's possible response to that parameter passing at run time: "ABL allows you to pass a dynamic temp-table handle as a static temp-table parameter of a method. However, if at run time the passed dynamic temp-table schema does not match the schema of the static temp-table parameter, the AVM raises an error." The following sentence refers to run-time actions that the AVM can perform using a particular ABL feature: "The ABL socket object handle allows the AVM to connect with other ABL and non-ABL sessions using TCP/IP sockets."

References to ABL data types

ABL provides built-in data types, built-in class data types, and user-defined class data types. References to built-in data types follow these rules:

- Like most other keywords, references to specific built-in data types appear in all UPPERCASE, using a font that is appropriate to the context. No uppercase reference ever includes or implies any data type other than itself.

- Wherever integer appears, this is a reference to the INTEGER or INT64 data type.

- Wherever character appears, this is a reference to the CHARACTER, LONGCHAR, or CLOB data type.

- Wherever decimal appears, this is a reference to the DECIMAL data type.
• Wherever numeric appears, this is a reference to the INTEGER, INT64, or DECIMAL data type.

References to built-in class data types appear in mixed case with initial caps, for example, Progress.Lang.Object. References to user-defined class data types appear in mixed case, as specified for a given application example.

Typographical conventions

This manual uses the following typographical and syntax conventions:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bold</strong></td>
<td>Bold typeface indicates commands or characters the user types, provides emphasis, or the names of user interface elements.</td>
</tr>
<tr>
<td><em>Italic</em></td>
<td>Italic typeface indicates the title of a document, or signifies new terms.</td>
</tr>
<tr>
<td>SMALL, BOLD CAPITAL LETTERS</td>
<td>Small, bold capital letters indicate OpenEdge key functions and generic keyboard keys; for example, GET and CTRL.</td>
</tr>
<tr>
<td>KEY1+KEY2</td>
<td>A plus sign between key names indicates a simultaneous key sequence: you press and hold down the first key while pressing the second key. For example, CTRL+X.</td>
</tr>
<tr>
<td>KEY1 KEY2</td>
<td>A space between key names indicates a sequential key sequence: you press and release the first key, then press another key. For example, ESCAPE H.</td>
</tr>
</tbody>
</table>

**Syntax:**

- **Fixed width**
  - A fixed-width font is used in syntax, code examples, system output, and file names.
- **Fixed-width italics**
  - Fixed-width italics indicate variables in syntax.
- **Fixed-width bold**
  - Fixed-width bold italic indicates variables in syntax with special emphasis.
- **UPPERCASE fixed width**
  - ABL keywords in syntax and code examples are almost always shown in upper case. Although shown in uppercase, you can type ABL keywords in either uppercase or lowercase in a procedure or class.
- **Period (.) or colon (:)**
  - All statements except DO, FOR, FUNCTION, PROCEDURE, and REPEAT end with a period. DO, FOR, FUNCTION, PROCEDURE, and REPEAT statements can end with either a period or a colon.

- **[ ]**
  - Large brackets indicate the items within them are optional.
- **[]**
  - Small brackets are part of ABL.
OpenEdge messages

OpenEdge displays several types of messages to inform you of routine and unusual occurrences:

- **Execution messages** inform you of errors encountered while OpenEdge is running a procedure; for example, if OpenEdge cannot find a record with a specified index field value.

- **Compile messages** inform you of errors found while OpenEdge is reading and analyzing a procedure before running it; for example, if a procedure references a table name that is not defined in the database.

- **Startup messages** inform you of unusual conditions detected while OpenEdge is getting ready to execute; for example, if you entered an invalid startup parameter.

After displaying a message, OpenEdge proceeds in one of several ways:

- Continues execution, subject to the error-processing actions that you specify or that are assumed as part of the procedure. This is the most common action taken after execution messages.

- Returns to the Procedure Editor, so you can correct an error in a procedure. This is the usual action taken after compiler messages.

- Halts processing of a procedure and returns immediately to the Procedure Editor. This does not happen often.

- Terminates the current session.

OpenEdge messages end with a message number in parentheses. In this example, the message number is 200:

```
** Unknown table name table. (200)
```

If you encounter an error that terminates OpenEdge, note the message number before restarting.
Obtaining more information about OpenEdge messages

In Windows platforms, use OpenEdge online help to obtain more information about OpenEdge messages. Many OpenEdge tools include the following Help menu options to provide information about messages:

- Choose Help > Recent Messages to display detailed descriptions of the most recent OpenEdge message and all other messages returned in the current session.
- Choose Help > Messages and then type the message number to display a description of a specific OpenEdge message.
- In the Procedure Editor, press the HELP key or F1.

On UNIX platforms, use the OpenEdge pro command to start a single-user mode character OpenEdge client session and view a brief description of a message by providing its number.

To use the pro command to obtain a message description by message number:

1. Start the Procedure Editor:

   ```
   OpenEdge-install-dir/bin/pro
   ```

2. Press F3 to access the menu bar, then choose Help > Messages.
3. Type the message number and press ENTER. Details about that message number appear.
4. Press F4 to close the message, press F3 to access the Procedure Editor menu, and choose File > Exit.
This chapter defines and introduces the products, components, and technologies available for developing application and integration services in OpenEdge®. It also describes how distributed application computing forms the basis for application and integration services, and how this form of computing compares to the client/server computing models that have preceded it. For details, see the following topics:

- What are application and integration services?
- What is distributed application computing?
- A comparison of computing models

What are application and integration services?

Application and integration services depend first and foremost on the concept of a service. A service is a discrete, typically coarse-grained function that solves a particular application problem and is available for use by a variety of applications (clients of the service) within a particular deployment environment.
Services and service-oriented architecture

Exactly what a service is and what it does depends very much on the particular application problem that it solves and the environment in which the service is deployed. It also depends on how accessible and available the service happens to be to potential clients that might make use of it in that deployment environment. Typically, service accessibility and availability depends on the network interfaces that it supports within its environment, which could be anything from a small local area network (LAN) to a global wide-area network (WAN). Services might support protocols limited to an intranet or support the widest possible available protocols on the Internet.

Interaction with a service

In any case, a client typically uses a service by:

1. Identifying a problem to be solved
2. Locating a service to solve that problem
3. Establishing a relationship (connection or binding) to the service that it locates
4. Invoking one or more requests on the service designed to help solve the problem
5. Terminating the relationship with (disconnecting or unbinding from) the service

An example of a service might be an inventory service that supports various requests to query, update, and otherwise manage a particular inventory, or even a range of inventories, depending on the generality of the inventory function.

Services—the basis for SOA

A key advantage of services is that they can be used as the basis for developing a service-oriented architecture (SOA). An SOA is an application development, deployment, and management infrastructure whereby the major functions of the application are delivered as services. Development occurs using service-based development tools, and deployment and management infrastructure is focused on deploying and managing business functions as services.

IT environments typically use services for two basic purposes:

- To encapsulate business functionality that can be re-used to solve the same problem in more than one application context. An example of this might be a single calendar service that can be accessed by a strategic planning system, a project scheduling system, and an airline reservation system.

- To allow the features of one application domain to be accessible and usable by another application domain—that is, to integrate multiple application domains into what is effectively a single system. A typical situation for this type of service involve two businesses, where one has acquired the other. In this situation, the service might allow inventory and accounting for the acquired business to be accessed and managed from the inventory and accounting systems of the business that has acquired it.

Thus those services that facilitate solutions to particular application problems, such as a calendar service, fall into the category of application services. Those services that address the problem of integrating separate applications into a larger application infrastructure, such as joining the accounting systems of two separate businesses, fall into the category of integration services. OpenEdge provides products and components that support both categories of service in the development of an SOA.
Application and integration services in OpenEdge

Service and SOA support in OpenEdge targets different levels of granularity and infrastructure, with varying capacities to support application and integration services.

Application service support

The OpenEdge Application Server and its components provide the core technologies for developing application services in OpenEdge. Some components of the OpenEdge Application Server (for example, OpenEdge Web services) also support the development of integration services.

The two core technologies that form the basis of application service support in OpenEdge include the OpenEdge AppServer™ and WebSpeed®, both of which support the development of application services using Advanced Business Language (ABL) and the WebSpeed variant, SpeedScript®. The AppServer supports OpenEdge sessions that run ABL procedures (remote procedures) that are invoked from requests sent by client programs running in separate application sessions that are often running on separate machines. In response, the AppServer returns the results of each remote procedure execution as output parameters or return values as though executed in the same session as the client. As such, the remote procedures available on a single AppServer platform combine to create an application service that is available to clients designed to access it.

WebSpeed supports SpeedScript sessions that run SpeedScript procedures (called Web objects) that are invoked from HTTP Get and Post requests sent over the Internet from a Web browser (HTTP client) and passed by a Web server to the appropriate WebSpeed session. In response, WebSpeed returns the results of each Web object execution as an HTML page that can be interpreted and displayed in the Web browser. As such, the Web objects available on a single WebSpeed platform combine to create an application service that is available to any HTTP client with access to the Web site that handles the WebSpeed requests. Thus, WebSpeed is specialized to provide application services to Web browser clients.

Additional components, such as Web services tools and the Open Client Toolkit, enhance the accessibility and availability of application services provided by these core technologies. For more information on Web services tools, see Integration service support on page 19. The Open Client Toolkit supports the Web services tools and also supports the development of non-ABL clients that can directly access AppServer application services. For more information on Open Client support, see OpenEdge Open Clients on page 55.

For more information on how the OpenEdge Application Server supports application service development, deployment, and management, see OpenEdge Application Server Architecture on page 25.

Integration service support

The OpenEdge Adapter for SonicMQ®, Web services tools, and the OpenEdge Adapter for Sonic ESB® provide the core technologies for developing integration services in OpenEdge. The SonicMQ Adapter allows an OpenEdge (or SpeedScript) application to become a client of the Java™ Message Service (JMS) backbone provided by SonicMQ. As a JMS client, the OpenEdge application can exchange messages in a variety of formats with other OpenEdge or non-OpenEdge applications that also use SonicMQ as their JMS backbone. For more information on the OpenEdge Adapter for SonicMQ, see OpenEdge Messaging and ESB on page 89. For more information on SonicMQ, see the SonicMQ product documentation.
The Web services tools support the development and deployment of Web services based on existing AppServer application services that can be consumed by any Web service client. Relying largely on the Open Client Toolkit for development and the Web Services Adapter (WSA) for deployment, OpenEdge Web services thus provide a means to integrate AppServer application services with any application domain that supports consuming Web services. The Web services tools also include support for the ability of OpenEdge applications to consume industry Web services in a manner very similar to accessing AppServer application services. Thus, OpenEdge applications can integrate application services developed on most any other platform, such as Java or Microsoft .NET. For more information on OpenEdge support for Web services, see Web Services—Architecture and Tools on page 67.

The OpenEdge Adapter for Sonic ESB allows ABL procedures to be accessed from the Sonic Enterprise Service Bus (ESB). ABL procedures are exposed in one of two ways: the Native Invocation methodology, or the Web Service Invocation methodology. When exposed to the Enterprise Service Bus, ABL procedures can then be integrated with other services and made available to a wide variety of application types and platforms using the integration technology provided by Sonic ESB. For more information on the OpenEdge Adapter for Sonic ESB, see OpenEdge Messaging and ESB on page 89. For more information on Sonic ESB, see the Sonic ESB product documentation.

Services foundations in OpenEdge

The whole basis for services and SOA support in OpenEdge is in its support for distributed application computing. The following sections describe this support and compare it to the client/server computing models that have preceded it.

What is distributed application computing?

_Distributed application computing_ is an architectural approach to designing application software that runs across a network. It maximizes your processing capabilities by allowing you to distribute portions of your business application throughout your enterprise. By emphasizing a flexible, modular use of computer resources, your enterprise can potentially gain greater performance returns, realize the benefits of customized deployment strategies, and provide a security model to ensure the integrity of your application data.

Two related and important aspects of distributed application computing include the:

- Partitioning of an application on page 20
- N-tier model on page 21

The OpenEdge Application Server provides features and functionality that allow you to take full advantage of both.

The remaining text in this section details each of these aspects.

Partitioning of an application

_Application partitioning_ is the process of segmenting application logic from the user interface and data, and running it across multiple computers on a local or wide area network (LAN or WAN). Depending on your network configuration and performance goals, you can partition your application logic among many available resources on your network.
The ability to partition an application allows common business functions or data to be managed using computing resources most appropriate for the task and at the same time make them available to clients with a variety of user interfaces. Depending on how you partition your application, you can significantly reduce network traffic, hence enhancing overall application performance.

For example, if you use a network largely dedicated to connecting many clients to business function computers (application servers) that handle all the data access and thereby shield clients from this task, you can spare the network from the bulk of increased traffic that otherwise results from passing high volumes of data between data sources and these many clients. Instead, most of the data traffic goes between the data sources and the few application servers running the business functions. These business functions then aggregate and filter the high-volume data, passing on only those data bits and records that client applications require for presentation to users.

**N-tier model**

*N-tier model* is a computing model for implementing application partitioning. It supports flexible network structuring so that you can distribute application logic and the processing load among many machines across your distributed network. Because the n-tier model can support an unlimited number of clients, application servers and data sources, you can break up monolithic applications and flexibly reconfigure them to run in a network environment that best supports your computing needs.

The n-tier model supports the logical separation of user interface, application logic, and data access across two or more machines. Because of this flexibility, application partitioning in a distributed environment is closely tied to two basic deployment approaches:

- The logical three-tier model
- The physical n-tier model

The *logical three-tier model* supports separating the application logic from the user interface in the application implementation, and moving it to a remote server where the application data resides. A configuration based on the logical three-tier model can remain a physical client/server application in that it is deployed across two machines. However, the user interface, application logic, and data are logically separated from one another because the application logic owns access to all the data. In contrast, the *physical n-tier model* supports both logically and physically discrete segments. Each segment-user interface, application logic, and data—can be deployed on separate machines across the enterprise even with widely-separated resources in each segment (as over a WAN).

In an OpenEdge distributed environment, you are not restricted to using just these models, and these models can co-exist in a distributed environment. This chapter presents these models as a means to understand the various capabilities available in the OpenEdge distributed environment.

**A comparison of computing models**

This section briefly identifies the processing capabilities available with the traditional two-tier client/server model in contrast to capabilities available in the n-tier models available with OpenEdge distributed application computing. This comparison helps to illustrate some of the strategic advantages that the OpenEdge distributed application computing model offers.
Traditional two-tier model

Traditional client/server development is based on a logical and physical two-tier computing model. It is deployed across two machines and connected through a network. The user interface and application logic are tightly integrated and located on a client machine, while the data resides on a separate server machine.

The following figure shows this two-tier configuration. Often, the client application runs on a limited desktop machine; the real computing power is reserved for moving data on the server.

Figure 1: Traditional two-tier model

Although applications based on this model are relatively easy to build, they are generally limited in size and computing power and tend to rely on a great deal of custom code. These limitations can:

- Restrict the possibility of increased performance because all data required for the application logic is also passed to the same machine that manages the user interface. Thus, user interface response can suffer while waiting for application logic to process the data coming over the network.
- Reduce maintainability by allowing enhancements to propagate changes throughout the application code.
- Reduce data security because the user interface is essentially one with the application logic that handles the data.
- Make integration difficult because the higher volume of custom code and often-specialized data formats prevent any convenient interface with other application domains, perhaps similarly limited.
- Constrain any growth in access requirements because the application logic is tied to a single user interface instance.
- Hinder re-use of common functions because the code must be duplicated on each machine that hosts the application.

OpenEdge distributed application computing models

In contrast, OpenEdge distributed application computing, through the introduction of the AppServer and WebSpeed, expands computing capabilities beyond the limited boundaries of the two-tier model. This section describes the n-tier models described in this chapter in comparison with this two-tier model: the logical three-tier model and the physical n-tier model. As previously mentioned in this chapter, these models help illustrate how you can distribute your application in an OpenEdge distributed environment. You are not limited to using only these models.
Logical three-tier model

The logical three-tier model, as shown in the following figure, has a user interface that is physically separate from another machine that contains the application logic and the data. Physically, all three logical tiers can also be hosted on a single machine. Therefore, physically there are only one or two machines, but technically, or logically, there are three tiers: the user interface, the application logic, and the data.

Figure 2: Logical three-tier model

![Logical three-tier model diagram]

The configuration of the logical three-tier model promotes the writing of business logic and deploying them on an application server close to the data. An OpenEdge implementation of this model has the AppServer or WebSpeed connect to the database server using shared memory; the need to access the database over the network has been eliminated, providing faster data access to the business logic that processes it.

Another important feature of this model is the fact that the AppServer or WebSpeed remotely processes the information and only returns the results to the user interface. This capability minimizes the number of network messages while it delivers the specific data you need.

Physical n-tier model

In addition to the logical three-tier model, the OpenEdge distributed environment supports physical n-tier configurations. Although the n-tier model does not capitalize on the use of shared memory, and introduces an additional network connection that you do not have in the logical three-tier model, the deployment flexibility of this model might have significant benefits from an overall enterprise perspective.

The following figure shows how application logic can be distributed to dedicated machines without any local database connections.

Figure 3: Physical n-tier model

![Physical n-tier model diagram]
For example, you might want to configure the AppServer or WebSpeed on a system that is a dedicated computation engine with no locally attached database. Then you would use the AppServer to configure a data processing engine to access and filter bulk data from a closely connected database. Thus, you might have a physical three-tier configuration consisting of one or both of the following:

- **Web browser > WebSpeed > AppServer**
- **GUI/Character > AppServer > AppServer**

With the AppServer, you can build arbitrary complex applications across any number of computing tiers, based on the business problem that you are trying to solve.

The remaining chapters in this manual help you to understand the OpenEdge Application Server and all of the components that OpenEdge supports for application services development and integration services deployment, and how you can design and implement these services as an integral part of your distributed application computing environment.

