OpenEdge® Application Server: Developing AppServer Applications
Notices

© 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 11.4.0

OpenEdge Application Server: Developing AppServer Applications
# Table of Contents

## Preface

- Purpose ........................................................................................................... 13
- Audience ......................................................................................................... 14
- Organization .................................................................................................. 14
- Using this manual .......................................................................................... 14
  - References to ABL compiler and run-time features .................................... 15
  - References to ABL data types ..................................................................... 15
- Typographical conventions .............................................................................. 15
- Examples of syntax descriptions ..................................................................... 17
  - Long syntax descriptions split across lines ............................................... 18
  - Complex syntax descriptions with both required and optional elements .... 19
- Example procedures ........................................................................................ 19
- OpenEdge messages ....................................................................................... 20
  - Obtaining more information about OpenEdge messages ......................... 21

## Chapter 1: AppServer and Client Interaction ............................................. 23

- Understanding AppServer operating modes .................................................... 23
- State-reset operating mode ............................................................................. 24
  - General AppServer-client interaction ......................................................... 24
  - Connection management ............................................................................ 25
  - Performance considerations ...................................................................... 25
  - Context management .................................................................................. 25
- State-aware operating mode ........................................................................... 26
  - General AppServer-client interaction ......................................................... 26
  - Connection management ............................................................................ 26
  - Performance considerations ...................................................................... 27
  - Context management .................................................................................. 27
- Stateless operating mode .................................................................................. 27
  - General AppServer-client interaction ......................................................... 28
  - Connection management ............................................................................ 28
  - Performance considerations ...................................................................... 29
  - Context management .................................................................................. 29
- State-free operating mode ............................................................................... 30
  - General AppServer-client interaction ......................................................... 30
  - Connection management ............................................................................ 31
  - Performance considerations ...................................................................... 32
  - Context management .................................................................................. 33
- Session models and application services .......................................................... 33
Chapter 3: Programming ABL Client Applications

Programming for the application session model.............................................................................. 88
ABL for programming AppServer client procedures .................................................................................. 88
Accessing AppServer resources from a client .......................................................................................... 94
Steps for accessing AppServer resources ............................................................................................... 94
Features of client-AppServer interactions.............................................................................................. 95
Connecting an application service or AppServer instance..................................................................... 96
Creating a server object handle .............................................................................................................. 96
Establishing a connection with the CONNECT( ) method ...................................................................... 96
Connection parameters argument ......................................................................................................... 97
Session-managed application arguments ............................................................................................... 102
Connection examples ............................................................................................................................. 103
Connecting to a session-managed AppServer ....................................................................................... 103
Connecting to a session-free AppServer ............................................................................................... 104
Accessing the connection ID on a session-managed client ...................................................................... 104
Accessing client context regardless of session model ............................................................................ 105
Managing context from the ABL client with OERequestInfo objects .................................................. 106
Running and managing remote procedures ............................................................................................ 107
Running remote procedures .................................................................................................................. 107
Synchronous remote procedures .......................................................................................................... 107
Asynchronous remote procedures ......................................................................................................... 107
RUN statement options for remote procedures ...................................................................................... 107
Passing class-based objects as parameters between an AppServer and an ABL client .......................... 112
Remote procedure behavior .................................................................................................................. 113
Understanding proxy procedure handles .............................................................................................. 113
Accessing proxy procedure handles ..................................................................................................... 113
Comparing proxy and local procedure handles ...................................................................................... 114
Deleting remote persistent procedures ................................................................................................. 114
Remote procedure code examples ........................................................................................................ 115
Example 1: Remote non-persistent procedure example ........................................................................ 115
Example 2: Remote persistent procedure example ................................................................................ 116
Example 3: Remote persistent procedure example using the FUNCTION statement .......................... 117
Example 4: Remote non-persistent procedure with a class-based object as a parameter...................... 118
Disconnecting from an AppServer instance ......................................................................................... 118
Using the DISCONNECT( ) method ..................................................................................................... 119
Disconnecting from a session-managed AppServer ................................................................................. 119
Disconnecting from a session-free application service ........................................................................... 120
Deleting the server handle ..................................................................................................................... 120
Handling conditions and return values ................................................................................................. 120
Raising the STOP condition .................................................................................................................. 120
Chapter 4: Design and Implementation Considerations