For more information on the:

- OpenEdge Application Server and the development of application services, including support for integration services, see [OpenEdge Application Server Architecture](#) on page 25
- Development of integration services in particular, see [OpenEdge Messaging and ESB](#) on page 89
OpenEdge Application Server Architecture

This chapter provides a high-level architectural overview of the OpenEdge® Application Server, its components, and the general context in which it is used. The OpenEdge Application Server provides the core components and tools for the development of OpenEdge application services and the client applications that access them. For details, see the following topics:

• OpenEdge Application Server overview
• Understanding the AppServer
• AppServer clients overview
• AppServer adapters overview
• Understanding WebSpeed
• Unified Broker framework

OpenEdge Application Server overview

The following figure shows an overview of the OpenEdge Application Server architecture.
This architectural overview shows the Application Server in context, with the shaded areas representing OpenEdge components. The solid one-way arrows show access relationships between "client" (start of arrow) and "server" (point of arrow) components. The dotted arrows show alternate access relationships. For example, OpenEdge Web services represent an alternate access route between clients and the AppServer. The wide two-way arrows show key intermediate processes through which certain clients must communicate in order to access the OpenEdge Application Server.
Understanding the AppServer

The OpenEdge Application Server itself consists essentially of the AppServer (see Understanding the AppServer on page 27), WebSpeed (see Understanding WebSpeed on page 31), and associated adapters (not shown) that enhance or make possible access between certain client types and the AppServer. For example, OpenEdge Web services depend on an adapter to deploy, manage, and provide access to them from industry Web service clients. In this case, the adapter also encapsulates the Open Client interface required to allow AppServer application services to be accessed by Web service clients. For more information on the adapters for the AppServer, see AppServer adapters overview on page 31. Another section (Understanding WebSpeed on page 31) describes the supported client types for the OpenEdge Application Server.

Note that the AppServer and WebSpeed Transaction Server are shown both as application server components and as clients of the ultimate OpenEdge Application Server component, the AppServer. Note also that OpenEdge data sources can be accessed by any ABL (Advanced Business Language) component except the WebClient™. Finally, note that OpenEdge Web services are shown both in relation to the OpenEdge Application Server and as the industry Web services that are provided by OpenEdge.

From the overall architecture, you can see that the OpenEdge Application Server is accessible in one way or another from most industry clients, and the OpenEdge Application Server itself can access other application servers in the form of industry Web services. This shows much of the open technology inherent in the OpenEdge platform.

All of the OpenEdge components in this architecture also provide options for security that depend on the component and its function, including features for implementing password-based authentication, encryption of code and data, and data privacy on connections between components using HTTPS on the Internet or an OpenEdge Secure Sockets Layer (SSL) implementation on the intranet. For more information, see OpenEdge Getting Started: Core Business Services - Security and Auditing.

Note: While OpenEdge does support SSL connections to the OpenEdge RDBMS, OpenEdge does not support SSL connections from OpenEdge applications to DataServers.

Understanding the AppServer

The AppServer is an application server component that runs ABL procedures in response to direct client requests over a network. It helps you to build complex distributed applications by allowing a client application to call ABL procedures or user-defined functions remotely as if they were running locally on the client machine. A remote procedure or remote user-defined function is thus a procedure or function that runs in an OpenEdge (AppServer) session that is separate from that of the client, and typically on a separate machine from the client. In an ABL client, you access and invoke a remote procedure or user-defined function using a variation of the RUN statement or the FUNCTION statement (and function invocation). In an Open Client or Web service client, you invoke the remote procedure as a method on an object defined by the Open Client or OpenEdge Web service interface.

OpenEdge supports Web services in two basic ways:

- Producing Web services from AppServer application services
- Consuming industry Web services from within an ABL client

For more information on OpenEdge Web service support, see Web Services—Architecture and Tools on page 67
AppServer architecture

The following figure shows the architecture of the AppServer. The dotted arrows and borders show optional interactions and components.

**Figure 5: AppServer architecture**

An AppServer application can include the following components, potentially distributed on the following machines, as shown in the above figure:

1. **AppServer client** — This can be:
   - Any OpenEdge application, including a GUI or Character client, a WebClient, a WebSpeed agent, or another AppServer agent
   - An Open Client (.NET or Java)
   - An industry Web service client
   For more information, see AppServer clients overview on page 29.

2. **AppServer adapter** — (Optional) This can be the:
   - AppServer Internet Adapter
   - Web Services Adapter or OpenEdge Adapter for Sonic ESB
   For more information, see AppServer adapters overview on page 31.

3. **NameServer** — A process that accepts a client request for connection to an AppServer and directs that request to the appropriate AppServer instance. This is optional depending on the
AppServer configuration. If your application service is supported by multiple AppServer instances, you must use the NameServer for effective connection management.

4. AppServer — A core product set that contains the following components:

- **AppServer agent** — A process that executes ABL procedures in response to remote procedure requests and returns the results as ABL output parameters, return values, and errors. An AppServer agent is essentially an OpenEdge session with its user interface replaced by a network interface.

- **AppServer broker** — The process that manages client connections and its pool of AppServer agents, or agent pool, to execute client requests.

An AppServer instance:

- Is one AppServer broker and the pool of AppServer agents that it manages
- Has a specified configuration and makes a well-defined set of ABL procedures available as an application service for execution by client applications
- Can have any number of AppServer agents to service remote procedure requests from any number of client applications, depending on licensing and available resources

An AppServer installation is the AppServer part of an OpenEdge Application Server environment installed on a particular machine. Depending on your licensing and resources, you can configure and run multiple AppServer instances for a given AppServer installation.

The AdminServer is the core of the Unified Broker framework used to manage the AppServer and most of its related components. For more information on the framework, see Unified Broker framework on page 33.

Further information on the AppServer

For more information on how client applications connect (or bind) to and interact with one or more AppServer instances, see AppServer for OpenEdge Applications on page 37. For more information on AppServer components and how to configure and manage them, see the information on AppServer administration in OpenEdge Application Server: Administration. For complete information on developing AppServer application services and the client applications that use them, see OpenEdge Application Server: Developing AppServer Applications. For information on managing the ABL procedures of an application service, see OpenEdge Deployment: Managing ABL Applications.

AppServer clients overview

As described in the previous section, the AppServer supports a variety of OpenEdge and non-OpenEdge clients, each of which has its own capabilities and requirements. In addition to the general OpenEdge client support, the OpenEdge AppServer also supports 10.2A03 (or later) and 10.2B01 (or later) AppServer clients. This section describes the AppServer clients supported in OpenEdge and where to find more information about them.

In general, depending on the client and the OpenEdge configuration, a client can access an AppServer using the following methods:

- Direct TCP/IP connection to the AppServer with or without SSL tunneling and with or without the help of a NameServer (accessed using UDP)
Indirect Internet connection using HTTP or HTTPS (see AppServer adapters overview on page 31)

Indirect Web service connection using SOAP over HTTP or HTTPS (see AppServer adapters overview on page 31)

ABL clients

The ABL clients include any OpenEdge application, with or without its own user interface:

- GUI or Character clients, including the WebClient
- Batch clients
- WebSpeed agents as clients
- Other AppServer agents as clients

All ABL clients, except the WebClient, are supported on the appropriate platforms (Windows for GUI clients) supported by OpenEdge. The WebClient is supported on the following Windows platforms: NT, ME, 2000, and XP.

For basic information on developing OpenEdge applications, see the ABL programming documentation, starting with OpenEdge Getting Started: ABL Essentials. For information on developing direct ABL clients of the AppServer, see AppServer for OpenEdge Applications on page 37 and OpenEdge Application Server: Developing AppServer Applications. For information on managing the procedures of an ABL client application, see OpenEdge Deployment: Managing ABL Applications. For information on deploying and managing a WebClient application, see OpenEdge Deployment: WebClient Applications.

OpenEdge Open Clients

OpenEdge Open Clients include .NET and Java applications developed to access the AppServer using the Open Client Toolkit. (This also includes Web service clients, described in Web service clients on page 30.) Using the Open Client Toolkit, you can generate a client interface based on an existing AppServer application service in a form native to the client language platform (.NET or Java). You then write the client application using this client interface to access the AppServer. For more information on Open Clients, see OpenEdge Open Clients on page 55 and OpenEdge Development: Open Client Introduction and Programming.

Web service clients

Web service clients include any industry Web service client, including an OpenEdge application accessing the AppServer as a Web service. An ABL client can access the AppServer as a Web service in the same way it can consume any industry Web service. However, Progress Software Corporation recommends, where possible, that you access the AppServer from an ABL client using a direct AppServer connection to achieve optimal performance. For more information, see ABL clients on page 30. For more information on how an ABL client can consume a Web service as a Web service client, see Web Services—Architecture and Tools on page 67.

Any Web service client access to an AppServer depends on the OpenEdge Web Services Adapter, which allows any Web service client to consume an AppServer application service as an industry Web service. For more information, see AppServer adapters overview on page 31.
AppServer adapters overview

OpenEdge supports the following adapters that allow clients to access an AppServer application service using industry standard protocols:

- **AppServer Internet Adapter (AIA)** — Allows a client to access the AppServer over the Internet, using HTTP or HTTPS. This allows clients to make the same calls to the AppServer application service through an Internet firewall that they can make using a direct TCP/IP connection. For more information, see [AppServer Internet Adapter](#) on page 59.

  **Note:** The AIA also provides Internet access to the integration services provided by the OpenEdge Adapter for SonicMQ BrokerConnect for OpenEdge applications. For more information on the OpenEdge Adapter for SonicMQ, see [OpenEdge Messaging and ESB](#) on page 89.

- **Web Services Adapter (WSA)** — Allows a Web service client to access an AppServer application service (using SOAP) as an industry standard Web service hosted on a Java servlet engine (JSE) or JSE-enabled Web server. To accomplish this, you must also define a Web service interface for the application service using the Open Client Toolkit. For more information, see [Web Services—Architecture and Tools](#) on page 67.

- **OpenEdge Adapter for Sonic ESB** — Allows an OpenEdge service hosted on the Sonic ESB to be accessed as part of workflow processes managed by the Sonic ESB. The AppServer application service is accessed either by an OpenAPI when the Native Invocation methodology is employed or by using SOAP messaging when the Web Service Invocation methodology is employed. As an OpenEdge service hosted on the Sonic ESB, the AppServer application service is available to applications using a wide variety of standard protocols. For more information, see [OpenEdge Messaging and ESB](#) on page 89.

While these adapters provide open access to an AppServer from various AppServer clients, each adapter, itself, accesses the AppServer directly using a TCP/IP connection with or without SSL tunneling, and with or without the help of a NameServer (accessed using UDP). For more information on OpenEdge support for SSL, see [OpenEdge Getting Started: Core Business Services - Security and Auditing](#).

Understanding WebSpeed

*WebSpeed* is an ABL development and deployment environment that enables you to build robust, data-driven Internet transaction processing applications for the World Wide Web (WWW). WebSpeed provides application services that add database connectivity and transaction management to browser-based applications. Essentially, WebSpeed provides client access to the functionality of an OpenEdge application server where a Web browser is the client platform. In addition to building Web-based applications by design, you can use WebSpeed to Web-enable existing OpenEdge applications that previously ran stand-alone or as an AppServer-based application.

WebSpeed architecture

The following figure shows the WebSpeed architecture. The dotted arrows and borders show optional interactions and components.
A WebSpeed application can include the following components, potentially distributed on the following machines shown in the above figure:

1. **HTML client** — This is typically a Web browser that provides the user interface and run-time engine for the HTML application.

2. **WebSpeed Messenger** — A process that runs on a Web server and handles the transfer of data between the Web server and the WebSpeed agent during a single Web transaction.

3. **NameServer** — A process that accepts a client request for connection to a WebSpeed Transaction Server and directs that request to the appropriate Transaction Server instance. This is optional depending on the Transaction Server configuration. If your application service is supported by multiple WebSpeed Transaction Server instances, you must use the NameServer for effective connection management.

4. **WebSpeed Transaction Server** — A core product set that contains the following components:
   - **WebSpeed agent** — A process that executes ABL procedures in response to Web requests from a WebSpeed Messenger and returns results as HTML. Thus, a WebSpeed agent is essentially an OpenEdge session with its user interface replaced by HTML that can present the user interface in a Web browser.
   - **WebSpeed broker** — The process that manages connection requests from a WebSpeed Messenger and its pool of WebSpeed agents (*agent pool*).

A **WebSpeed Transaction Server instance**:
   - Is one WebSpeed broker and the pool of WebSpeed agents that it manages.
   - Has a specified configuration and makes a well-defined set of ABL procedures (*SpeedScript Web objects*) available as an application service for execution by client applications.
   - Can have any number of WebSpeed agents to service remote procedure requests from any number of Web browser applications, depending on licensing and available resources.
The WebSpeed Messenger is the point of contact for a given WebSpeed application service, which can be supported by one or more WebSpeed Transaction Server instances. Each WebSpeed application service must have at least one WebSpeed Messenger instance configured for it on a Web server.

A WebSpeed Transaction Server installation is the WebSpeed application service component of an OpenEdge Application Server environment that is installed on a particular machine. Depending on your licensing and resources, you can configure and run multiple WebSpeed Transaction Server instances for a given WebSpeed installation.

A WebSpeed Messenger installation is the Web server component of an OpenEdge Application Server environment that is installed on a particular machine. You can configure and run multiple Messenger instances for a given WebSpeed Messenger installation.

The AdminServer is the core of the Unified Broker framework used to manage the WebSpeed Transaction Server and most of its related components. For more information on the framework, see Unified Broker framework on page 33.

Further information on WebSpeed

For more information on how browser applications interact with one or more WebSpeed Transaction Server instances and the tools for developing WebSpeed applications, see WebSpeed on page 101. For more information on WebSpeed components and how to configure and manage them, see the information on WebSpeed administration in OpenEdge Application Server: Administration.

For complete information on developing WebSpeed application services and the HTML applications that use them, see OpenEdge Application Server: Developing WebSpeed Applications. For a look at a few sample WebSpeed applications and instructions for how to run them, see OpenEdge Getting Started: WebSpeed Essentials.

Unified Broker framework

OpenEdge server products use a common administration framework. This framework consists of the AdminServer, management utilities, and OpenEdge Management or OpenEdge Explorer for performing most tasks. This section provides a general overview. For complete information on the framework, see OpenEdge Getting Started: Installation and Configuration.

AdminServer

An AdminServer is installed on every system where you install an OpenEdge server product that includes the OpenEdge RDBMS, OpenEdge DataServer, OpenEdge Adapter for SonicMQ BrokerConnect, AppServer, AppServer Internet Adapter, Web Services Adapter, NameServer, or WebSpeed Transaction Server. The AdminServer must be running in order to use OpenEdge Management or OpenEdge Explorer, or the command-line configuration utilities. These tools contact the AdminServer to perform the actual configuration of your OpenEdge server products.

In Windows systems, the AdminServer starts automatically and runs as a Windows service (AdminService for OpenEdge). For UNIX-based systems, a command-line utility (PROADSV) is used to start and stop the AdminServer.
Note: For the complete syntax of the PROADSV utility, see the chapter on setting up your environment in *OpenEdge Getting Started: Installation and Configuration*.

### OpenEdge Management or OpenEdge Explorer

OpenEdge Management/OpenEdge Explorer is a browser-based tool. You can start OpenEdge Management or OpenEdge Explorer from the Start menu on the Windows taskbar. With this tool, you can do the following:

- Create, start, stop, and query the status of OpenEdge server products
- Configure OpenEdge server products
- Start additional or trim back running server or agent components of an OpenEdge server product

You can use OpenEdge Management or OpenEdge Explorer from any machine to manage remote WebSpeed installations on either Windows or UNIX machines. For more information about using OpenEdge Management or OpenEdge Explorer to create or configure OpenEdge Application Server products and components, see *OpenEdge Application Server: Administration, OpenEdge Management and OpenEdge Explorer: Configuration*, and the OpenEdge Management or OpenEdge Explorer online help.

### Command-line utilities

The command-line utilities allow you to manage existing OpenEdge server component configurations, as described in the following table.

**Table 1: OpenEdge server product management command-line utilities**

<table>
<thead>
<tr>
<th>Utility name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ServerMAN</strong>¹</td>
<td>Where <em>Server</em> is a unique abbreviation for the server product. This utility starts an OpenEdge server product instance, queries its status, starts and stops additional server or agent components, trims by a certain number of servers or agents (depending on the product), and stops the OpenEdge server product instance.</td>
</tr>
<tr>
<td><strong>ServerCONFIG</strong>¹</td>
<td>Where <em>Server</em> is a unique abbreviation for the server product. Validates an OpenEdge server product instance configuration in properties files (<em>ubroker.properties</em>).</td>
</tr>
</tbody>
</table>

### The ubroker.properties file

All of the administrative functions that you can perform by using OpenEdge Management/OpenEdge Explorer or the command-line utilities rely on the AdminServer to interact with other OpenEdge server product components. This interaction is based on configuration information maintained in the properties file (*ubroker.properties*) that is installed with the server product.

¹ In Windows, the utility name is the command for running it. On UNIX, the command for the utility is the utility name in all lowercase.
The `ubroker.properties` file resides in the `OpenEdge-install-path\properties` directory. A readme version of the file (`ubroker.properties.README`) includes comments relevant to setting properties for your server configuration.

For information about changing the server configuration by editing the `ubroker.properties` file, see the comments included in the property readme file. For an overview of this file and for information on modifying the `ubroker.properties` file, see *OpenEdge Getting Started: Installation and Configuration*. For information on modifying this file for the OpenEdge Application Server and related server products, see *OpenEdge Application Server: Administration*. 
The AppServer is the core of OpenEdge® support for application services in an n-tier environment, and provides an overview of its central concepts. It is the mechanism that allows you to distribute an OpenEdge application across multiple machines in a network and to provide OpenEdge application services to both ABL (Advanced Business Language) and non-ABL clients (Open Clients and Web service clients). It does this by supporting separate sessions that communicate by having one session (the client session) establish a connection (or binding) with an AppServer. The connected AppServer then provides a separate session to execute ABL procedures and to return any results to the client. The AppServer supports connections for several different operating modes, each of which provides a different mix of resource consumption and performance characteristics.

This chapter describes the basic concepts and architecture required to begin developing AppServer application services and their clients, and where to go for more information. For details, see the following topics:

- Application partitioning
- Reasons to partition
- How does a remote procedure execute?
- Connection-based model
- Configurations
- Connection process
Application partitioning

To take advantage of the AppServer you must split your application into application partitions. The AppServer provides the mechanism for these application partitions to communicate using a remote procedure calling mechanism.

Splitting your application into application partitions promotes a flexible and economical use of computing resources. For example, you can partition your application such that most compute intensive portions are co-located with the OpenEdge RDBMS or an OpenEdge DataServer. You set up the partition by installing the AppServer at this site and access it from your client machine using the appropriate remote procedure call mechanism for your client type.

Using the same machine for the application partition that accesses the database allows you to take advantage of shared memory access, eliminating the need to access the database over the network. The AppServer allows you to configure these application partitions on your network to meet your organization’s performance, deployment, and security objectives. For example, configuring the AppServer with sole access to the database provides a layer of security for the database. Client applications can only query and update tables indirectly through the AppServer. The ABL code that actually performs the updates and queries is not located on the client machine. Rather, it is located on the database machine and is executed by the AppServer.

Reasons to partition

Reasons to partition your application using the AppServer include:

- **Performance** — Using the AppServer together with the NameServer, which offers connection management and load balancing across multiple AppServer instances, you can significantly improve the performance of your distributed application because the AppServer allows you to distribute the processing load across the machines in your network. Network traffic is minimized, which can result in a dramatic improvement in client application throughput and/or response time depending on how you program your application and configure it with the AppServer.

  For Web service and ABL clients, you can maximize AppServer process utilization and availability by configuring the AppServer to use a state-free (stateless and connectionless) operating mode.

  In an ABL client, you can further improve the performance of your distributed application, especially one that has an event-driven ABL client, by initiating procedure requests asynchronously. By default, a procedure executes on the AppServer as a **synchronous request**, requiring the client to wait for the result before continuing execution. If you initiate an **asynchronous request**, the ABL client continues execution without waiting for the result. Thus, the user can continue to execute client functions after submitting one or more asynchronous requests for execution. The client then processes the results of each asynchronous request as it becomes available.

- **Security** — You can restrict access to data, as required, but at the same time take advantage of the ability to distribute and share data throughout your network. In addition to standard OpenEdge RDBMS security features and any other security features that your enterprise employs, the AppServer allows you to protect your data at the **business logic level** (where business rules are applied in your application, also known as the **application service level**). The AppServer accesses your database with user authorization that is independent of the client application. Using the AppServer allows you to restrict database access to the AppServer user, thus forcing all client applications to go through the AppServer to gain access to your corporate...
data. It is the business logic that you write using ABL and deploy with the AppServer that provides this security at the application service level.

- **Maintenance** — Because your core business logic is deployed at your AppServer site rather than at the client application sites across the network, maintenance of your distributed application is simplified significantly. A modification to the business logic only needs to be made at the single AppServer site rather than at each of the client sites across the network.

- **Open Client Access** — Using the Open Client Toolkit, you can build a Java or .NET client application that executes your business logic on an AppServer. Using additional OpenEdge Web services tools, you can also allow most any Web service client, including ABL clients, to access your application service as a Web service. Java, .NET, and Web service clients all view the application service provided by an AppServer as one or more well-defined objects (Open Client objects), depending on how you organize the application service procedures on the AppServer. A single AppServer instance can simultaneously support ABL, Java, .NET, and general Web service client applications, depending on the operating mode that you choose to configure.

- **Intelligent Data Interface** — As the interface to the OpenEdge RDBMS, the AppServer provides functionality similar to database triggers and stored procedures. In addition to native OpenEdge RDBMS triggers, you can use procedures running on the AppServer to filter or otherwise process database queries and updates.

### How does a remote procedure execute?

Once a client has established a physical or logical connection to an AppServer (see **Connection-based model** on page 41), the client application can initiate requests to run remote procedures (or user-defined functions). A **remote procedure** is a procedure executed by an AppServer agent as a result of being directly called by a client application. An ABL client can execute a remote procedure by invoking a **RUN** statement on an external procedure with the **ON SERVER** option or on an internal procedure or user-defined function within a previously executed remote persistent procedure. A Java, .NET, or Web service client can execute a remote procedure by invoking a method (on an Open Client object) that maps to the remote procedure or user-defined function on the AppServer.

A client application can also execute a remote procedure persistently (**remote persistent procedure**). In this case, the procedure leaves its context available to the AppServer session after it returns (exactly like any persistent procedure in an ABL client session). The persistent procedure context is not deleted until the client deletes it or until the client disconnects from the AppServer instance that maintains it.

### The concept of separate sessions

An AppServer and its client applications run in separate sessions. Essentially, this means that they have no context in common. For Java, .NET, or Web service client applications that run in separately built and executed processes, this is clear. For ABL client applications, it is not so obvious because, although both the ABL client and the AppServer run in separate processes, they both run OpenEdge sessions in which some context is exchanged using a common ABL mechanism.
To reinforce the concept of separate sessions, it is helpful to imagine the existence of a processing boundary between the two sessions. Only remote procedure or user-defined function requests can break through this processing boundary. An ABL client invokes these requests using the `RUN` statement, function invocation, and the `CONNECT()` and `DISCONNECT()` methods on a server object handle. For more information, see the information on programming ABL clients in *OpenEdge Application Server: Developing AppServer Applications*.

The following figure shows where an ABL client application and its corresponding AppServer session maintain processing capabilities that are separate and distinct from each other.

**Figure 7: Separate and distinct ABL sessions**

The term *client session* refers to all processing activities that occur within the context of the client application. Similarly, the term *AppServer session* refers to all processing activities that take place exclusively in the context of AppServer agents running on the AppServer.

**Note:** A given AppServer session exists in exactly one AppServer agent at any given moment. However, depending on the operating mode configured for an AppServer, the requests that a client application sends over a given connection can be processed in one or more AppServer agent sessions. For more information, see the information on understanding AppServer operating modes in *OpenEdge Application Server: Developing AppServer Applications*.

These two sessions maintain a clear separation of context: persistent procedures, transactions, database connections, AppServer connections, and database trigger events are separately scoped and operate only in the session in which they are invoked.

For example, even if a database is connected within an AppServer session, the client application must also connect to the database using a separate database connection if it wants direct access. For more information on managing database access in a distributed application environment, see *OpenEdge Application Server: Developing AppServer Applications*. However, for a WebClient™, the only way to access a database through ABL is through an AppServer session (see *OpenEdge Deployment: WebClient Applications*).

For Java, .NET, and Web service clients, specially prepared methods perform the same function, as shown in the following figure.
Connection-based model

Communications between a client application and an AppServer that provides an application service is connection-based. That is, before a client can send a remote procedure request, it must establish a connection that provides an AppServer to execute the request. The client can then send the remote procedure request in the context of that connection and can continue to send remote procedure requests using the same connection until the client terminates the connection. This connection can either be a physical connection or a logical connection, depending on the AppServer operating mode.

Physical connections

In a physical connection, both a specific client session and a specific AppServer that provides the application service are connected and functionally aware of each other. The client session invokes a remote procedure or user-defined function that is supported by the application service for execution on the AppServer to which it is connected. This allows the AppServer to maintain context for the client session between requests for as long as the two remain connected. A physical connection has varying degrees of mutual session awareness and context management facility, depending on the chosen AppServer operating mode.

The following operating modes support physical connections:

- State-aware
- State-reset
- Stateless

All these operating modes and their physical connections support an application service session model known as session-managed.

For more information on the:
State-aware, state-reset, and stateless operating modes, see the information on understanding AppServer operating modes in *OpenEdge Application Server: Developing AppServer Applications*

Session-managed application model, see the information on session models and application services in *OpenEdge Application Server: Developing AppServer Applications*

OpenEdge supports physical connections for all client types, but in a special manner for Web service clients, because client connections to all Web services are inherently logical (see *Logical connection components* on page 44).

**Physical connection components**

A client application physically connects to an AppServer with or without the help of a NameServer, as shown in the following figure.

**Figure 9: Physical connection model**

The dotted arrows indicate optional communications required to establish a connection between a client application and AppServer instances using a NameServer. The NameServer is an optional component in the AppServer architecture. However, the NameServer is almost always used to support access to multiple AppServer instances that support a single application service. Otherwise, the client has to manage connections to each AppServer on its own, greatly increasing code complexity and connection management.

The NameServer maintains a list of the available AppServers in the network that provide a given application service. When the client application wants to connect to an AppServer, it requests an available AppServer from the NameServer and establishes the connection with the AppServer supplied. For more information on NameServers and how they coordinate AppServer connections for client applications, see the information on AppServer run-time components and operation in *OpenEdge Application Server: Administration*. For more information on how AppServers maintain connections for client applications, see *Connection process* on page 49.
Physical connection IDs

When an AppServer accepts a connection request from a client, the AppServer broker assigns a connection identifier (ID) that uniquely identifies the connection between a client and an AppServer. The AppServer broker and all AppServer agents use the connection ID as an identifier when they log any information associated with the connection. OpenEdge also makes this connection ID available to both the client application and the AppServer application, which they can use to maintain any connection-related information, such as an application log.

This connection ID has the following syntax:

```
appserver-host::appserver-name::global-unique-id
```

**appserver-host**

The host name of the machine where the connected AppServer is running.

**appserver-name**

The name of the connected AppServer configuration. For more information on how you can specify this name, see the chapter on AppServer administration in *OpenEdge Application Server: Administration*.

**global-unique-id**

A value guaranteed to be globally unique for all time. This value can never be generated by some other connection at the same or a different AppServer within a single computer network.

In general, you can compare connection IDs strictly for equality, but other types of comparisons are meaningless.

Logical connections (bindings)

In a *logical connection* (also known as a *binding* from the viewpoint of the client), the client is aware of how to reach an AppServer that provides a specific application service, but it has no awareness of and maintains no persistent connection to any specific AppServer instance. Instead, using its knowledge of how to reach potential AppServers that provide the specified application service, the client invokes remote procedures and user-defined functions for execution on any appropriate AppServer that is available.

A logical connection really represents a binding between the client session and the application service itself, and OpenEdge determines the specific AppServer resource to execute each remote request. Thus, depending on the number of AppServers and agents available to support logical connections to a given application service, many client requests can be executed in parallel.

Clearly, there is no way for an AppServer involved in a logical connection to maintain context between requests for a specific client session because no AppServer is ever aware of the client whose request it services until the moment it receives the request. However, logical connections provide maximum availability of AppServer resources for multiple client sessions and they therefore more readily scale for larger numbers of client requests.

Only one operating mode supports logical connections: State-free. The state-free operating mode and its logical connections support an application service session model known as session-free.
For more information on the:

- State-free operating mode, see the information on understanding AppServer operating modes in *OpenEdge Application Server: developing AppServer Applications*
- Session-free application model, see the information on session models and application services in *OpenEdge Application Server: developing AppServer Applications*

**Logical connection components**

A client application logically connects to an application service supported by one or more AppServer instances, as shown in the following figure. The dotted arrows indicate paths of communications required for multiple AppServer instances to service multiple client requests for the same application service. The NameServer is also an optional component for managing session-free connections. As long as there are enough AppServer agents and computing power to run them, a single AppServer can still scale up for higher client loads. However, if you have multiple AppServer instances supporting an application service, you lose the benefits afforded by load balancing, which is available using the NameServer.

**Figure 10: Logical connection model**

OpenEdge supports logical connections for ABL and Web service clients only. As shown in the above figure, note that for an ABL client, OpenEdge manages all access to AppServer resources from within the client session.
Web service clients and logical connections

For Web service clients, OpenEdge hosts all application service management, including access to AppServer resources, in a remote Web service session. This Web service session is managed by the OpenEdge Web Services Adapter (WSA) and it is this session to which the Web service client is logically bound. However, in a session-managed model, the Web service session is physically connected to a single AppServer. In this case, the AppServer and the WSA can simulate an active physical connection to the client using context information that it passes in Web service messages between the AppServer and the client. For more information on how Web service clients work with both session-free and session-managed Web services, see OpenEdge Development: Web Services.

Using a single AppServer instance in the connection

If only a single AppServer supports the application service (with or without a NameServer), a logical connection between a client session and that application service funnels all application service requests from this client to that one AppServer. In this case, the AppServer remains available to receive requests from all clients bound to this application service and manages its own pool of agents to service all such client requests. Although only one AppServer receives these requests, client requests are effectively handled in parallel based on the number of agents that it maintains, thereby achieving some efficiencies in the use of available computational resources.

Using multiple AppServer instances in the connection

If you provide multiple AppServer instances to service client requests for an application service, the NameServer required to manage these requests maintains a list of the available AppServer instances on the network that support the given application service. When a client session binds to an application service, OpenEdge uses the list of AppServer instances provided by the NameServer to establish a connection pool on the client’s behalf.

A connection pool is created in a client session and includes a list of physical connections to AppServer instances that support the same application service. So, when a client binds to the application service, OpenEdge requests connections to AppServer instances that support that application service and uses these connections to service the client requests. The more AppServer instances and agents that are available to support the connection, the more the application service can service client requests in a truly parallel fashion. OpenEdge maintains this connection pool throughout the life of the logical connection, adding and trimming connections as required to satisfy specified client requirements given the AppServer resources that are actually available. With the load balancing option provided by the NameServer, you can also have more client traffic directed to AppServer instances running with more agents and more powerful computing resources.

For more information on NameServers and how they support logical connections to application services, see the information on AppServer run-time components and operation in OpenEdge Application Server: Administration and the information on session models and application services in OpenEdge Application Server: Developing AppServer Applications. For more information on how OpenEdge maintains logical connections for client applications, see Connection process on page 49.

Connecting an AppServer or application service

ABL clients use the connect( ) method on a server object handle to establish an AppServer connection or application service binding. Java, .NET, and Web service clients use similar methods to connect to an AppServer or bind to the Web service (application service).
For more information on connecting or binding from a:

- **ABL client** — See the information on programming ABL clients in *OpenEdge Application Server: Developing AppServer Applications*

- **Java or .NET Open Client** — See *OpenEdge Development: Open Client Introduction and Programming, OpenEdge Development: Java Open Clients and OpenEdge Development: .NET Open Clients*

- **Web service client** — See the information on client access to OpenEdge Web services in *OpenEdge Development: Web Services*

**Disconnecting an AppServer or application service**

A client application can disconnect an AppServer or application service at any time. An ABL client disconnects by using the `DISCONNECT()` method on the server object handle to the AppServer or application service. A Java, .NET, or Web service client uses a similar method to disconnect the AppServer or Web service.

**Configurations**

The AppServer affords great flexibility in how you distribute an application. In addition to the most basic configuration, one client application connected to one AppServer, you can distribute your applications using any combination of the following basic configuration scenarios:

- **Multiple client applications connected to one AppServer** on page 46

- **Client applications connected to multiple AppServers** on page 47

- **AppServers connected to one another** on page 48

Each of these configuration scenarios is described in the following sections.

**Note:** Each AppServer agent in any AppServer configuration scenario has the same database access capabilities as a full ABL client, including access to OpenEdge DataServers and remote database servers. Each agent has the responsibility to manage its own database connections. This is especially important in stateless or state-free AppServers, where the same agent might handle requests from multiple clients with no relation to one another, potentially leaving database connections open from one client to the next.

**Multiple client applications connected to one AppServer**

The following figure shows a connection to an AppServer from multiple client applications, with the AppServer configured with a local database. In this scenario, all running AppServer agents share self-service access to the same database. Client applications, however, have access only to data that the AppServer provides and never access the database directly. To regulate this data access, you can implement many different security arrangements on the AppServer. Each one typically restricts remote procedure access and execution based on any combination of user-based (authorization) and application-based (functional) security.
Also, each client application can be quite different and unrelated to the others, except that they all require data that the same AppServer provides. In the course of interacting with the AppServer, a client application might call a remote procedure that connects and accesses an additional database connected only to the AppServer agent handling that particular request.

**Client applications connected to multiple AppServers**

A client application can connect to multiple AppServers on the same, or on different, machines. The following figure shows how this configuration might look with a client application connected to three different AppServers running (respectively) on one UNIX and two Windows machines.
In this scenario, the UNIX AppServer accesses a local database. Each Windows AppServer accesses the same remote database server (or DataServer) running on UNIX, which thus implements a three-tier distribution of application resources.

AppServers connected to one another

AppServer agents can connect as ABL clients to other AppServers. The following figure shows three AppServer machines (one in Windows and two UNIX) and one client application connected in an n-tier fashion.
Figure 13: AppServer-to-AppServer connections


Note that there is a separate AppServer session within each AppServer agent and the connection that an AppServer agent establishes with another AppServer is part of its AppServer session. That is, a connection that an AppServer agent establishes with another AppServer is part of its session context and is not shared with any other AppServer agent. For example, each AppServer agent running on AppServer-2 establishes and manages, as client, its own unique connection to AppServer-3.

Connection process

The connection processes for establishing physical and logical connections is discussed in the following section.

To establish a physical connection:
• **Directly to an AppServer without a NameServer** — A non-Web service client must know the network location (host and port number) of an available AppServer instance to which it can connect. A Web service client invokes a special method that binds the client session to the appropriate Web service deployed to the WSA. This deployed Web service is configured to directly access a specific AppServer that provides the required application service. A non-Web service client or a Web service deployed to the WSA can also make this connection using Secure Sockets Layer (SSL) tunneling in order to secure the connection over an intranet.

• **Directly to an AppServer using the NameServer** — A non-Web service client must know the application service that it needs to access and the network location (host and port number) of the controlling NameServer of an available AppServer instance to which it can connect that has registered with the NameServer to support the specified application service. A Web service client invokes a special method that binds the client session to the appropriate Web service deployed to the WSA. This deployed Web service is configured to access the controlling NameServer that provides the physical AppServer connection required to access the application service. A non-Web service client or a Web service deployed to the WSA can also make this connection using SSL tunneling in order to secure the connection over an intranet.

• **Over the Internet using the AppServer Internet Adapter (AIA)** — A non-Web service client must know the HTTP/S URL to the AIA to access an AppServer directly, or it must know this same URL and the required application service to access an AppServer using a controlling NameServer. In any case, the AIA makes the AppServer connection according to how it is configured in OpenEdge.

To establish a logical connection:

• **Directly to an available AppServer in the session-free connection pool** — A non-Web service client must know the application service to which it needs to connect and the network location (host and port number) of the controlling NameServer for all AppServer instances that have registered with the NameServer to support the specified application service. A Web service client invokes a special method that binds the client session to the appropriate Web service deployed to the WSA. This deployed Web service is configured to access the appropriate controlling NameServer for all AppServer instances that support the application service. A non-Web service client or a Web service deployed to the WSA can also make the physical AppServer connection using SSL tunneling in order to secure the connection over an intranet.

• **Over the Internet using the AppServer Internet Adapter (AIA)** — A non-Web service client must know the application service to which it needs to connect and the HTTP/S URL to the AIA running on a Web server. In any case, the AIA makes each physical connection to an AppServer from the session-free connection pool according to how it is configured in OpenEdge.

The client application requires no other knowledge about the AppServer to which it connects except the published API (the entry points and prototypes) required to access the AppServer's remote procedures. You provide this API to client programmers based on your AppServer application and the type of client they are programming.