Primary performance considerations

Minimizing network traffic

Off loading resource-intensive processing tasks

Choosing an AppServer operating mode

Application characteristics to consider

Managing larger amounts of context

Managing smaller amounts of context

Limiting bound connections

Considering ample computer resources

Considering the future

Setting the number of AppServer agents

Choosing a transaction model

Using load balancing

Secondary performance considerations

Connect and disconnect performance issues

Using the AppServer Connect procedure

Performance-related issue for the CONNECT( ) method

OpenEdge Application Server: Developing AppServer Applications
Preface

For details, see the following topics:

- Purpose
- Audience
- Organization
- Using this manual
- Typographical conventions
- Examples of syntax descriptions
- Example procedures
- OpenEdge messages

Purpose

This manual provides comprehensive information on a range of topics associated with developing application services with the AppServer™ and the applications that use them. These topics include: how an AppServer and client interact, specific ABL (Advanced Business Language) programming information to build distributed applications using the AppServer, sample code used in AppServer applications, a discussion of issues related to designing and implementing distributed applications, and debugging AppServer applications. These topics also include information that pertains to using different types of clients with the AppServer, including ABL clients, Open Clients, and Web service clients when the AppServer is used to implement an OpenEdge Web service. Some topics document AppServer features also available for use with the OpenEdge™ Adapter for SonicMQ BrokerConnect (SonicMQ BrokerConnect) as noted. For more information on the OpenEdge Adapter for SonicMQ, see OpenEdge Development: Messaging and ESB.
Audience

This book is designed as a guide for anyone wanting to understand and use the AppServer to build distributed applications. However, application programmers will find the information in this guide most helpful to understand how to program and maintain ABL for an AppServer application.

Organization

AppServer and Client Interaction on page 23
Describes the basic operating modes and configuration options that you can use to define an AppServer and the interactions that a client can have with it.

Programming the AppServer on page 49
Describes how to program the AppServer, including both application procedures and the types of procedures that you can define to set up and manage the AppServer session context for your application procedures. It also describes how to manage context, including transactions, for the different AppServer operating modes and handle error information that you pass to client applications. Code examples illustrate specific programming points.

Programming ABL Client Applications on page 87
Describes how to program a client application, with emphasis on ABL clients. It explains how to connect to and disconnect from an AppServer, run and manage remote procedures, and handle error information returned from the AppServer. Code examples illustrate specific programming points.

Design and Implementation Considerations on page 137
Identifies and describes various issues related to the design and implementation of the AppServer. This chapter highlights specific performance, deployment, and security issues, and introduces some programming techniques to help you write AppServer procedures.

Debugging on page 179
Describes two enhancements to the Application Debugger that support debugging distributed applications and explains how to use the AppServer log file to troubleshoot your application.

Connecting to AppServers Using a URL on page 187
Describes how to use a URL to connect to an AppServer (or SonicMQ BrokerConnect). For an AppServer, this applies to ABL clients and Open Clients. For the SonicMQ BrokerConnect, this applies only to ABL clients.

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><strong>Italic</strong></td>
<td>Italic typeface indicates the title of a document, or signifies new terms.</td>
</tr>
<tr>
<td><strong>SMALL, BOLD CAPITAL LETTERS</strong></td>
<td>Small, bold capital letters indicate OpenEdge key functions and generic keyboard keys; for example, GET and CTRL.</td>
</tr>
<tr>
<td><strong>KEY1+KEY2</strong></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><strong>KEY1 KEY2</strong></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:**

<table>
<thead>
<tr>
<th>Fixed width</th>
<th>A fixed-width font is used in syntax, code examples, system output, and file names.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fixed-width italics</strong></td>
<td>Fixed-width italics indicate variables in syntax.</td>
</tr>
<tr>
<td><strong>Fixed-width bold</strong></td>
<td>Fixed-width bold italic indicates variables in syntax with special emphasis.</td>
</tr>
<tr>
<td><strong>UPPERCASE fixed width</strong></td>
<td>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.</td>
</tr>
<tr>
<td><strong>Period (.) or colon (:)</strong></td>
<td>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.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Large brackets indicate the items within them are optional.</td>
</tr>
<tr>
<td>[]</td>
<td>Small brackets are part of ABL.</td>
</tr>
<tr>
<td>{ }</td>
<td>Large braces indicate the items within them are required. They are used to simplify complex syntax diagrams.</td>
</tr>
<tr>
<td>{}</td>
<td>Small braces are part of ABL. For example, a called external procedure must use braces when referencing arguments passed by a calling procedure.</td>
</tr>
</tbody>
</table>
### Examples of syntax descriptions