For HTTPS (over the Internet) or SSL (over the intranet) connections, the client, Web server (for Internet connections), and AppServer (for intranet connections) must have appropriate keys and digital certificates installed in order to authenticate and encrypt communications across these connections. For more information, see *OpenEdge Getting Started: Core Business Services - Security and Auditing*.
Application services

An application service has both a general and specific definition for an AppServer. In general, an application service is the entire business function provided by an AppServer. An application service also has a specific identity in the form of a name that identifies it to controlling NameServers and connecting clients. An application service name is essentially an arbitrary alias for the AppServer that provides its particular business function. An AppServer instance can register with more than one application service name to identify its function. Because the different application service names registered by an AppServer are simple aliases, a client application can call any remote procedure or user-defined function supported by the AppServer instance no matter what application service name the client uses to connect the AppServer.

Note: For Open Clients and OpenEdge Web services, the available remote procedures and user-defined functions are limited by a well-defined interface that you specify and generate appropriately for the client type. Therefore, this interface represents a proper subset of the functionality available in the application service. For more information, see OpenEdge Development: Open Client Introduction and Programming.

For example, an AppServer instance that accesses an inventory database with certain ABL procedures might support an application service named "Inventory." An AppServer instance that accesses an employees database with certain other ABL procedures might support an application service named "Employees." On the other hand, the same AppServer instance might support both the "Inventory" and "Employees" application services.

Assigning application service names

You specify the application service names that an AppServer supports when you configure an AppServer instance. Client applications use the application service names you specify to identify to a NameServer an AppServer instance with which to connect. The NameServer then chooses a particular AppServer instance to connect based on whether it supports the application service specified by the client.

For multiple AppServer instances that support the same application service name, you typically configure all such AppServer instances with functionally identical capabilities. Most important among these common capabilities include using the same ABL procedure installation and configuration, and assigning the same operating mode to each AppServer instance.

Fault-tolerant and load-balanced AppServer support

When distributing connection requests for fault-tolerant and load-balanced AppServers, the controlling NameServer assumes that all AppServer instances that support the same application service also support the same API. The NameServer does not validate this assumption. You must ensure that the same published API is available on all AppServer instances that register the same application service name with a NameServer. For more information on fault-tolerant and load-balanced AppServers, see the information on AppServer administration in OpenEdge Application Server: Administration. For more information on how OpenEdge manages connection requests for fault-tolerant and load-balanced AppServers, see the information on session models and application services in OpenEdge Application Server: Developing AppServer Applications.
Default service

A client application can connect to an AppServer without specifying an application service. In this case, the NameServer uses whatever AppServer registers itself as the default service. You can specify any AppServer instance as the default service during AppServer configuration. However, Progress Software Corporation recommends that you avoid relying on the default service and provide explicit application service names for all AppServer instances that rely on a controlling NameServer for client connections. For more information on AppServer configuration, see *OpenEdge Application Server: Administration*.

Connection operation

The way an AppServer connection works depends partly on the AppServer operating mode. This section describes the basic procedure that the OpenEdge interface follows in order to set up and manage the actual communications with an AppServer for the different operating modes. The AppServer supports four operating modes, each of which supports a different type of physical or logical connection.

In the figures that follow, the OpenEdge interface is the OpenEdge communications layer that manages physical and logical connections to an AppServer on behalf of a client. For ABL clients and Open Clients, this function resides in the client session context or in the AIA (if used). For Web service clients, this function resides in the WSA.

**Note:** The following procedure assumes that you are using a NameServer. Without a NameServer, the same basic procedure is followed, but all communications goes directly to a single AppServer instance with the AppServer broker managing all of its client connections from beginning to end.

In general, a client application initiates and manages the connection with an AppServer according to the following procedure:

1. The client application notifies the OpenEdge interface to send a connection request for a specified application service to a specified NameServer. The OpenEdge interface then sends the request to the NameServer, which the OpenEdge interface identifies using a host and port provided by the client application, as shown:
Note: If the client makes a connection request directly to an AppServer host and port, either through a Web service or the AIA, the procedure continues from here with step 3.

If you have configured fault-tolerant NameServers, the replicated or neighbor NameServers attempt to resolve the connection. If no such NameServer can resolve the connection, the client receives a connection error.

2. For a physical connection, the resolving NameServer chooses a particular AppServer instance that supports the specified application service (and that balances client load, if so configured) and provides the OpenEdge interface with the network location of the AppServer broker for that AppServer. For a logical connection, the OpenEdge interface requests connections to one or more AppServer brokers that support the specified application service with which to initialize the client connection pool, as shown:

3. For a physical connection or a logical connection directly to a single AppServer, the OpenEdge interface initially sets up a connection with the AppServer broker, as shown:
4. For a physical connection, the AppServer broker either maintains the connection directly between itself and the OpenEdge interface (state-less operating mode) or provides the OpenEdge interface with the address of a particular AppServer agent to which it can remain connected for the duration of the connection (state-reset or state-aware operating modes). For a logical connection (state-free operating mode), the OpenEdge interface maintains the client connection pool for one or more AppServer connections, consulting the NameServer for each request, as required by the logical connection parameters, as shown:

5. Once the connection is established according to the operating mode and connection parameters, all remote procedure and user-defined function requests from that one client are sent over this connection. In the case of a logical connection (state-free), multiple requests from the client can be executed in parallel, depending on the AppServer resources available.

**Note:** For multiple logical connections to a given application service, all bound clients can access identical AppServer connections, because a state-free AppServer has no physical connection to its clients and it is available to execute requests from all clients as fast as its own agent pool can handle the requests.

6. When the client no longer needs the connection, it disconnects from the single AppServer or all AppServers in its connection pool (for logical connections) by notifying the OpenEdge interface to send a disconnect request to each affected AppServer.

For more information on how to set up and work with a physical or logical connection, see the information on programming client applications in *OpenEdge Application Server: Developing AppServer Applications*. 
OpenEdge® provides tools that allow you to:

- Build a .NET or Java application that runs as an AppServer client. Such a client application accesses the AppServer application service in a manner that models the way an ABL (Advanced Business Language) client accesses an AppServer application service.

- Produce a Web service based on an AppServer application service that can be consumed from any industry Web service client. Because such a Web service is based on ABL technology, OpenEdge refers to it as an OpenEdge Web service (analogous to a Java Web service built with Java technology).

- Incorporate ABL procedures into Sonic ESB. The OpenEdge Adapter for Sonic ESB, supports two methodologies: Native Invocation and Web Service Invocation.

In OpenEdge, .NET or Java AppServer clients and OpenEdge Web services share common features that make them all Open Clients of the AppServer. The two most important features of Open Clients are that:

- They all rely on a common client interface definition to model access to ABL procedures called the Open Client object model.

- The client interface for each Open Client can be defined using the Open Client Proxy Generator (ProxyGen), the essential tool of the Open Client Toolkit.

For details, see the following topics:

- Overview of the Open Client object model
- Overview of the Open Client Toolkit
Overview of the Open Client object model

The following figure shows the relationship between Open Clients, the client interface, and the AppServer application service.

Figure 14: Open Client interface—the Open Client object model

The shaded elements represent OpenEdge components. The arrows show the run-time relationships, where the client application invokes operations through the client interface that are, in turn, executed remotely on the AppServer with all required inputs and outputs expressed as parameters on the operations.

Client interface format

The client interface itself differs in format among the types of Open Client:

- For a .NET client, it is a .NET assembly.
- For a Java client, it is a set of Java classes.
- For a Web service client, it is whatever native mechanism on the client platform best represents the client interface definition, which is made available to the client by OpenEdge in the standard Web service manner, using Web Service Definition Language (WSDL) presented in a WSDL file.

For .NET and Java Open Clients, OpenEdge can generate the client interface as well as the definition. For Web service clients, OpenEdge generates the client interface definition, but leaves it to the client platform (as is standard) to generate the client interface itself.

Definition of the client interface

The Open Client object model, which is the basis of the client interface definition, consists of three types of interface objects that model the structure of an AppServer application service. These interface objects include the:

- **AppObject** — Represents the complete set of AppServer external procedures in the application service, consisting of one or more application methods (non-persistent procedures), **SubAppObjects** (AppObject subset) and **ProcObjects** (persistent procedures) and also represents the point of physical or logical connection between the client and the AppServer.
• **SubAppObject** — Represents a subset of the AppObject, consisting of one or more application methods (non-persistent procedures) and ProcObjects (persistent procedures)

• **ProcObject** — Represents a single persistent procedure, consisting of any number of methods (internal procedures and user-defined functions)

So, depending on the type of client, these objects might be represented differently, with object methods represented as invocable operations, be they functions, procedures, subroutines, or indeed methods—whatever is most natural for the client platform (.NET, Java, or whatever is the Web service client platform). In all cases, the Open Client implementation ensures that invocation of the appropriate method representation executes the corresponding ABL code on the AppServer.

For .NET and Java Open Clients, you can use ProxyGen to generate proxy objects that encapsulate the appropriate client interface using a custom API. You can also access the client interface using a single common API (OpenAPI). The .NET or Java OpenAPI allows you to access any AppServer application service without the need to generate and program to a custom set of proxy objects.

For more information on the:

• Open Client object model, see *OpenEdge Development: Open Client Introduction and Programming*

• .NET Open Client interface, see *OpenEdge Development: .NET Open Clients*

• Java Open Client interface, see *OpenEdge Development: Java Open Clients*

• Web service Open Client interface, see the information on programming Web service clients to consume OpenEdge Web services in *OpenEdge Development: Web Services*. For an overview of OpenEdge Web services and OpenEdge support for Web services in general, see *Web Services—Architecture and Tools* on page 67

---

**Overview of the Open Client Toolkit**

The following figure shows how OpenEdge allows you to define a client interface using the Open Client Toolkit.
Using the ProxyGen tool, you specify the procedures of the AppServer application service that you want to include in the client interface and how you want to represent them in terms of the Open Client object model. From this common interface definition, you can then generate:

- **.NET Open Client** — Proxy objects in the form of a .NET assembly.
- **Java Open Client** — Proxy objects in the form of a set of Java classes.
- **Web service Open Client** — The client interface definition in a form that eventually becomes a WSDL file for use by Web service clients (or client developers) to generate the client interface itself.
- **Sonic Web Service** — The Sonic Web Service definition for the OpenEdge Adapter for Sonic ESB Web Service Invocation methodology. The Web Service Invocation methodology takes a WSM file as its primary input, and you can edit it with a custom OpenEdge resource editor accessed through the Sonic Management Console.
- **Sonic Native** — The Sonic Native Invocation file (.esboe file) for the OpenEdge Adapter for Sonic ESB Native Invocation methodology. An invocation file is dropped onto the canvas of the Sonic ESB Process Editor in Sonic Workbench to create a new step in an ESB process.

The Open Client Toolkit contains additional tools to facilitate Open Client implementation, depending on the type of Open Client. For more information on ProxyGen and the entire Open Client Toolkit, see *OpenEdge Development: Open Client Introduction and Programming*. 
The AppServer Internet Adapter (AIA) Web-enables the AppServer or OpenEdge Adapter for SonicMQ BrokerConnect (for ABL—Advanced Business Language—clients only) by supporting HTTP and HTTPS protocols for sending information between the client and AppServer (or OpenEdge Adapter for SonicMQ BrokerConnect) across the Internet. Internet access provides a way for clients to access an AppServer (or OpenEdge Adapter for SonicMQ BrokerConnect) when the clients and server are separated by firewalls that require connections using HTTP or HTTPS. Using the AIA, you can access the AppServer over the Internet with any Web-enabled ABL client, .NET Open Client, or Java Open Client.

For information on the AppServer, see AppServer for OpenEdge Applications on page 37 and for information on the OpenEdge Adapter for SonicMQ, see OpenEdge Messaging and ESB on page 89. You can also use the OpenEdge Knowledge Base on the Web (http://www.Progress.com/support) for information about Web-enabled ABL clients. For information on Open Client applications, see OpenEdge Development: Open Client Introduction and Programming.

Note: In this chapter, all occurrences of the term "AppServer" can also mean "OpenEdge Adapter for SonicMQ BrokerConnect."

For details, see the following topics:

- General architecture
- Using the AppServer Internet Adapter with HTTP
- Using the AppServer Internet Adapter with HTTPS
- How clients use HTTPS
- AppServer Internet Adapter and proxy servers
General architecture

The following figure shows the general AIA architecture.

Figure 16: AppServer Internet Adapter (AIA) architecture

Running AppServer applications over the Internet can present challenges because of the restrictive access policies enforced by many company networks. Companies often use firewalls to protect their internal systems from unauthorized access. A firewall is a computer and software that sits between a private network and the Internet. The firewall restricts the types of protocols that can travel into the internal network. The firewall often restricts the ports these protocols can use. For example, it is common to limit traffic originating outside a firewall to those applications using HTTP on port 80.

AIA is a Java servlet that is run by a Java servlet engine (JSE) and allows Java programs to run on a Web server. AIA supports deployment of distributed applications over the Web by extending the AppServer architecture to support the HTTP and HTTPS protocols. When you use AIA, communication between an AppServer client and an AppServer is encapsulated within standard HTTP requests and tunnels those requests through a Web server. AIA then converts requests that are encapsulated within HTTP or HTTPS to the standard AppServer protocol. This conversion process allows AppServer clients to communicate through firewalls without requiring a network administrator to allow additional protocols or ports to be accessed through a firewall.

HTTPS is HTTP tunneled through a Secure Socket Layer (SSL) connection rather than the standard unencrypted TCP/IP connection. HTTPS is used when privacy, integrity, and Web server authentication are needed in addition to HTTP tunneling over the Internet.
Using the AppServer Internet Adapter with HTTP

When using HTTP, all requests between the client and the AppServer go through a Web server that forwards each request to the AppServer. Depending on the client type, an AppServer client specifies HTTP tunneling for use with OpenEdge as follows:

- For an ABL client, including WebClient, the client uses the `-URL` connection parameters of the server handle by specifying a URL with an HTTP protocol.
- For an Open Client, you specify the URL in the AppObject proxy object that you can create with ProxyGen.

The URL contains the host and port of the Web server, the path to the AIA instance, as well as the requested application service. For information about how a client connects to an AppServer using HTTP:

- For ABL clients, see *OpenEdge Application Server: Developing AppServer Applications*
- For Open Clients, see *OpenEdge Development: Open Client Introduction and Programming*

Each AIA instance can either connect directly to an AppServer or use a controlling NameServer to connect. How the connection is made is specified when you configure the AIA instance. Each request sent from an AppServer client to the Web server is encapsulated within one or more standard HTTP post requests. The URL contained within the post request indicates to the Web server that it should forward the request to AIA.

When the AIA instance receives the connection request from an AppServer client, the requested application service name is included along with the `userid`, `password`, and `appserver-info` values that are normally passed in a connection request to an AppServer. AIA then passes the application service name either to the controlling NameServer and the Name Server passes the request to the AppServer, or AIA passes the request directly to the AppServer. For more information, see the information on configuring and managing the AppServer Internet Adapter in *OpenEdge Application Server: Administration*.

**Note**: Because the HTTP protocol is connectionless, you cannot use the termination of the connection with the AppServer client as an indication that the AppServer client’s connection to the AppServer should be terminated. Because of failures at the client site, as well as the network between the client site and the Web server, it is possible that an AppServer client could connect to an AppServer and never gracefully disconnect. To guard against the possibility of an AIA instance maintaining orphaned connections, you can use the `idleConnectionTimeout` property in the `ubroker.properties` file to configure an AIA instance to automatically terminate the connection if a connection is idle for more than the specified time. For more information, see *OpenEdge Application Server: Administration*. For information on modifying the `ubroker.properties` file using the `mergeprop` utility, see *OpenEdge Getting Started: Installation and Configuration*. 
Supported client platforms

HTTP is supported on the platforms indicated for the following clients:

- The WebClient is supported on the following Windows platforms: NT, ME, 2000, and XP.
- The full ABL client is supported on all platforms supported by OpenEdge.
- The Java Open Client is supported on all platforms that have Javasoft-compatible JVMs.
- The .NET Open Client is supported on all Windows platforms that have the .NET Framework installed.

Supported server platforms

On server machines, you can use any Web server that supports HTTP. For more information, see the information on configuring and managing the AIA in OpenEdge Application Server: Administration.

Using the AppServer Internet Adapter with HTTPS

HTTPS allows Web-enabled clients to connect to an AppServer using a secure connection provided by SSL tunneling over the Internet. The requirements for using HTTPS depend on the client type. For information on using HTTPS in ABL clients (including WebClient) to access a Secure AppServer, see the information on AppServer connections using a URL in OpenEdge Application Server: Developing AppServer Applications. For information on using HTTPS in Open Clients to access a Secure AppServer, see OpenEdge Development: Open Client Introduction and Programming.

HTTPS allows Web-enabled clients with OpenEdge Client-Side Security to connect to a Secure AppServer over the Internet. OpenEdge Client-Side Security access depends on the type of client you use:

- **ABL client** — An ABL client does not necessarily include OpenEdge Client-Side Security. To include OpenEdge Client-Side Security with your ABL client, you must install one of the following OpenEdge products:
  - OpenEdge Application Server Basic Edition
  - OpenEdge Application Server Enterprise
  - Client Networking
  - WebClient
  For more information on OpenEdge installation, see OpenEdge Getting Started: Installation and Configuration.

- **Open Client** — This installation comes with a number of distribution packages that provide client-side security for Open Clients to access a Secure AppServer. You must select the appropriate package and include it with your application. For more information about the distribution packages available for Open Clients to access a Secure AppServer, see OpenEdge Development: Open Client Introduction and Programming.
Once you have installed a client that includes OpenEdge Client-Side Security and have a Secure AppServer, you can use HTTPS to access the AppServer. HTTPS extends HTTP by executing the HTTP protocol across an SSL connection rather than an unencrypted TCP/IP connection. When an SSL connection is established between the AppServer client and the Web server, the HTTP protocol is executed in the context of the encrypted and authenticated channel. Using the HTTPS protocol to connect to a Secure AppServer gives the client the capability of tunneling through firewalls and sending encrypted data.

How clients use HTTPS

The following figure shows the AIA architecture using a secure Internet connection.

Figure 17: AppServer Internet Adapter architecture with HTTPS

Note: If the client uses HTTP to connect, and, on the server side, the httpsEnabled property in the ubroker.properties file is set to one (1), the AIA instance redirects the client to retry the Web server (or JSE) connection, using an HTTPS-protocol URL that the AIA returns automatically.
When using HTTPS, OpenEdge establishes an SSL connection with the Web server (or JSE). As part of establishing the connection, OpenEdge verifies server access by comparing a digital certificate that has been installed on the server with a corresponding digital certificate installed on the client. If OpenEdge determines that the server certificate is not valid, the connection to the AIA instance is denied. These digital certificates are based on public and private encryption keys issued by a trusted Certificate Authority (CA).

When checking a certificate, any OpenEdge Internet client verifies that:

1. The server certificate is signed by one of the trusted root CA certificates installed on the client machine.
2. The certificate has not expired by comparing the current time to the timestamps included in the certificate.
3. The certificate host name in the URL is the same as the common name in the certificate Subject field.

   This is an optional verification that ensures the host machine that the client connects to is the intended host machine. This verification is done by comparing the host name the user specified in the URL with the host name in the certificate returned by the Web server.

By default, OpenEdge performs host verification. However, when establishing a connection, a client application might indicate that host verification should not be performed. For WebClients and ABL clients, a client can indicate that host verification should not be performed by using the --nohostverify connection parameter. For Open clients, the client uses the RunTimeProperties.setNoHostVerify method on the runtimeProperties object to indicate that host verification should not be performed.

If any one of these steps determines that the certification is not valid, then the connection to the AIA instance fails.

**Supported client platforms**

HTTPS is supported on the platforms indicated for the following clients:

- WebClient is supported on the following Windows platforms: NT, ME, 2000, and XP.
- The full ABL client (including AppServer and WebSpeed agents) is supported on all platforms on which OpenEdge runs except AIX and all 64-bit platforms.
- Java Open Client is supported on all platforms that include Javasoft-compatible JVMs.
- The .NET Open Client is supported on all Windows platforms that have the .NET Framework installed.

At the client machine, the root CA certificate for all server certificates must be installed before HTTPS can be used. For WebClients and ABL clients, the root CA certificates must be installed in the OpenEdge-Install-Directory/certs directory, where OpenEdge-Install-Directory is the OpenEdge installation path. Root CA certificates for publicly available and well-known CAs, such as VeriSign®, are automatically installed during OpenEdge installation.

For more information on managing digital certificates:

- For ABL clients, see OpenEdge Getting Started: Core Business Services - Security and Auditing
- For Open Clients, see OpenEdge Development: Open Client Introduction and Programming
Supported server platforms

For an Internet-secure AppServer Internet Adapter, you can use any Web server or Java servlet engine (JSE) that supports HTTPS. For more information on configuring the AIA for secure Internet access, see OpenEdge Application Server: Administration.

AppServer Internet Adapter and proxy servers

An HTTP proxy server is a special type of client-side firewall that limits access to the Internet to the host machine on which the proxy server is running. Other machines running within the client network must access the Internet indirectly by going through the proxy server.

The following figure shows the AIA architecture with a proxy server.

Figure 18: AIA architecture and HTTP proxy server

There are many different types of HTTP proxy servers available. Certain types of HTTP proxy servers require that the client application know the location of the proxy server and establish a connection to the proxy server according to the specification set out in the HTTP standard. If the client network is protected by a proxy server for which an explicit connection is needed, the client application must specify the proxy host, port user, and password.
For WebClients and other ABL clients, the host, port, user, and password are specified using the Proxy Host (`-proxyhost`) and Proxy Port (`-proxypport`) startup parameters. For Open Clients, the host, port, userid, and password are specified using methods on the `runTimeProperties` object.

For more information on using a proxy server:

- With ABL clients, see *OpenEdge Deployment: Startup Command and Parameter Reference*
- With Open Clients, see *OpenEdge Development: Open Client Introduction and Programming*
Web Services—Architecture and Tools

Web services represent an industry standard for producing and deploying services over a network. In OpenEdge®, you can produce and deploy Web services based on operations written in ABL (Advanced Business Language) and that run on the AppServer. These OpenEdge Web services are consumable from most any industry Web service client using standard communications and message protocols. You can also consume industry Web services and invoke Web service operations from any OpenEdge application as an industry Web service client. This chapter describes Web services and how OpenEdge supports them.

For more information on OpenEdge support for OpenEdge Web services and as well as support for OpenEdge applications accessing Web services, see OpenEdge Development: Web Services. For more information on deploying and managing OpenEdge Web services, see OpenEdge Application Server: Administration.

For details, see the following topics:

• What Web services are and how they are used
• OpenEdge support for consuming Web services
• OpenEdge support for producing Web services
• Benefits of producing Web services using OpenEdge
• OpenEdge Web services architecture
• Tools for building and managing OpenEdge Web services
What Web services are and how they are used

A **Web service** is an application that can be consumed (accessed and used) over the Internet (or an intranet) using industry-standard protocols. To consume this application, a client (**Web service client**) invokes operations that are described using Web Services Description Language (WSDL) and sent to the application using Simple Object Access Protocol (SOAP) over HTTP (or HTTPS). The application executes the operations and returns the results to the client using SOAP over HTTP/HTTPS. A Web service **provider** (Web application server) hosts the Web service and manages all communications between it and its clients. Simply put, Web services provide an industry-standard way for all types of client applications to call functions on all types of application servers, over any network configuration that supports SOAP over HTTP, and where the application program interface (API) can be described using WSDL.

**Note:** Web services standards mention other possible communications and transport protocols besides SOAP over HTTP. However, OpenEdge supports the most common Web service standard, which is SOAP over HTTP.

The key words in defining Web services are **industry standards**. Web services represent a widely accepted set of industry standards that allow distributed applications to communicate and exchange data without concern for each other's specific platform requirements.

These standards combine and extend technologies that are universally accepted for one purpose (HTTP for the World Wide Web) or are now widely used to develop standard applications for many purposes (XML). They have arisen from a collaboration of industry representatives who have formed standards bodies to develop and support them. Two of the most important such bodies include the Web Services Interoperability Organization (WS-I), which promotes the development of standardized interactions between Web services and tools to validate them, and the World Wide Web Consortium (W3C), which coordinates and publishes the specifications for most of the standards that support Web services development, such as HTTP and the XML-based standards, SOAP and WSDL.

The following sections describe the basic components of Web services and the standard architecture that supports them.

**Web services and the telephone system**

Essentially, Web service technology is to Web service clients and Web services what the world wide telephone system is to the parties that use telephone lines. The world wide phone system provides a standard way for different types of phones and other devices to communicate and also provides a standard way for parties to locate and call each other on the phone system. As long as a caller knows a few basic facts about the party they are trying to call, such as the phone number, language, and identity (or function) of the party, the caller can establish virtually immediate contact to tell (or send) the party information and to receive replies from them, no matter where in the world the two parties are connected to the phone system. This works for all compatible parties on the phone system, whether human or machine. In fact, both the number of parties and the uses that they make of the phone system have increased steadily over time, including such innovations as e-mail and faxes, as well as the original voice communications for which it was intended. It all works because of the agreed-upon standards developed in support of the basic telephone system.
For computer applications with access to a network, the same thing essentially applies to Web services. As long as two applications agree on (or can translate) the protocols, data types, and operations used to talk with each other, one such application (acting as client) can tell the other application to perform some function on its behalf or can request information from the other application, no matter where the two applications are located on the network or what execution environment they are running in. Like the phones, faxes, modems, other devices, and their users in the telephone system, the number and types of applications for Web services is potentially limitless.

**Earlier attempts to standardize interoperability**

Earlier attempts to solve the problem of general application interoperability have failed because they have relied on technologies that are not universally accepted, not generally available, or are too difficult to use. For example, four of the more common previous alternatives include EDI, COM/DCOM, RMI, and CORBA, and these have failed as a basis or alternative to Web services for some of the following reasons:

- Electronic Data Interchange (EDI) is very expensive to implement, proprietary, and single-purposed.
- Common Object Model (COM) and Distributed Common Object Model (DCOM) are limited exclusively to Microsoft platforms.
- Remote Method Invocation (RMI) is strictly a Java technology.
- Common Object Request Broker Architecture (CORBA) is based on open standards. But CORBA standards are numerous and complex, requiring a relatively large and expensive effort for vendors to implement. Also, because there is little common agreement on standards interpretation among CORBA implementors, different implementations tend to be inconsistent with and difficult to use with each other.

Web services, in contrast, rely on relatively simple standards that already exist to support the Internet and World Wide Web (such as XML and HTTP) and are therefore already widely accepted and used in the industry.

**Standard Web services architecture**

The following figure shows the basic Web services architecture supported by the industry. This architecture is common to all implementations of Web services.
Figure 19: Web services architecture

In the above figure, the set of components required for Web services appear with gray shading. The dashed arrow shows that the WSDL reflects the application server interface, but does not come directly from the application server, itself. The following sections describe how this architecture works and provide more information on the advantages that have increasingly made Web services an industry standard.

Web services client

In general, the typical Web services architecture includes an application acting as the Web services client (as shown by the ABL client in Figure 19 on page 70). This Web services client is an application that can access one or more Web services, typically using the APIs provided by client interfaces. A Web service client interface simplifies client programming by presenting the Web service interface directly in the native programming language of the client (ABL, Java, .NET, and so on). The API to access this Web service interface is often encapsulated in the client interface as objects designed for the particular client language. Client development platforms either allow you to generate a complete client interface directly from a given WSDL file or generate complete documentation on how to access and invoke operations in the Web service, further simplifying client development.

For more information on how OpenEdge supports client access to Web services from ABL, see the OpenEdge support for consuming Web services on page 76.
Run-time process

To invoke a Web service operation, the client application makes a request, which is essentially a remote procedure call, by invoking an operation (method on a client interface object, function, or procedure, as is appropriate for the client language). This operation invocation causes the client to create an industry-standard, SOAP-formatted message, which includes any input or output parameters that the method requires. The client then sends the message, using an HTTP (or HTTPS) Post method, to a Web service provider that hosts (or provides) the Web service.

The Web service provider then normally relies on a Web server (which has an HTTP Listener) to receive the HTTP Post method and pass the client SOAP message to a SOAP processor, which also runs in conjunction with the Web server. The SOAP processor translates the message and passes it to a business interface, which in turn, translates it into a form that can be consumed by an appropriate application server.

The application server that implements the Web service receives the translated message and invokes the operation in the business logic that corresponds to the requested Web service method or procedure. The method or procedure performs some function and returns a result (if only to indicate success or failure) that includes any values returned as output parameters and possibly a return value.

This process reverses itself to return the result (output or return values) to the client. The application server passes a message that includes the output parameter values back to the Business Interface and SOAP processor, which converts it to a SOAP message. The Web server then passes the SOAP message as an HTTP Response to the Web services client where the message is read and interpreted by a client-side SOAP processor as the return or output parameter values from the original Web service method.

Design and development process

At this point, a question remains: How does the client application identify and understand how to invoke the methods and procedures of a particular Web service? This is the function of WSDL, the XML-based industry standard that describes the content and format of SOAP messages used to interact with a particular Web service.

The developer of a Web service typically has design and development tools to automatically generate WSDL files. Thus, they can design and generate a WSDL file that defines the Web service interface to the business logic they have written (or will write) for an application server. The typical developer of a client application also has code generators that can read this WSDL file and automatically generate the code required to construct and parse SOAP messages to access the Web service from within their client application.

Because the WSDL files used to define Web services conform to industry standards, the Web services development platform used to make the application server accessible as a Web service is completely independent from the Web service client development platform. Each platform can thus be provided by different vendors as long as they both generate conforming SOAP messages. This means that if the client developer knows how to read WSDL and the Web service developer knows how to write WSDL, and both have the understanding to generate and interpret compatible SOAP messages, they can each manually write all their own code to (respectively) invoke Web service operations and serve Web service clients. However, as for most programming languages and Internet standards, many available client and Web service development platforms provide tools and code libraries to make Web services programming much more productive.
Application servers

The heart of any Web service is the business logic that runs on an application server (as shown in Figure 19 on page 70). Web service support exists for many existing application server platforms, and new application server platforms dedicated to supporting Web services are also available. Existing server applications can often be packaged as Web services with little or no modification to application code. For example, existing server applications written in Microsoft C# can be supported as C#.NET Web services using ASP.NET. Similarly, Java server applications based on Enterprise Java Beans (EJBs) are often supported as Web services by vendors of Java Servlet Engines (JSEs) using Java servlets to implement both the SOAP processing and interface to the EJB application. Thus, the EJB application might run also as a Java servlet on the Web server, in a separate JVM local to the Web server, or on a separate machine that hosts the application on a network in n-tier fashion.

In all cases, Web services provide some gateway between SOAP and the application server interface (the SOAP processor and business interface shown in Figure 19 on page 70). This gateway translates Web service messages between SOAP and the interface for the business function supported by the server application. Therefore, the OpenEdge Web services tools provide equivalent support to create Web services from both new and existing AppServer applications.

HTTP/S

HyperText Transfer Protocol (HTTP), along with its secure cousin, HTTPS, is the most widely used application-level protocol for transporting remote invocation messages between a Web services client and the Web service provider, typically a Web server or JSE. As such, HTTP/S is the modern telephone line for the communications portion of Web services. It is ubiquitous and meets the basic requirements fulfilled by the telephone line, handling secure communications between Web service clients and providers located anywhere in the world.

Thus, most Web service clients can exchange messages with Web services using a standard HTTP/S Post method and Response. Again, because it is already widely used for secure and nonsecure Internet data communications in a connectionless mode, it is already the prevailing Web services standard, avoiding the need to create a new Web services communications protocol or the client/server technology to support it.

SOAP

Simple Object Access Protocol (SOAP) is an XML-based standard that defines the protocol to format messages exchanged between a Web services client and a Web service. Thus, a SOAP message generally contains one of the following:

- The identity and parameter values for a single method or procedure invocation on a single Web service
- The parameter values for the response from a Web service invocation back to the calling client

SOAP is an open messaging protocol that supports the exchange of data using multiple SOAP message formats (combinations of SOAP communications style and SOAP representation, or use). Depending on the format, SOAP messages pass data using XML schema or a combination of XML schema and a custom encoding. SOAP also supports multiple bindings that use different communications protocols to transport these messages. However, because of the ubiquity of HTTP/S, SOAP over HTTP is the prevailing standard protocol binding for Web services.
Again, SOAP, as a widely available XML standard, allows virtually any existing Web service client technology that can code and decode SOAP to use Web services with no additional support. However, the complexity of human programming is such that most Web service client vendors provide run-time libraries and code generation tools (see WSDL on page 73) to enhance their existing client platforms, making them more Web-service-ready.

Web services client platforms (OpenEdge, Microsoft .NET, Apache AXIS, and so on) and Web Service Providers typically support only a few of the possible SOAP message formats. However, in order for a given client to access a Web service supported by a given Web service provider, the Web service must use a SOAP format that the client supports. Thus, flexible Web service providers support Web services using the SOAP formats that are supported by most Web service clients. For more information on SOAP and the SOAP formats that OpenEdge supports for Web services built in ABL as well as Web services accessed by ABL client applications, see OpenEdge Development: Web Services.

**WSDL**

WSDL is another XML-based standard that supports typical Web service implementations, and is used to describe Web services to potential Web service clients. A single WSDL file typically defines one Web service and its operations sufficiently to program a client to the specified Web service interface. However, to use this Web service interface in a client application effectively (as with any interface) usually requires documentation in addition to the WSDL definition itself, especially to describe the requirements for data that is input to the interface.

The WSDL standards provide several basic XML elements to define a Web service, including its name, the set of operations that it supports, the data types of parameters on those operations, the SOAP format of the messages used to invoke operations and return their results, and the exact protocol bindings and services that support the Web service; in short, everything that a potential Web services client needs to use the Web service, wherever it is hosted on the Internet.

Theoretically, Web service developers can write their own WSDL file to define a Web service they are building. However, most Web service development vendors provide tools to automatically generate the WSDL file directly from the application server interface and additional information, including the SOAP format and the location of the Web service, to make the Web service available to potential Web service clients. Similarly, most Web service client platform vendors also provide tools to take the WSDL file as input and automatically generate the code for the client interface that the client application calls to invoke operations on the specified Web service.

For more information on WSDL and how OpenEdge uses it to define the client interface for OpenEdge Web services as well as to define the client interface for industry Web services accessed by ABL clients, see OpenEdge Development: Web Services.

**Web services in action**

So, for example, suppose that an application, such as a global weather forecaster, needs to know the current outdoor temperature of Paris, France, as shown in the following figure.
It might invoke a `getTemp` operation to obtain this information on a standard WorldWeather Web service located on the Internet in New York.

The WorldWeather Web service can provide this information to a client application, such as the global weather forecaster, as long as it is connected to the Internet and knows how to communicate as a Web service client. If this WorldWeather Web service acts, in turn, as a Web service client, it might also access standard LocalWeather Web services running in each city of the world. By invoking a `getTemp` operation on the appropriate LocalWeather Web service, the WorldWeather Web service can also return the current temperatures of New York, Moscow, or any other city to any Web service client application that requests it.

The LocalWeather Web service running in any one city can be built with a computer language and be running on a computer or operating system that is different from all others. As long as each LocalWeather Web service communicates with the Internet as a "standard" Web service, the WorldWeather Web service can get the information it needs to satisfy all client requests for city temperatures anywhere in the world that hosts LocalWeather Web services.

This scenario assumes that the WorldWeather Web service already knows the identity and location of all available LocalWeather Web services or has a way of locating any such Web service that it requires.
Who works with Web services?

The people who work with Web services serve three fundamental technical roles:

- Web service developers
- Deployers
- Client developers

For small Web services deployments, within a single intranet for example, the same person might well assume all three basic roles. But it is the separation of function, in part, that makes the same Web services technology useful for both small and large deployments.

Web service developers

The Web service developer is the one who designs and builds a Web service application, including the code that runs on the application server and the Web service interface to that code defined in the WSDL file. This role might split between the one who writes and maintains the application server code and the one who defines and maintains the WSDL file, especially if the Web service developer is enabling an existing server application as a Web service.

Deployers

The Web service deployer is the person who hosts and makes the Web service and its definition (WSDL file) available to established or potential Web service clients. Often, the same person also maintains the Web service provider environment (see Figure 19 on page 70) that hosts the Web service on the Internet (or an intranet).

Client developers

The Web service client developer is essentially the user of the Web service, the person who develops a client application to use a Web service based on the definition provided by a WSDL file they obtain from a Web service deployer. The client developer ensures that the Web service they choose for their application uses a SOAP message format supported by their own Web service client technology. They also ensure that they have the necessary documentation (often supplied by the Web service deployer) that explains the client programming model required by the Web service they are using.