In this example, `ACCUM` is a keyword, and `aggregate` and `expression` are variables:

**Syntax**

```
ACCUM aggregate expression
```

`FOR` is one of the statements that can end with either a period or a colon, as in this example:

```
FOR EACH Customer NO-LOCK:
   DISPLAY Customer.Name.
END.
```

In this example, `STREAM stream`, `UNLESS-HIDDEN`, and `NO-ERROR` are optional:

**Syntax**

```
DISPLAY [ STREAM stream ] [ UNLESS-HIDDEN ] [ NO-ERROR ]
```

In this example, the outer (small) brackets are part of the language, and the inner (large) brackets denote an optional item:

**Syntax**

```
INITIAL [ constant [ , constant ] ]
```

A called external procedure must use braces when referencing compile-time arguments passed by a calling procedure, as shown in this example:

**Syntax**

```
{ &argument-name }
```
In this example, EACH, FIRST, and LAST are optional, but you can choose only one of them:

**Syntax**

```plaintext
PRESELECT [ EACH | FIRST | LAST ] record-phrase
```

In this example, you must include two expressions, and optionally you can include more. Multiple expressions are separated by commas:

**Syntax**

```plaintext
MAXIMUM ( expression, expression [, expression ] ... )
```

In this example, you must specify MESSAGE and at least one expression or SKIP ( ( n ) ], and any number of additional expression or SKIP ( ( n ) ] is allowed:

**Syntax**

```plaintext
MESSAGE { expression | SKIP ( ( n ) ] } ...
```

In this example, you must specify {include-file, then optionally any number of argument or &argument-name = "argument-value", and then terminate with }:

**Syntax**

```plaintext
{ include-file
  [ argument | &argument-name = "argument-value" ] ... }
```

### Long syntax descriptions split across lines

Some syntax descriptions are too long to fit on one line. When syntax descriptions are split across multiple lines, groups of optional and groups of required items are kept together in the required order.

In this example, WITH is followed by six optional items:

**Syntax**

```plaintext
WITH [ ACCUM max-length ] [ expression DOWN ]
  [ CENTERED ] [ n COLUMNS ] [ SIDE-LABELS ]
  [ STREAM-IO ]
```
Complex syntax descriptions with both required and optional elements

Some syntax descriptions are too complex to distinguish required and optional elements by bracketing only the optional elements. For such syntax, the descriptions include both braces (for required elements) and brackets (for optional elements).

In this example, `ASSIGN` requires either one or more `field` entries or one `record`. Options available with `field` or `record` are grouped with braces and brackets:

**Syntax**

```
| { record [ EXCEPT field . . . ] } }
```

Example procedures

This manual may provide example code that illustrates syntax and concepts. You can access many of the example files, and details for installing them, from the following locations:

- A self-extracting Documentation and Samples file available on the OpenEdge download page of the Progress Software Download Center
- The OpenEdge Product Documentation Overview page on Progress Communities:


Once installed, you can locate the example files for this manual in the following path under the OpenEdge Documentation and Samples installation directory:

<table>
<thead>
<tr>
<th>This directory . . .</th>
<th>Contains examples for the following documents . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>src/prodoc/dotnetobjects</td>
<td>OpenEdge Development: GUI for .NET Programming</td>
</tr>
<tr>
<td>src/prodoc/dynamics</td>
<td>The Progress Dynamics documentation</td>
</tr>
<tr>
<td>src/prodoc/getstartoop</td>
<td>OpenEdge Development: Object-oriented Programming</td>
</tr>
<tr>
<td>src/prodoc/handbook</td>
<td>OpenEdge Getting Started: ABL Essentials</td>
</tr>
<tr>
<td>src/prodoc/interfaces</td>
<td>OpenEdge Development: Programming Interfaces</td>
</tr>
<tr>
<td>src/prodoc/json</td>
<td>OpenEdge Development: Working with JSON</td>
</tr>
</tbody>
</table>
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**.
AppServer and Client Interaction

This chapter describes how the AppServer™ supports interaction with a client to provide general purpose application services and how a client can interact with an AppServer to access these application services.