When to use a Web service

Some programmers familiar with other distributed application environments, such as traditional client/server and other n-tier architectures, might initially think Web services unnecessarily complicated for their needs. Others might find Web services simpler to work with than what they are used to, but less efficient at run time because the content communicated between Web services and their consumers is text-based (HTTP and XML). The main advantages already noted (see Earlier attempts to standardize interoperability on page 69) include the reliance on widely available (ubiquitous) standards that are already in place for other uses (HTTP/S and XML). Added to this is the relative simplicity (compared to CORBA, for example) of supplemental standards, like SOAP and WSDL, needed to complete the general toolset required for implementation.
Flexibility and integration

The ubiquity of these standards supports application distribution and integration on a wide scale, from local client/server to global n-tier, with minimal additional work on infrastructure to increase the scale. In addition, the application language and execution environment of a Web service is often completely different from the environment of the client application that is using it. Thus, while execution for some Web service applications might be less efficient in real time than an equivalent proprietary solution, the proprietary solution is likely to be far less flexible and interoperable than the Web service solution. Thus, where development time and flexibility is more critical than computational performance, Web services can offer greater efficiency and economy overall. On the other hand, a Web service might also represent the highest performance option available, especially where no proprietary solution currently exists.

Note: The least appropriate application for a Web service is one that depends on maximum data throughput over the wire in a given period of time or one that exists in a homogenous environment behind the firewall.

Web service flexibility also facilitates the automation and integration of processes between various business partners, such as manufacturers, suppliers, and banks. Once a business relationship exists among these partners, the external integration of their applications using Web services can actually increase the efficiency (that is, performance) of complex business processes, regardless of the performance of the individual partner applications.

Adaptability and longevity

Web service technologies focus on standards for how a client application talks to a server application, instead of how and with what tools the client or server application are built. That is, Web services separate the application logic and business function from the means of application communication and interoperability, much as the application user interface is now commonly separated from its business logic. Thus, Web services can provide a means to extend the life of legacy applications that have served well for one domain, making them potentially serviceable for many new and different domains. For example, a local-weather-reporting application that has served a single community might now become part of a global weather-reporting system by adapting it as a Web service. At the same time, the function and scope of the application in its original operating environment can often remain unchanged.

OpenEdge support for consuming Web services

The following figure shows how an OpenEdge application can consume a Web service as a Web service client.
The dark shaded portions are the part of the client that are dedicated for Web service access. This picture is very much the same for any Web service client platform. However, in place of Java, VB.NET, or some other language, OpenEdge supports ABL.

As with other client platforms, OpenEdge relies on the WSDL file provided by the Web service to define the Web service interface for the client. In OpenEdge, the same WSDL file must be accessible at development time in order to allow the OpenEdge WSDL Analyzer to generate client interface documentation and at run time in order to allow the client application to consume the Web service. For more information, see OpenEdge Development: Web Services.

OpenEdge support for producing Web services

In OpenEdge, you can produce and deploy Web services using the AppServer, OpenEdge Web service tools, and the OpenEdge Web Services Adapter (WSA). The Web service tools allow you to develop a Web-service-enabled AppServer application using the Open Client Toolkit, then to deploy this application as a Web service using the Unified Broker framework. You can define Web services using the Open Client Toolkit much as you define traditional Open Client proxies. However, instead of generating Open Client proxies for immediate access by Open Client applications, you generate a client interface definition, and deploy this definition as a Web service that you then manage within the context of the WSA. For more information on the Open Client Toolkit, see OpenEdge Open Clients on page 55.
The WSA is a Java servlet running on a Web server or stand-alone JSE. This Java servlet serves as the SOAP-to-business interface gateway and Web service management engine for your Web service. Thus, the WSA provides user access to the Web service WSDL file and supports Web service administration; it also manages the exchange of Web service SOAP requests and generates SOAP responses between Web service clients and AppServers at run time.

For more information on the architecture and tools for producing, deploying, and managing Web services in OpenEdge, see OpenEdge Web services architecture on page 79 and Tools for building and managing OpenEdge Web services on page 80.

Benefits of producing Web services using OpenEdge

The OpenEdge provides the most practical way to Web service-enable AppServer application services. OpenEdge supports Web services in full compliance with industry standards and adds capabilities that uniquely enable both existing and new AppServer application services to be consumed as Web services. Taken together, OpenEdge provides the benefits described in the following sections:

- Support for industry standards on page 78
- Language and platform independence on page 78
- Opportunities for reusing ABL business logic on page 79
- Standard, well-defined, and flexible client interface model on page 79
- More consistent client development model on page 79

Support for industry standards

Web services built with OpenEdge support current industry standards for WSDL, SOAP, and XML Schema. For more information on the specific Web service standards supported by OpenEdge, see OpenEdge Development: Web Services.

Language and platform independence

Web services built with OpenEdge permit language and platform-independent access to ABL business logic from non-ABL clients. Thus, the passing of data between the client and OpenEdge Web service application is largely transparent to the client language or platform.

Note: The exchange of certain dynamic parameters requires additional information about the data definitions that can be conveyed, in a language-independent fashion, to the client using standard XML schema type definitions.
Opportunities for reusing ABL business logic

A Web service implementation can protect your development investment by supporting future application integration through increased usability of existing ABL business logic. Packaging an AppServer application as a Web service opens it up for integration with a potentially infinite number of other applications.

Standard, well-defined, and flexible client interface model

Web services, generally, support a standard and well-defined, yet flexible, interface model for client development. This allows clients written in almost any language and running on almost any platform that supports the requirements defined by WSDL to access ABL business logic.

More consistent client development model

OpenEdge supports a more consistent client development model than what is available for other Open Client applications. Once a Web service is deployed and available, client applications can be developed on any platform that supports Web service consumption using the same WSDL file. Other Open Clients require a different Open Client proxy for each supported Open Client platform (.NET or Java, for example).

OpenEdge Web services architecture

The following figure shows the basic Web services architecture shown in Figure 19 on page 70, with the generic components replaced by corresponding OpenEdge components. In the OpenEdge version of the architecture, the WSA provides the SOAP-to-business interface gateway, which in this case is a gateway between SOAP and the application service provided by the AppServer.
In this figure, the set of components required for Web services appear with gray shading, and the OpenEdge-implemented members of this set appear with darker gray shading. Otherwise, the OpenEdge Web services architecture provides features that are virtually identical to any other Web services architecture in the industry. OpenEdge supports these features with the components of the Web service tools architecture. Thus, OpenEdge supports all phases of Web service development, deployment, management, and client run-time access. OpenEdge also supports secure intranet connections between the WSA and AppServer using the Secure Sockets Layer (SSL). For more information, see *OpenEdge Getting Started: Core Business Services - Security and Auditing*.

### Tools for building and managing OpenEdge Web services

OpenEdge supports the full range of OpenEdge Web services development, deployment, and administration, including:

- **Web service development tools** on page 81
- **WSA configuration and Web service deployment tools** on page 81
- **Client development tools** on page 84
- **Run-time tools** on page 84
- **Security tools** on page 86
- **Integrated Web services tool set** on page 86
Web service development tools

The following figure shows OpenEdge tools that you use to build a Web service, using the WorldWeather Web service (see Web services in action on page 73) as an example.

Figure 23: OpenEdge Web service development tools

You can first build an AppServer application using OpenEdge development tools from OpenEdge Studio, such as the Procedure Editor, OpenEdge AppBuilder, Progress Dynamics®, or even a simple text editor and ABL compilation scripts. Similar to developing a .NET or Java Open Client, you compile the completed AppServer procedures (or their prototypes) and provide the resulting r-code as input to the OpenEdge Open Client Proxy Generator (ProxyGen) utility. However, instead of generating Open Client proxies, ProxyGen generates the Web service (client interface) definition in the form of a Web service mapping (WSM) file used during Web service deployment. Optionally, you can also generate a WSDL file to develop a Web service client prior to deploying the Web service for testing and production access. For more information on client development and testing, see Client development tools on page 84.

In the example, your AppServer application includes procedures, such as getTemp.p, that combine to form the interface of the WorldWeather Web service. In ProxyGen, you define the WorldWeather project (WorldWeather.xpxg) from this interface and generate the WSM file (WorldWeather.wsm) to deploy the Web service and generate the optional WSDL file (WorldWeather.wsdl) to develop client code for testing Web service development.

For more information on how the AppServer supports Web services, see OpenEdge Application Server: Developing Applications. For more information on how the Open Client object model and ProxyGen support Web service development, see OpenEdge Development: Open Client Introduction and Programming and OpenEdge Development: Web Services.

WSA configuration and Web service deployment tools

The following figure shows OpenEdge tools used to configure and manage a WSA and to deploy and manage Web services.
The WSA is the central component for all aspects of Web service deployment, configuration, and run-time management. As Figure 24 on page 82 shows, you can manage the entire WSA and Web service configuration using the Unified Broker framework, which accesses the WSA as an OpenEdge server to which Web services are deployed. For more information on the Unified Broker framework and how to work with OpenEdge servers, see OpenEdge Getting Started: Installation and Configuration.

Note: The AdminServer shown in Figure 24 on page 82 can reside on the same machine as the WSA (local) or on a separate machine (remote).
On the Web server, the WSA is installed as a single JSE Web application for which you must create instances to do meaningful work. The dashed arrows in the Figure 24 on page 82 indicate that each WSA instance is created from its definition in the ubroker.properties file, which is read by WSA to create the instance. When you deploy a Web service, you deploy it to a WSA instance, which defines the root Universal Resource Locator (URL) used to access the Web service and handles all of its client communications. Each WSA instance thus manages its own set of deployed Web services.

The WSA comes installed with a default WSA instance, wsa1, but no default deployed Web services. You can deploy sample Web services provided with the OpenEdge installation. For more information on these samples, see OpenEdge Development: Web Services.

To begin using the WSA, you must configure it using OpenEdge Management/OpenEdge Explorer or the WSAMAN command-line utility. (This assumes you already installed and configured the JSE to recognize the WSA as a Web application. For more information, see OpenEdge Application Server: Administration.) Similar to other OpenEdge servers, you can create one or more uniquely named WSA instances and set properties for each one in the WSA's ubroker.properties file.

For each new WSA instance that you create, two files are created, default.props and DeployedServices.ds. The default.props file contains the default values to set for the run-time properties of each Web service that you deploy to the instance, and the DeployedServices.ds file contains the identity and deployment information of each Web service that is already deployed to that instance.

**Note:** Before using a WSA instance that is newly created using OpenEdge Management or OpenEdge Explorer, you must register the instance with the JSE as a servlet and restart the JSE to run it.

Once you have created and started up a WSA instance, you can deploy a Web service to it using OpenEdge Management/OpenEdge Explorer or the wsaman utility. Input for the deployment includes the WSM file generated for the Web service by ProxyGen (WorldWeather.wsm, in Figure 24 on page 82) and additional deployment information that you specify, such as a friendly name (FriendlyName) for the Web service. The deployment output includes three files created in the context of the WSA instance:

- **FriendlyName.props** — WorldWeather.props in Figure 24 on page 82, the Web service properties file containing the Web service run-time property values, initially set to the values in the WSA instance's default.props file.

- **FriendlyName.wsdl** — WorldWeather.wsdl in Figure 24 on page 82, the WSDL file that contains the Web service definition. This definition describes to a Web service client how to code and decode SOAP messages for the Web service, and specifies the interface for operations that the client must use to send these messages.

- **FriendlyName.wsad** — WorldWeather.wsad in Figure 24 on page 82, the Web services application descriptor (WSAD) file that defines the Web service to the WSA instance. This file describes the Web service to the WSA so that it can code and decode client SOAP messages for the Web service, and specifies how to access the AppServer application that executes SOAP requests.

Thus, Figure 24 on page 82 shows the WorldWeather Web service deployed to the WSA instance, wsa1, and also another WSA instance, wsa2, that has no Web services deployed to it.

For more information on Web services administration in OpenEdge, see OpenEdge Application Server: Administration.
Client development tools

The following figure shows OpenEdge tools used to develop a Web service client to consume OpenEdge Web services.

**Figure 25: Web service client development tools**

Typically, you, as the client developer, obtain the WSDL file for the Web service (WorldWeather.wsdl in the above figure) by downloading it across the Internet or intranet.

Once you have the WSDL file, and any other required documentation on how to use the Web service interface, you can create the client application to access the Web service. Typically, Web service client toolkits provide a client interface generator that reads the WSDL file and generates source code for the client interface objects, which you can then access directly from the client application. These objects provide the interface that the client application needs to call Web service operations. Together with any SOAP library functions that the client requires, they generate and consume the SOAP messages to access and interact with the Web service.

At a minimum, you might have to manually read the WSDL file and hand-code the Web service client interface from it, using a library to handle the SOAP messages. But this is becoming a rare requirement, and no longer necessary using many Web service client platforms (such as Microsoft .NET, Iona XMLBus™, or OpenEdge). The OpenEdge Web services client platform provides the WSDL Analyzer to generate complete documentation for the client interface.

For more information on OpenEdge support Web services clients in general as well as the ABL client in particular, see *OpenEdge Development: Web Services*.

Run-time tools

The following figure shows tools used to run and test a Web service together with a Web service client.
Run-time tools and the Web service URL

When running and testing an OpenEdge Web service and its client, it is helpful to understand how the tools involved in Web service execution are represented in the Web service URL. There are up to three major components between a Web service client and the Web service it tries to access. All of these components typically participate in the URL path to the Web service and its WSDL file. Listed in their order of appearance in the URL, these components are:

- **Web server and JSE** — Provides the HTTP listener and the JSE to run the WSA. This component optionally participates in the URL, depending on how you install and configure it:
  - **Web-server-based** — A full-featured Web server together with a JSE that is integrated with or installed separately from the Web server.
  - **JSE-based** — A stand-alone JSE installed with an integrated HTTP listener.

If this component has a Web-server-based configuration, it participates in the URL, mapping to the Web server context, which includes the connection between the Web server and JSE. If it has a JSE-based configuration, there is no URL path component to represent a Web server connection to the JSE. In Figure 26 on page 85, the Web server is separate from the JSE, and its connection to the JSE is represented by its own URL path component, bedrock.
• **Web application (WSA)** — Provides the WSA context for OpenEdge (see Figure 24 on page 82 and the WSA area in Figure 26 on page 85). This is the global context for all WSA instances that run under the control of the same WSA Web application, and is represented in Figure 26 on page 85 by the URL component, *quarry*.

• **Servlet (WSA Instance)** — Defines the context of each WSA instance, through which individual Web services are deployed and accessed. In Figure 26 on page 85, there are two WSA instances, and they are represented by the URL path components, *fred* and *wsa2*.

Thus, a root URL for the WSA instance, *fred*, maps neatly to the Web service run-time architecture shown in Figure 26 on page 85, as in the following example:

```
http://servicehost:80/bedrock/quarry/fred
```

### WSA as a run-time tool

At run time, each SOAP request to a Web service is handled by the specified WSA instance. The WSA decodes and encodes SOAP messages according to the Web service interface defined in the WSAD file (*WorldWeather.wsad* in Figure 24 on page 82). It also manages Web service run-time resources as defined by the Web service property settings (*WorldWeather.props* in Figure 24 on page 82).

### Tools for testing and debugging

Another tool that you might use to test and debug a Web service client is a SOAP viewer. You can configure a SOAP viewer to intercept all messages sent between the Web service client and the WSA instance. You can then view and possibly modify them (depending on the viewer) before you allow them to continue to their destination. Depending on the viewer, you might have to alter the Web service URL in the client to accommodate the viewer. OpenEdge provides SOAP viewing tools with the installation, but they are also widely available. For more information on testing and debugging Web services and Web service clients, see *OpenEdge Development: Web Services*.

### Security tools

OpenEdge supports a variety of authentication, authorization, and access control options that can secure a Web services installation at one or more levels of access, from the entire Web server context to individual WSA administration functions and general access control for individual Web services. For more information, see *OpenEdge Development: Web Services* and *OpenEdge Application Server: Administration*.

### Integrated Web services tool set

The following figure shows how the tool set for OpenEdge Web services is integrated in OpenEdge.
In the above figure, the tool set required for OpenEdge Web services appear with gray shading, and the OpenEdge-implemented members of this tool set appear with darker gray shading. The unshaded components participate as "users" (such as Client dev tools in the figure) or "enablers" (such as AppServer 1) of Web services that are produced and deployed with OpenEdge. The components outlined in dashes and related with white arrow heads show optional elements, which in OpenEdge are useful for preparing client code to test Web service development. The dashed arrow heads in black show virtual relationships, where data moves, but not the actual path that it takes through the network.

Note that the major activities labeled in this architecture take place in different times and spaces, as described in previous sections of this chapter. For example, the OpenEdge Web service development activity occurs in a totally separate context from the WSA configuration or Web service deployment activities. Only after Web service development is complete does the noted .wsm file become input to OpenEdge Management or OpenEdge Explorer for Web service deployment.
For a step-by-step description of how to develop, deploy, and manage OpenEdge Web services in OpenEdge, see *OpenEdge Development: Web Services*. 
OpenEdge® provides the OpenEdge Adapter for SonicMQ® to implement Java Message Service (JMS) messaging in any OpenEdge application using SonicMQ®, the reliable messaging application from Progress Software Corporation. It also provides the OpenEdge Adapter for Sonic ESB® to allow OpenEdge ABL procedures to be incorporated into ESB processes on the Sonic Enterprise Service Bus. ABL procedures are accessed through Sonic ESB either directly when using the Native Invocation methodology, or as a Web Service when using the Web Service Invocation methodology.

This chapter describes the architecture and where to find further information on using these adapter products.
For details, see the following topics:

- OpenEdge Adapter for SonicMQ
- OpenEdge Adapter for Sonic ESB

OpenEdge Adapter for SonicMQ

Application programmers can utilize JMS messaging from ABL (Advanced Business Language) with the OpenEdge Adapter for SonicMQ, allowing the OpenEdge application to exchange messages with applications, including applications written in other languages, that also use SonicMQ. You can write ABL applications that access the OpenEdge Adapter for SonicMQ through an ABL-JMS API that works the same on all platforms and in all configurations (GUI, character, WebClient, AppServer, WebSpeed, and batch).
ABL-JMS API

The ABL-JMS API is strongly integrated with the ABL programming model and style, including the ABL event model. The API provides ABL persistent procedures to represent the JMS connection, session, and Message objects. The ABL programmer uses the methods in these procedure objects for JMS message delivery, acknowledgement, and recovery. The API supports the two basic types of JMS messaging:

- **Point-to-Point (PTP)** — (one-to-one communication) A producer sends a message to a queue, where it is received by a single consumer, no matter how many consumers are listening to the queue.

- **Publish and Subscribe (Pub/Sub)** — (one-to-many communication) A producer publishes a message to a topic, where all consumers subscribing to the topic receive the message.