For details, see the following topics:

- Understanding AppServer operating modes
- Session models and application services
- Understanding synchronous and asynchronous requests

Understanding AppServer operating modes

The basis of AppServer support for client interaction is the AppServer operating mode, which determines how client requests can be processed by the AppServer. The AppServer supports any one of four possible operating modes. Depending on the operating mode, the interaction between the AppServer and client conforms to one of two possible session models:

- **Session-managed** — Where a single AppServer handles all requests from a single client to a single application service, and the client manages a physical connection to the AppServer
- **Session-free** — Where one or more AppServers can handle all requests from a single client to a single application service, and the client manages no physical connection to any single AppServer

Regardless of the operating mode, a client can invoke synchronous or asynchronous requests to the AppServer, which the client manages depending on the session model.
When you configure an AppServer, you specify an operating mode that indicates how the AppServer broker manages its AppServer agent pool and assigns client requests to AppServer agents within the pool. The choice of operating mode has a potentially major affect on the performance and design of your AppServer application, including both the procedures that run on the AppServer and your client application.

Any ABL procedure running on the AppServer can determine the operating mode it is running on by reading the value of the SERVER-OPERATING-MODE attribute on the SESSION system handle. This is a CHARACTER parameter that returns the name of the operating mode as a string:

- "STATE-RESET"
- "STATE-AWARE"
- "STATELESS"
- "STATE-FREE"

The sections that follow describe these operating modes in detail.

For information on the AppServer architecture and an overview of AppServer operating modes, see *OpenEdge Getting Started: Application and Integration Services*. For information on how to configure the operating mode for an AppServer instance, see *OpenEdge Application Server: Administration*.

### State-reset operating mode

When OpenEdge sends a client application's connection request to an AppServer that is running in state-reset operating mode, the AppServer broker connects the client application directly to one of the agents in its AppServer agent pool. The connected AppServer agent remains dedicated to the client application for the life of the connection.

Between connections, OpenEdge reinitializes the AppServer session to its initial state. This is the state it is in immediately after the AppServer agent is started and after all startup parameters are processed.

**Note:** State-reset operating mode is conceptually similar to the operation of AppServers prior to Progress Version 9.

### General AppServer-client interaction

The general interaction between a state-reset AppServer and client proceeds as follows:

1. The client connects to the AppServer broker with or without the help of a NameServer, which provides the address of the broker from a list of AppServers that support a specified application service.

   **Note:** The algorithm the NameServer uses to choose the broker depends on whether the NameServer is configured for load balancing. For more information, see *Load balancing and session models* on page 39.

2. The AppServer broker connects the client to an available AppServer agent, which is initialized to its initial startup state, with any database connections and context removed that were initialized by any client previously connected to this agent. The broker is then available to accept a connection request from another client.
3. The client invokes an application service request (synchronously or asynchronously).

4. The connected agent executes the request and returns the results to the client.

5. The client handles the results of the application service request and either invokes another request as in Step 3 or disconnects from the AppServer as in Step 6.

6. When the client no longer requires the application service, it disconnects from the AppServer agent and cleans up after its connection to the application service, making the AppServer agent available to handle another client connection.

The number of available simultaneous client connections to a state-reset AppServer corresponds to the number of AppServer agents that it can start up and manage at one time.

**Connection management**

When the AppServer receives a connection request, the AppServer broker chooses an AppServer agent that is not currently dedicated to some other client application's connection. If all AppServer agents are unavailable, the AppServer broker might create another AppServer agent for the connection, or it might return an error. The result depends on how many AppServer agents are running and the values of various configuration settings.

When the client application disconnects from its AppServer agent, the AppServer agent notifies the AppServer broker that it is now available to service another client application connection.

**Note:** In AppServers prior to Progress Version 9, the AppServer broker terminates the disconnected AppServer agent.

To help an AppServer session perform common connection tasks, such as user authentication, you can configure a Connect procedure that runs for each connection request and a Disconnect procedure that runs for each disconnection request. For more information on these procedures, see Programming the AppServer on page 49.