Messages are processed when the application is in a `WAIT FOR` or other IO-blocking state. To facilitate non-UI applications, such as AppServer processes or batch processes, the OpenEdge application can also call the ABL-JMS API `waitForMessages` procedure to process messages. OpenEdge uses the existing ABL error mechanisms to handle ABL-JMS errors.

An OpenEdge application written to take advantage of the ABL-JMS API can talk with another application without knowing whether it is an ABL or a non-ABL application. Java features are mapped to ABL; for example, Java `float` to ABL `DECIMAL`. For ABL-to-ABL messaging, an application can package ABL data within standard messages, for example, to send a static or dynamic temp-table.

OpenEdge Adapter for SonicMQ architecture

The OpenEdge Adapter for SonicMQ consists of the following three connection modes:

- OpenEdge Adapter for SonicMQ ClientConnect
- OpenEdge Adapter for SonicMQ ServerConnect
- OpenEdge Adapter for SonicMQ BrokerConnect

OpenEdge Adapter for SonicMQ ClientConnect

The OpenEdge Adapter for SonicMQ ClientConnect (ClientConnect) architecture combines the adapter with the OpenEdge client process. ClientConnect runs as a process started and managed by OpenEdge clients directly when an application requires messaging. There is a single adapter process per client process, as shown in the following figure. The SonicMQ Broker acts as a service point for all JMS sessions.
ClientConnect provides a direct connection between an ABL client and the SonicMQ Broker. This connection provides complete and reliable end-to-end messaging, and supports message fault tolerance. Additionally, ClientConnect supports JMS domain unification, client persistence, server-based message selectors, serialized connection objects, temporary destinations, TempTable messages, DataSet messages, and enhanced XML support.

**OpenEdge Adapter for SonicMQ ServerConnect**

The OpenEdge Adapter for SonicMQ ServerConnect (ServerConnect) architecture combines the adapter with the OpenEdge Servers (WebSpeed and AppServer). ServerConnect runs a single adapter process per ubroker process, allowing multiple server processes to connect to this single adapter process, as shown in the following figure. The SonicMQ Broker acts as a service point for all JMS sessions.
ServerConnect is a direct connection to the SonicMQ Broker. This connection provides complete end-to-end messaging. Additionally, ServerConnect supports JMS domain unification, client persistence, fault tolerant connections, server-based message selectors, serialized connection objects, temporary destinations, TempTable messages, DataSet messages, and enhanced XML support.

Progress Software Corporation recommends that you use ServerConnect if your AppServer will be doing messaging. This way, you have a single server process to manage and access to features such as fault tolerance. (As compared to BrokerConnect, discussed below, that requires two server processes and is not fault tolerant.) You can use particular AppServer or WebSpeed servers to manage particular SonicMQ connections, minimizing the number of SonicMQ connections required. The embedded process that runs inside of the AppServer or WebSpeed server is multi-threaded and allows for multiple SonicMQ connections within the same process.

OpenEdge Management or OpenEdge Explorer allows you to manage the ubroker properties required for ServerConnect.

**OpenEdge Adapter for SonicMQ BrokerConnect**

The OpenEdge Adapter for SonicMQ BrokerConnect (BrokerConnect) architecture consists of both a Server to the OpenEdge Client and a Client to the SonicMQ Broker. BrokerConnect requires the client to access the adapter container by specifying connection parameters for the controlling NameServer or OpenEdge Adapter for SonicMQ instance (if not registered with a controlling NameServer) when the JMS session is created.

Each instance of BrokerConnect is a single broker process running one or more client threads (servers in OpenEdge Management or OpenEdge Explorer), as shown in the following figure. Each client thread talks to an OpenEdge application on one side and a SonicMQ Broker on the other side. BrokerConnect acts as the JMS client to the SonicMQ session. You can also start more than one BrokerConnect instance under a single controlling NameServer.
Figure 30: OpenEdge Adapter for SonicMQ BrokerConnect

BrokerConnect is not a direct connection to the SonicMQ Broker. Therefore, BrokerConnect cannot provide the Quality of Service (QOS) between an ABL client and the SonicMQ Broker that ClientConnect and ServerConnect offer.

BrokerConnect is managed by the Unified Broker framework, which includes OpenEdge Management/OpenEdge Explorer and related command-line utilities.

BrokerConnect supports JMS domain unification, server-based message selectors, serialized connection objects, temporary destinations, TempTable messages, DataSet messages, and enhanced XML support. BrokerConnect does not support fault tolerance and client persistence.

You should use BrokerConnect if you are concerned about the size of your ABL session or server process, or machine resources.

Note: BrokerConnect is functionally equivalent to the OpenEdge Adapter for SonicMQ architecture supported in releases earlier than Release 10.1A.

OpenEdge Adapter for Sonic ESB

Sonic ESB is an industry-leading application integration framework that provides exceptional performance, reliability, and security. Its service-based architecture supports the deployment of discrete applications (services) that exchange messages according to sophisticated automated workflow processes. It also supports the exposure of deployed applications as standard Web services that can process requests sent through HTTP or HTTPS (HTTP/S) from any compliant client application external to the Sonic ESB.
The OpenEdge Adapter for Sonic ESB enables AppServer applications to participate as services on the Enterprise Service Bus (ESB) and take part in workflow processes managed by the Sonic ESB. The OpenEdge Adapter for Sonic ESB supports two distinct methodologies:

- Native Invocation methodology
- Web Service Invocation

**Note:** Prior to OpenEdge Release 10.1C, the Web Service Invocation methodology was the only available option.

### Native Invocation methodology

When a Sonic ESB process calls an OpenEdge service using the Native Invocation methodology, ABL procedures are called directly via an OpenAPI call to an OpenEdge Application Server. The Native Invocation methodology depends on an invocation (.esboe) file. An .esboe file is typically based on annotations within your ABL source, but can also be generated from Proxygen, and is imported directly into Sonic Workbench.

The Native Invocation methodology provides the following benefits:

- Simplified exposure of ABL code as a service
- Simplified process of mapping ABL parameters to Sonic messages
- Reduced overhead, improving run-time performance over Web Service Invocation methodology, by eliminating conversions to and from SOAP messages

### Declarative vs. non-declarative approach

OpenEdge developers have the choice of two approaches for exposing ABL procedures and functions for the Native Invocation methodology:

- **Declarative** — Captures information about publicly exposable procedures in the source code through the use of annotations
- **Non-declarative** — Captures information about publicly exposable procedures through the use of a tool such as ProxyGen

The declarative approach is recommended to OpenEdge developers as a best practice. When following the declarative approach, developers enter relevant information for making a procedure, function or external procedure publicly exposable when the source code is written. This information is then stored with the source code, and during the build process captured as an invocation (.esboe) file. In cases where the developer does not want to capture information about publicly exposable procedures with the source code, the non-declarative approach is supported.

### Annotations

Annotations are a methodology to capture information in the source code that extends the ABL language syntax. Validation is available to confirm that the information entered adheres to the pre-defined syntax and format.
At a minimum the required annotation is simply a declaration that the ABL procedure is to be publicly available. The best practice is to add annotations to the ABL source code, through the use of Progress Developer Studio for OpenEdge Annotation wizard. However, annotations can be added to the source by whatever means the developer uses to create the source code. For information on the syntax of a Sonic ESB annotation, or instructions on adding annotations with OpenEdge, see *OpenEdge Development: Messaging and ESB.*

.esboe files

The Native Invocation methodology of the OpenEdge Adapter for Sonic ESB, depends on an invocation file for the easy integration of services in Sonic ESB business process via itineraries. For ABL procedures and functions, this invocation file is the .esboe file. A .esboe file supplies the information necessary for Sonic to make a call to an OpenEdge Application Server to execute a single ABL procedure. Once created, the .esboe file can be dropped onto the canvas of the Sonic ESB Process Editor in Sonic Workbench to create a new step in an ESB process. For more information on generating an ESB process, see your Sonic documentation.

Creation of the invocation file depends on the developer’s choice of a declarative or non-declarative approach to development. The following table details the options.

<table>
<thead>
<tr>
<th>Source code development approach (declarative/ non-declarative)</th>
<th>Source code development tool</th>
<th>.esboe file generation</th>
</tr>
</thead>
</table>
| Declarative (annotated code)                                 | Progress Developer Studio for OpenEdge | • Automatically generated when source is saved to disk, when the project has **Build Automatically** selected  
• Automatically generated when **Clean** is selected for the project, and **Build Automatically** is selected  
• Automatically generated when the source is compiled |
|                                                               | Text editor, or other source code generator other than Progress Developer Studio for OpenEdge | Run the **ESBOEGEN** utility using compiled source code. |
| Non-declarative (code not annotated)                        | Any                         | Run **ProxyGen** using compiled source code. |

For details on using creating .esboe files, see the *OpenEdge Development: Messaging and ESB* guide.
Web Service Invocation methodology

When employing the Web Service Invocation methodology, OpenEdge services are transformed into OpenEdge Web services that are deployed (installed) into a Sonic ESB container. At run time the OpenEdge Adapter for Sonic ESB receives SOAP messages from the SonicMQ Broker, converts them to ABL, and sends them to the AppServer to be processed. The adapter then receives ABL messages from the AppServer, converts them to SOAP, and returns them to the broker where they are then routed according to a Sonic ESB process itinerary or a Web service reply mechanism, as appropriate.

For more information on OpenEdge Web services and how they are created, see Web Services—Architecture and Tools on page 67

In addition, for each OpenEdge service, the OpenEdge Adapter for Sonic ESB processes the WSM (or WSD) file. These files are described in the following sections.

WSM files

The Web Service Mapping (WSM) file contains the XML-encoded instructions that enable the OpenEdge Adapter for Sonic ESB to process SOAP messages according to the needs of the OpenEdge application. A WSM file is created for each service using the OpenEdge Open Client Proxy Generator tool (ProxyGen). For more information on ProxyGen and how to create WSM files, see OpenEdge Development: Open Client Introduction and Programming.

In the process of installing an OpenEdge service in the Sonic ESB environment, the OpenEdge Adapter for Sonic ESB takes the WSM file as its primary input.

WSDL files

The Web Services Description Language (WSDL) file is an optional XML file generated by ProxyGen and used by the Sonic Web Services Invocation framework to generate a proper SOAP message. It contains information required by programmers of client applications that will call the service. It is a standard component widely used in the industry for development of Web service clients, however the WSDL file generated by ProxyGen is specifically for use with Sonic’s Web Service Invocation and is not appropriate for use with outside clients. WSDL files are stored as resources in Sonic ESB and can be viewed with Sonic Workbench. For more information on the WSDL files generated for the OpenEdge Adapter for Sonic ESB and how Web service clients must use them, see OpenEdge Development: Web Services.

The following figure is a simplified illustration of the function of the OpenEdge Adapter for Sonic ESB in the context of Sonic ESB. The components and functions illustrated in the this figure are explained in more detail in the sections that follow.
Figure 31: An example of the OpenEdge Adapter for Sonic ESB in context

Sonic Management Console

The Sonic Management Console (SMC) is the graphical interface to a set of tools for managing the entire Sonic messaging and application environment, including both Sonic ESB and the required underlying messaging infrastructure, SonicMQ. SMC is the primary means of performing administration and configuration tasks for all services and the elements (such as process definitions) and resources (such as service definitions) that support them.

A command-line tool, called the ESB Admin Tool, is also available for most of these administrative tasks.
Note: The OpenEdge Adapter for Sonic ESB does not appear as such in the Sonic Management Console. It is the presence of the two OpenEdge Services entries (OpenEdge Native Services and OpenEdge Web Services) in the ESB Configured Objects that indicates that the adapter is installed. Furthermore, the OpenEdge Adapter for Sonic ESB does not appear in the user interface of any OpenEdge tool.

For more information on these Sonic ESB management tools, see the Sonic ESB product documentation.

Sonic ESB container

All Sonic ESB services are deployed to a Sonic ESB container. The container hosts various components, including ESB processes, in addition to services. It also provides a number of support functions.

The example in Figure 31 on page 97 shows two OpenEdge services, supported by the OpenEdge Adapter for Sonic ESB, and two non-OpenEdge services. For Service1, OpenEdge has an associated WSM file and an associated WSDL file, and uses the Web Service Invocation methodology. For Service2, OpenEdge has an associated .esboe file, and uses the Native Invocation methodology.

For more information on the Sonic ESB container, see the Sonic ESB product documentation.

SonicMQ container

SonicMQ is the messaging backbone on which the Sonic ESB relies for communications. Each Sonic ESB container must be deployed to a SonicMQ container.

For more information on the SonicMQ container, see the SonicMQ product documentation.

SonicMQ Broker

The SonicMQ messaging system is based on one or more brokers that establish connections with clients, perform routing functions, and implement security measures. Every client of SonicMQ has a persistent connection to a SonicMQ Broker.

As illustrated in Figure 31 on page 97, a SonicMQ Broker can be configured with an HTTP/S acceptor. In effect, such an acceptor lets the broker function as a Web server, enabling Sonic ESB to expose its services as standard Web services that process requests received via the Internet from external clients.

For more information on the SonicMQ Broker, see the SonicMQ product documentation.

AppServer

The AppServer is the OpenEdge facility for the storage and execution of business logic, including OpenEdge services executed by Sonic ESB. For each client request to a service, the AppServer executes the application code, processing the input from the client and returning results and errors accordingly.
Although the illustration in Figure 31 on page 97 represents the AppServer as a single entity, this is a simplification of the server architecture. The AppServer launches AppServer agents, which run as separate processes controlled by the AppServer broker and actually perform the run time functions. The AppServer often works in conjunction with a NameServer, which provides directory and routing services.

Note that the connection between the OpenEdge Adapter for Sonic ESB and the AppServer can be tunneled through an OpenEdge implementation of the Secure Sockets Layer (SSL) to provide AppServer intranet connection security in addition to any security managed by Sonic ESB.

For more information on the AppServer, see Chapter 1, AppServer for OpenEdge Applications on page 1. For more information on secure connections to the AppServer, see OpenEdge Getting Started: Core Business Services - Security and Auditing.

OpenEdge services—usage and management

With Sonic ESB, an OpenEdge service can be accessed directly as a Web service by an industry Web service client, or it can participate in workflow processes managed by Sonic ESB. The functions of the OpenEdge service are managed and made accessible entirely within the Sonic ESB environment. For more information on how to access and use an OpenEdge service, see OpenEdge Development: Messaging and ESB. For information on how to deploy (install) and manage OpenEdge services in the Sonic ESB environment, see the information on managing the OpenEdge Adapter for Sonic ESB in OpenEdge Application Server: Administration.
This chapter is an overview of WebSpeed components and how they function. For details, see the following topics:

- Introduction to WebSpeed
- Advantages of WebSpeed
- WebSpeed components
- How WebSpeed handles a request

## Introduction to WebSpeed

WebSpeed is both a development and a deployment platform for Web applications. Web applications are accessed by end users through a Web browser. WebSpeed Web applications typically involve some interaction with a data source. You can implement queries, updates, and the addition or deletion of records.

With WebSpeed, you can develop and deploy:

- Intranet applications that allow internal users to access and modify data
- Internet applications that allow external, consumer access (for example, a shopping cart application)
- Extranet, business-to-business applications
WebSpeed includes examples of all three types of application. See *OpenEdge Getting Started: WebSpeed Essentials* for a description of the intranet, internet, and extranet sample applications and instructions on how to run them.

**Advantages of WebSpeed**

Some of the advantages of WebSpeed as an application development and deployment environment are:

- Native database support includes access to OpenEdge, ORACLE, and MS SQL Server, plus ODBC access to DB2, Sybase, MSAccess, Informix, and other data sources.
- WebSpeed is compatible with any ISAPI, NSAPI, or CGI-compliant Web servers.
- Wizards and templates can be used to create professional, multi-page Web applications with little or no coding, significantly cutting development time while preserving the ability to customize your application.
- A flexible programming model lets you structure development based on your project requirements, embedding SpeedScript directly into your HTML pages or using HTML mapping to bind HTML files to business logic.
- Scripting Lab enables you to test code fragments quickly before adding them to your WebSpeed application, increasing productivity and helping you create feature-rich user interfaces and browser-side validation routines using JavaScript, VBScript, and Java applets.
- An open architecture integrates with your choice of security solutions, including firewall, authentication, and encryption technologies using HTTPS. In addition, you can encrypt Web application messages within the intranet on the WebSpeed end to maintain local data privacy.
- A high-performance, scalable architecture delivers fast transaction processing and handles thousands of simultaneous users under peak load.
- Dynamic load balancing ensures high availability of transaction processing resources in a distributed, multi-tier environment.
- Flexible state management offers full support for extended database queries and updates using stateless, state-passing, or state-persistent Web objects.

For information about developing and deploying WebSpeed applications see *OpenEdge Application Server: Developing WebSpeed Applications*. For information about configuration and security issues, see *OpenEdge Application Server: Administration* and *OpenEdge Getting Started: WebSpeed Essentials*.

**WebSpeed components**

This section is a overview of the various components that comprise the WebSpeed architecture.

For information on how to install WebSpeed components, see *OpenEdge Getting Started: WebSpeed Essentials*. For information on how to configure these components after installation, see *OpenEdge Application Server: Administration*. 
Note: In addition to the components described in this section, your WebSpeed development or deployment environment requires a Web server and a Web browser. These components are not supplied with OpenEdge. Your WebSpeed application will also require a data source, which can be an OpenEdge RDBMS or a DataServer.

WebSpeed Messenger

The WebSpeed Messenger is a process running on a Web server that finds a WebSpeed Transaction Server to handle a client request. It can be configured to use a NameServer to find an appropriate Transaction server, or to link directly to a specific Transaction server.

It also handles the transfer of data between the agent and a Web server during a Web transaction. The Messenger is a CGI program or an ISAPI or NSAPI process, depending on the Web server and how you configure it.

While the Messenger participates in any HTTPS messaging to secure Web transactions over the Internet between the Web browser and Web server, OpenEdge also provides a native implementation of the Secure Sockets Layer (SSL) to secure the transaction between the Messenger and the WebSpeed Transaction Server on the intranet. For more information on OpenEdge SSL, see OpenEdge Getting Started: Core Business Services - Security and Auditing.

NameServer

The NameServer is a process that maintains a list of WebSpeed Transaction Servers. It is called by the WebSpeed Messenger in response to a client request.

Transaction servers register with the NameServer and inform the NameServer about the application services they provide. The NameServer can then direct client connection requests to a Transaction server that supports a requested application service. This provides scalability and location transparency to your applications.

Use of the NameServer is optional. However, without a NameServer a Messenger can only connect to a single Transaction server.

WebSpeed Transaction Server

At the core of WebSpeed is the Transaction server. This is the component that responds to a client request by running a WebSpeed application (also known as a Web object) and by formatting the HTML that is sent back to the requesting client.

The WebSpeed Transaction server is comprised of two components, a broker and an agent.

WebSpeed broker

The broker is a process that does the following:

- Registers the WebSpeed services that it provides to a NameServer for access by one or more HTML clients. The HTML client runs from within an Internet browser. (Note that the broker can also run without the NameServer. It can connect directly to the WebSpeed Messenger.)
- Manages connections between clients and a pool of WebSpeed agents.
• Maintains the status of each agent in its pool and dynamically scales the number of agents according to changing demand.

**WebSpeed agent**

The agent is a process that executes Web objects and performs database transactions. It dynamically creates the HTML that is sent to the client.

WebSpeed agents commonly run in stateless mode and are freed after they complete a client request. However, it is also possible to lock an agent to a client in order to maintain context between requests. In addition, you can preserve context using cookies, URL query strings, hidden HTML form fields, and connection identifiers.

**WebSpeed Workshop**

WebSpeed Workshop is the collective name for the tools and utilities that support Web application development. WebSpeed Workshop also includes a WebSpeed Transaction Server that runs in development mode for testing and running Web objects.

For more information about using WebSpeed Workshop for application development, see *OpenEdge Getting Started: WebSpeed Essentials*.

**AppBuilder**

The AppBuilder is a graphical tool that contains wizards, templates, compilers, and editors for building Web Objects. The AppBuilder is also a launching point for various database administration tools (the Data Dictionary, for example).

**WebTools**

WebTools is a set of browser-based utilities for managing your development environment and for testing your Web applications.

**SpeedScript**

SpeedScript is an interpreted, block-structured, and statement-oriented programming language based on ABL (Advanced Business Language). It provides native access to the OpenEdge RDBMS and DataServers, as well as to Web servers for Web page input and output.

SpeedScript enables you to quickly prototype, develop, and maintain re-usable application business logic. It also includes object-oriented features like super procedures and dynamic queries and buffers.

Depending on your application and the type of Web objects you are building, the AppBuilder can automatically generate some or all of the SpeedScript used to build your Web objects.

**JavaScript**

WebSpeed supports JavaScript as well as SpeedScript for application development.
It is a common practice to use both SpeedScript and JavaScript when developing WebSpeed applications. SpeedScript has advantages for developing the business logic of applications, while JavaScript is a good programming tool for adding user interface elements to Web applications.

**Progress Developer Studio for OpenEdge**

Progress Developer Studio for OpenEdge is an Eclipse-based OpenEdge product that supports creating and managing WebSpeed applications. Features include:

- A WebSpeed project type and a workspace that hosts the tools, editors and views that support the development and deployment of WebSpeed applications
- Templates and wizards for creating SpeedScript and CGI Wrapper files
- Full-featured editors for editing HTML, Embedded SpeedScript, and CGI Wrapper files
- Support for publishing WebSpeed applications to local and remote servers
- Support for running, testing, and debugging WebSpeed applications on local and remote servers
- Support for configuring and managing local and remote servers

In many respects, OpenEdge Developer Studio is an excellent alternative to the AppBuilder for WebSpeed application development. The ability to instantly publish your code to a variety of servers is a great advantage that is not available in other tools. However, OpenEdge Developer Studio does not support the HTML Mapping coding model.

For more information, see the Progress Developer Studio for OpenEdge help in the Development Tools section of the OpenEdge Product Documentation Overview page on Progress Communities: https://community.progress.com/technicalusers/w/openedgegeneral/1329.openedge-product-documentation-overview.aspx.

**Configuration and management utilities**

This section provides an overview of the management tools for WebSpeed. For more detailed information, see *OpenEdge Application Server: Administration*.

**OpenEdge Management or OpenEdge Explorer**

OpenEdge Management/OpenEdge Explorer is a browser-based tool that combines the functionality of all the command-line utilities. You can use it to start, create, delete, and modify WebSpeed brokers, agents, NameServers, and Messengers. You can also use OpenEdge Management or OpenEdge Explorer to start, create, delete, and modify database servers.

**Command-line utilities**

Command-line utilities are available for both the UNIX and Windows operating systems.

The **WTBMAN** utility allows you to control the operation and query the status of a configured WebSpeed Transaction Server. The **WSCONFIG** utility displays and validates property settings for the WebSpeed Transaction Server and the WebSpeed Messenger.

The **NSMAN** utility allows you to control the operation and query the status of a configured NameServer. The **NSCONFIG** utility displays and validates property settings for the NameServer.
The PROADSV utility (only on UNIX systems) allows you to start up and shut down the AdminServer.

**SpeedStart**

SpeedStart is a browser-based reference page that contains a list of URLs that you can use to check if WebSpeed components are running properly. You can access SpeedStart from your Windows Start Menu (Programs > Start > OpenEdge > SpeedStart).

**How WebSpeed handles a request**

This section describes how the components of the WebSpeed Transaction Server interact to respond to a request from a client browser.

1. The Web server receives the request from a Web browser. Each request is in the form of a URL, as shown:

```
+----------------+    +----------------+
|  Web browser   |    |   Web server    |
```

2. The Web server sends the request to a WebSpeedMessenger (either CGI, ISAPI, NSAPI, or WSASP), as shown:

```
+----------------+    +----------------+    +----------------+
|  Web browser   |    |   Web server    |    |   WebSpeed      |
       |              |    |                  |
```

3. The WebSpeed Messenger communicates with the NameServer, requesting an appropriate WebSpeed broker, as shown:

```
+----------------+    +----------------+    +----------------+    +----------------+
|  Web browser   |    |   Web server    |    |   WebSpeed      |    |   NameServer    |
       |              |    |                  |    |                  |    |                  |
```

**Note:** The NameServer can be eliminated. It is possible to configure the WebSpeed Messenger to contact the broker directly. However, if the NameServer is eliminated, the Messenger can only contact a single broker.

4. The Messenger initiates a connection to the WebSpeed broker at the address returned by the NameServer and forwards the request to that broker, as shown:
Note: If load balancing is enabled, the NameServer can manage multiple WebSpeed brokers that support the same application service. It will choose a broker based on load-balancing criteria.

5. The WebSpeed broker consults its pool of WebSpeed agents and assigns the request to an idle agent, as shown:

6. The WebSpeed agent and the WebSpeed Messenger establish a connection. The Web object name and the environment for the request are passed to the agent, as shown:

7. The WebSpeed agent executes the Web object. It also performs the necessary reads from and writes to a data source. Notice that the WebSpeed agent maintains its connection with the WebSpeed Messenger while the Web object is executing, as shown:
8. The WebSpeed agent passes the generated HTML Web Page to the Messenger, which sends it to the browser through the Web Server, as shown:

Assuming that this is a stateless transaction, the agent disconnects from the Messenger, and it updates its availability with the broker. No context for the transaction is maintained and the agent is free to service another request. (However, you can preserve context by using cookies, URL query strings, hidden HTML form fields, or connection identifiers.)

If the transaction is state-persistent, the agent remains locked to the Web browser client until a specified time-out period passes. However, in actual practice state-persistent WebSpeed applications are rare. Locked agents slow down performance.

For more information about managing application states and context, see *OpenEdge Application Server: Developing WebSpeed Applications*. 
Index

A

ABL

39, 62, 76, 89–90
ABL-JMS API 90
client access to Web services 76
clients
39, 62
secure Internet AppServer access 62
separation of context 39
JMS messaging 89
ABL-JMS API
90
overview 90
AdminServer
33
definition 33
Agent
31, 104
AppServer agent, WebSpeed agent 31
Agent pool
28, 31
AppServer definition 28
WebSpeed definition 31
AIA
31
AppServer Internet Adapter 31
AppBuilder 104
Application partitioning
20, 38
AppServer
38
reasons to implement 38
definition 20
Application servers
72
Web services 72
Application services
19, 45–46, 51–52
AppServer
45–46, 51–52
assigning names 51
connecting 45
default service 52
definition 51
disconnecting 46
load balancing support 51
OpenEdge 19
AppObject
56
definition 56
AppServer
27–29, 31, 38–39, 41, 43, 45–46, 49, 51–52,
60, 62–63, 98
adapters
28, 31
definition 28
overview 31
agent
28
definition 28
application services 51
architecture 27
broker
28
definition 28
clients
28, 60
communicating through firewalls 60
definition 28
configurations 46
connecting 45
connection process 49
connection-based model 41
definition 28
disconnecting 46
further information 29
installation
28
definition 28
instance
28
definition 28
intelligent data interface 38
Internet secure access 62
logical connections
43, 45, 49
establishing 49
multiple instances 45
maintenance benefits 38
Open Client access 38
operating modes
41, 52
connection process 52
session-managed 41
physical connections
41, 49
establishing 49
processing boundary 39
purpose 38
secure Internet access 63
security benefits 38
AppServer (continued)

   session
       39
       definition 39
   session-managed
       41
       definition 41
   single instance
       45
       logical connections 45
   Sonic ESB and 98

AppServer agent
       28
       definition 28

AppServer broker
       28
       definition 28

AppServer Internet Adapter
       31, 60–62, 65
       architecture 60
       definition 31
       HTTP and 61
       HTTPS and 62
       Java servlet 60
       proxy servers 65

AppServers
       51
       fault-tolerant 51

Architectures
       18, 25, 27, 31, 60, 63, 69, 76, 79, 81, 86, 90, 97
       ABL access to Web services 76
       AIA 63
       AppServer 27
       AppServer Internet Adapter 60
       CORBA 69
       OpenEdge Adapter for Sonic ESB 97
       OpenEdge Adapter for SonicMQ 90
       OpenEdge Application Server 25
       OpenEdge Web services 79
       OpenEdge Web services tools 86
       services-oriented 18
       Web services
           69
           standard 69
           Web Services Adapter 81
           WebSpeed 31

Asynchronous requests
       38
       definition 38

Benefits (continued)

   Web services 76
   Bindings
       45
       Logical connections 45
       broker 103
   Broker
       31, 103
       AppServer broker, WebSpeed broker 31
   BrokerConnect 92
   Business logic
       38
       definition 38

C

   Client interface
       56, 70
       Open Client 56
       Web services 70
   Client session
       39
       definition 39
   ClientConnect 90
   Clients
       18, 30–31
       ABL 30
       HTML 31
       services interaction 18
       Web service 30
       WebSpeed 31

COM 69

   Command-line utilities
       34, 105
       OpenEdge Management or OpenEdge Explorer
           defined 34

   Common Object Model 69
   Common Object Request Broker Architecture 69

   Configurations
       46–48
       AppServer 46
       AppServer to AppServer connections 48
       client to many AppServer connections 47
       many client to one AppServer connections 46

   CONNECT( ) method
       45
       AppServer 45
   Connection and AppServer 45

   Connection IDs
       43
       definition 43
   Connection pool
       45
       definition 45
   Connection process
       49, 51–52
Connection process (continued)
   AppServer application services 51
   AppServer overview 49
      operating modes compared 52
      operation 52
   Connection-based model
      41–42, 44
      logical connection components 44
      overview 41
      physical connection components 42
   Container
      98
      Sonic ESB 98
      SonicMQ 98
   CORBA 69

D
   Database
      102
      support 102
   DCOM 69
   Default service
      52
      AppServer
      52
         definition 52
default.props file 81
DeployedServices.ds file 81
Deployers of Web services 75
Developer Studio 105
Developers of Web services 75
DISCONNECT( ) method
   46
   AppServer 46
   Disconnecting an AppServer 46
   Distributed application computing
      20–21
      definition 20
      traditional models compared 21
   Distributed application computing models
      22
      OpenEdge 22
   Distributed Common Object Model 69
   Distributed sessions
      39
      processing boundary
      39
      ABL client 39
      Open Client 39

E
  EDI 69
   Electronic Data Interchange 69

F
   Fault-tolerant AppServers 51
   Files
      34, 81
      default.props 81
      DeployedServices.ds 81
      FriendlyName.props 81
      FriendlyName.wsad 81
      FriendlyName.wsdl 81
      ubroker.properties
         34, 81
      WSA configuration 81
      WSAD 81
   Formats
      72
      SOAP 72
      FriendlyName.props file 81
      FriendlyName.wsad file 81
      FriendlyName.wsdl file 81

H
   HTML client
      31
      definition 31
   HTTP
      60–61, 65, 72
      AIA usage 60–61
      proxy server 65
      tunneling 60
      Web services 72
   HTTPS
      60, 62, 72
      AIA usage 60, 62
      tunneling
      62
      secure Internet AppServer connections 62
      Web services 72

I
   Integration services
      18–19
      definition 18
      OpenEdge 19

J
   Java Message Service (JMS)
      89
   SonicMQ 89
   Java Open Client 57
   Java servlet engine (JSE)
      60, 72, 81, 85
Java servlet engine (JSE) (continued)
  AIA 60
  as application server 72
  URL for OpenEdge Web services 85
  Web applications in URL 85
  WSA and 81
JavaScript 104
JMS 89
  Java Message Service (JMS) 89

L
Logical connections 43–45, 49
  AppServer 45, 49
    establishing 49
    multiple instances 45
    single instance 45
  AppServer components 44
    definition 43
  Web service clients 45
Logical three-tier model 21, 23
  definition 21
  OpenEdge 23

M
Messenger 31, 103
  WebSpeed Messenger 31

N
N-tier model 21
  definition 21
N-tier models 21
  compared to traditional 21
  logical three-tier 21
  physical n-tier 21
NameServer 28, 31, 42, 44, 92, 103
  AppServer 42, 44
    logical connections 44
    physical connections 42
  AppServer architecture 28
  SonicMQ BrokerConnect connections 92
  WebSpeed definition 31
Native Invocation methodology 94
  .NET Open Client 57
NSCONFIG 105
NSMAN 105

O
Open Client object model 56
  object types 56
  overview 56
Open Client Toolkit 57, 77, 96
  overview 57
  Sonic ESB OpenEdge services 96
  Web services and 77
Open Clients 38–39, 56–57, 62
  AppServer access 38
    interface 56
    definition 56
    format 56
  secure Internet AppServer access 62
  separation of context 39
  tools overview 57
OpenEdge 19, 22–23, 27, 60, 76, 80, 90, 97, 105
  AIA architecture 60
  AppServer 27
    architecture 27
  distributed application computing models 22
  logical three-tier model 23
  OpenEdge Adapter for Sonic ESB architecture 97
  OpenEdge Adapter for SonicMQ architecture 90
  physical n-tier model 23
  services in 19
  Web services tools 76, 80
    building and managing Web services 80
    WSDL Analyzer 76
OpenEdge ABL 30
  clients 30
OpenEdge Adapter for Sonic ESB 31, 93–98
  annotations 94
  architecture and sample scenario 97
  definition 31
  ESBOE files 95
  Native Invocation files 95
  Native Invocation methodology 94
  OpenEdge services 98
  overview 93
  Web Service Invocation methodology 96
OpenEdge Adapter for SonicMQ 89–92
Physical n-tier model (continued)
OpenEdge 23
Point-to-Point messaging 90
PROADSV 105
PROADSV utility 33
Procedures 27
remote 27
ProcObject 56
definition 56
Progress Developer Studio for OpenEdge 105
Proxy servers 65
AIA 65
ProxyGen 57, 81, 96
Sonic ESB OpenEdge services 96
Web services and 81
PTP messaging 90
Pub/Sub messaging 90
Publish and Subscribe messaging 90

R
Remote Method Invocation 69
Remote persistent procedures 39
definition 39
Remote procedures 27, 39
definition 27
how to execute 39
Remote procedures and functions 27
definition 27
Requests 27
RMI 69

S
Secure Socket Layer (SSL) 60
Security 86
OpenEdge Web services 86
Separate sessions 39
processing boundary 39
ABL client 39
Open Client 39
Separation of context 39
ABL clients and AppServer 39
Open clients and AppServer 39
ServerConnect 91
Unified Broker framework
administration framework definition 33
URLs for OpenEdge Web services 85
User-defined functions
 27
  remote 27

W
Web servers
 85
  URL for OpenEdge Web services 85
Web Service Invocation methodology 96
Web service Open Client 57
Web services
  30, 68–73, 75–77, 79–80, 84
  ABL access 76
  application servers 72
  architecture 79
  OpenEdge 79
  benefits 76
  client 75, 84
  developers 75
  interface generators 84
  client definition 68
  client overview 70
  clients 30
  definition 30
deployers 75
developers 75
eyear standards attempts 69
HTTP/S 72
OpenEdge support for building 77
run-time process 71
sample scenario 73
SOAP 72
standard architecture 69
technical roles 75
telephone system comparison 68
tools in OpenEdge 76, 80
building in ABL 80
WSDL Analyzer 76
when to use 75
Web Services Adapter (WSA)
  31, 77, 81, 84–85
  architecture 81
definition 31, 77
  JSE Web application 81
  run-time operation 84
Web Services Adapter (WSA) (continued)
URL for OpenEdge Web services 85
Web Services Description Language (WSDL) 73, 76, 81, 96
ABL access to Web services 76
files 81
Sonic ESB OpenEdge services and 96
WebSpeed 31, 33, 101–106
advantages 102
agent 104
agent architecture 31
AppBuilder 104
application state 106
architecture 31
broker
    31
definition 31
clients 31
command-line utilities 105
components 102, 106
configuration and management 105
database support 102
further information 33
handling requests 106
introduction to 101
JavaScript 104
Messenger
    31, 103
definition 31
NameServer 103
NSCONFIG 105
NSMAN 105
OpenEdge Management or OpenEdge Explorer 105
PROADSV 105
sample applications 101
SpeedScript 104
SpeedStart 106
Transaction server 103
Transaction Server
    31
definition 31
WebTools 104
WebSpeed (continued)
Workshop 104
WSCONFIG 105
WTSBMAN 105
WebSpeed agent
    31
definition 31
WebSpeed broker
    31
definition 31
WebSpeed Messenger
    31
definition 31
installation
    31
definition 31
instance definition 31
WebSpeed Transaction Server
    31
definition 31
installation definition 31
instance
    31
definition 31
WebTools 104
Workshop 104
WSA 31
Web Services Adapter (WSA) 31
WSAD files 81
WSCONFIG 105
WSDL 73
Web Services Description Language (WSDL) 73
WSDL Analyzer 76
WTSBMAN 105
X
XML 68
Web services and 68