**Performance considerations**

Because an AppServer running in state-reset operating mode dedicates an AppServer agent to each client connection, the connected client is guaranteed of having each request handled immediately by the AppServer. Assuming that the general process load on the AppServer platform is light enough, this means that the client can expect relatively fast response time from the AppServer.

However, throughput in a state-reset connection tends to drop as client load increases, because the number of dedicated AppServer agents available to handle client connections is necessarily limited by the number of AppServer agents that can be started on the AppServer machine.

**Context management**

Because each AppServer agent is dedicated to a client connection, the AppServer session maintains all session context between client requests. Thus, for example, any global values or database connections that you set in remote procedures remain available for the life of the connection or until you change them during some remote procedure call.

In between connections, an AppServer session running in state-reset operating mode resets its context to the initial state. This initial state is its context after the AppServer agent first starts and after all AppServer agent startup parameters take effect, including databases connections.
State-aware operating mode

When OpenEdge sends a client application's connection request to an AppServer that is running in state-aware operating mode, the AppServer broker connects the client application directly to one of the agents in its AppServer agent pool. The connected AppServer agent remains dedicated to the client application for the life of the connection.

Unlike state-reset, OpenEdge does not reset the AppServer session context between connections. Whatever context remains when one connection finishes is available to the next connection.

General AppServer-client interaction

The general interaction between a state-aware AppServer and client proceeds as follows:

1. The client connects to the AppServer broker with or without the help of a NameServer, which provides the address of the broker from a list of AppServers that support a specified application service.

   Note: The algorithm the NameServer uses to choose the broker depends on whether the NameServer is configured for load balancing. For more information, see Load balancing and session models on page 39.

2. The AppServer broker connects the client to an available AppServer agent, which makes the session context available from any client previously connected to this agent. The broker is then available to accept a connection request from another client.

3. The client invokes an application service request (synchronously or asynchronously).

4. The connected agent executes the request and returns the results to the client.

5. The client handles the results of the application service request and either invokes another request as in Step 3 or disconnects from the AppServer as in Step 6.

6. When the client no longer requires the application service, it disconnects from the AppServer agent and cleans up after its connection to the application service, making the AppServer agent available to handle another client connection.

The number of available simultaneous client connections to a state-aware AppServer corresponds to the number of AppServer agents that it is configured to start up and manage at one time.

Connection management

When the AppServer receives a connection request, the AppServer broker chooses an AppServer agent that is not currently dedicated to some other client application's connection. If all AppServer agents are unavailable, the AppServer broker might create another AppServer agent for the connection, or it might return an error. The result depends on how many AppServer agents are running and the values of various configuration settings.

When the client application disconnects from its AppServer agent, the AppServer agent notifies the AppServer broker that it is now available to service another client application connection.

To help an AppServer session perform common connection tasks, such as user authentication, you can configure a Connect procedure that runs for each connection request and a Disconnect procedure that runs for each disconnection request. For more information on these procedures, see Programming the AppServer on page 49.
Performance considerations

Because an AppServer running in state-aware operating mode dedicates an AppServer agent to each client connection, the connected client is guaranteed of having each request handled immediately by the AppServer. Assuming that the general process load on the AppServer platform is light enough, this means that the client can expect relatively fast response time from the AppServer.

However, throughput in a state-aware connection tends to drop as client load increases, because the number of dedicated AppServer agents available to handle client connections is necessarily limited by the number of AppServer agents that can be started on the AppServer machine.

Context management

Unlike state-reset, an AppServer running in state-aware operating mode does not reset its session context between client connections. Instead, OpenEdge maintains any context created in the AppServer session until you specifically delete it in some procedure that you write.

For an AppServer running in state-aware operating mode you can also configure Startup and Shutdown procedures that run in every AppServer session when it starts up and shuts down. You can use these procedures to establish and discard run-time context that the AppServer maintains in common for all connections. For more information on the Startup and Shutdown procedures, see Programming the AppServer on page 49.

Like state-reset, because each AppServer agent is dedicated to a client connection, the AppServer maintains all session context between client requests. Thus, for example, any global values or database connections that you set in remote procedures remain available for the life of the connection or until you change them during some remote procedure call.

As noted in a previous section, an AppServer session running in state-aware operating mode maintains most of its context between client connections. However, it does delete any remote persistent procedures that are still active when the connection terminates.

Stateless operating mode

Unlike state-reset or state-aware, an AppServer running in stateless operating mode does not dedicate any AppServer agent to a client connection. Instead, all AppServer agents remain generally available to execute remote procedure and user-defined function requests from all connected client applications.

When a client connects to an AppServer that is running in stateless operating mode, the AppServer broker maintains the connection between itself and the client. The AppServer broker can then dispatch each remote procedure request from the client to any available AppServer agent to execute. Because any AppServer agent that executed a previous request from this client might not be available, the AppServer agent that handles subsequent requests might be different.

Note: The name of this operating mode ("stateless") refers to the lack of context that AppServer sessions automatically maintain between client requests. However, stateless operating mode does support additional mechanisms that you can use to maintain some context between requests, as described in the following sections.
General AppServer-client interaction

The general interaction between a stateless AppServer and client proceeds as follows:

1. The client connects to the AppServer broker with or without the help of a NameServer, which provides the address of the broker from a list of AppServers that support a specified application service. The broker is then available to handle application service requests from this client or to accept a connection request from another client.

   **Note:** The algorithm the NameServer uses to choose the broker depends on whether the NameServer is configured for load balancing. For more information, see Load balancing and session models on page 39.

2. The client invokes an application service request (synchronously or asynchronously).

3. The connected AppServer broker passes the client request to an available AppServer agent.

4. The agent executes the request and returns the results to the broker, which returns the results to the client. The AppServer agent is immediately available to handle an application service request from any client connected to the broker.

5. The client handles the results of the application service request and either invokes another request as in Step 2 or disconnects from the AppServer as in Step 6.

6. When the client no longer requires the application service, it disconnects from the AppServer broker and cleans up after its connection to the application service.

The number of available simultaneous client connections to a stateless AppServer is limited only by the required performance for each client connection. The performance of client connections depends on the availability of an AppServer agent to execute a client request. Agent availability depends on:

- The resources of the AppServer machine
- The number of agents the AppServer is configured to startup and manage at one time
- The number and types of client requests to the stateless AppServer

Connection management

By default, all connections on a stateless AppServer are *unbound*—that is, all AppServer agents are available to handle requests from any client connection. As long as an AppServer agent is not currently handling a client request, it can handle requests from any connected client. However, at any time during the handling of a client request, a client connection can become *bound*—that is, the AppServer agent handling the request becomes available to handle requests only from the client it is currently servicing. While a connection is bound, all requests from the connected client are handled by the same AppServer agent. In the same way it became bound, at any time during the handling of a client request, a bound connection can become unbound, at which point requests from the connected client can be handled by any available AppServer agent.

In general, then, a stateless AppServer agent is available to handle a client request as long as it is not currently processing a client request and as long as it is not waiting on a bound connection. If a stateless AppServer has no AppServer agents available to handle requests, all requests are queued at the AppServer broker until an AppServer agent becomes available.
To help an AppServer session perform common connection tasks, such as user authentication, you can configure a Connect procedure that runs for each connection request and a Disconnect procedure that runs for each disconnection request. For more information on these procedures, see Programming the AppServer on page 49.

A connection in the unbound state can transition to the bound state in two ways:

- A client application invokes **RUN PERSISTENT** to instantiate a remote persistent procedure in the context of the connection. The AppServer agent that handles the remote persistent procedure request thus becomes bound to the connection.

- The AppServer agent runs a procedure that sets the **SERVER-CONNECTION-BOUND-REQUEST** attribute on the **SESSION** handle to **TRUE**. The AppServer agent that sets this attribute thus becomes bound in the connection.

Note that it is the successful instantiation (by means of **RUN PERSISTENT**) of a remote persistent procedure that forces the connection to transition from the unbound state to the bound state. If a client fails in its attempt to instantiate a remote persistent procedure on an unbound connection, the connection remains unbound.

For more information on these procedures, see Programming the AppServer on page 49.

**Performance considerations**

Because a stateless AppServer passes requests from connected clients to any available AppServer agent, the number of clients that a single AppServer instance can handle using this operating mode is relatively high. That is, a relatively small number of AppServer agents can support many client connections. If you never set connections to the bound state, a stateless AppServer can make maximum use of its available AppServer agent resources to handle client requests. This operating mode thus maximizes throughput for large numbers of clients, especially when they maintain short connections.

**Context management**

An AppServer running in stateless operating mode does not reset its session context between client connections. Instead, it maintains almost all context created in an AppServer session until a particular AppServer agent terminates or the context is changed by some procedure running in the AppServer session.

For an AppServer running in stateless operating mode you can also configure Startup and Shutdown procedures that run in every AppServer session when it starts up and shuts down. You can use these procedures to establish and discard run-time context that the AppServer maintains in common for all connections. For more information on the Startup and Shutdown procedures, see Programming the AppServer on page 49.

Using a bound connection, the AppServer maintains all session context created between client requests (similar to a state-aware AppServer), because the bound connection maintains one AppServer agent to execute all requests for the same connected client until you reset the connection to the unbound state.

Using an unbound connection, the AppServer does not automatically guarantee that context created during a remote procedure request will be available when the same client makes the next request. As noted in a previous section, this is because a stateless AppServer cannot guarantee that the same AppServer agent will handle the next request. However, you can still maintain context created between requests from a connected client, but only by explicitly saving the created context using one of several mechanisms.

Two such mechanisms specially supported for stateless AppServers include:
• The SERVER-CONNECTION-CONTEXT attribute on the SESSION handle that allows you to pass CHARACTER data from one request to another on the same client connection

• Activate and Deactivate procedures that you can configure to execute before and after each remote procedure request

For more information on these mechanisms, see Programming the AppServer on page 49.

As noted in a previous section, OpenEdge does not automatically delete any AppServer session context between requests to a stateless AppServer. The only exception to this is if a client application disconnects from a stateless AppServer while it has remote persistent procedures active in the AppServer session. In that case, OpenEdge does automatically delete the active remote persistent procedures before disconnecting the client application.

State-free operating mode

Like stateless, an AppServer running in state-free operating mode does not dedicate any AppServer agent to a client connection. Also, like stateless operating mode, all AppServer agents remain generally available to execute remote procedure and user-defined function requests from all client applications. However, unlike stateless, an AppServer running in state-free mode is never physically connected to a single client (except for bound connections, as described in this section). Instead, clients are logically connected (bound) to the application service that the AppServer provides, and the AppServer broker serves, not as a physical connection point, but as a request listener.

When a client wants to send a remote procedure or user-defined function request for the specified application service, OpenEdge forwards the request to an AppServer that supports that application service. The AppServer broker, which is listening for requests from any client on the network, receives and dispatches the request for execution to any available AppServer agent in its agent pool. When the agent completes execution of the request, it returns the results to the broker, which passes them back to the client, and the agent becomes available to service another request that might be received and passed to it by the broker.

Note: The name of this operating mode ("state-free") refers to the lack of any context maintained by AppServer sessions between client requests. The state-free operating mode does support limited context management as described in the following sections. However, the benefits of the state-free operating mode can degrade significantly if you attempt any session context management in this mode.

General AppServer-client interaction

The general interaction between a state-free AppServer and client proceeds as follows:

1. The client sends a request to a NameServer to return a logical connection to a specified application service.

2. The NameServer returns to the client a list of AppServers that support the application service.

Note: The algorithm the NameServer uses to choose the list of AppServers depends on whether the NameServer is configured for load balancing. For more information, see the Load balancing and session models on page 39.
3. The client creates a certain number of physical connections for each AppServer in the list and maintains them in a **connection pool**, depending on pool connection parameter settings.

**Note:** Each physical connection in the pool is a separate TCP/IP connection to an AppServer broker in the list provided by the NameServer.

4. The client invokes an application service request (synchronously or asynchronously).

5. The client chooses an AppServer to handle the request based on the list returned by the NameServer and selects a physical connection that is available to send requests to the AppServer broker (a **free connection**). If there is no free connection to the AppServer broker, the client creates one (limited by the pool connection parameter settings).

6. The client sends the request to the AppServer broker over the **selected connection** (which is now unavailable for other client requests).

7. The connected AppServer broker passes the client request to an available AppServer agent. The broker continues to be available to handle other client requests.

8. The agent executes the request and returns the results to the broker, which forwards the results back to the client. The connection used by the client to the broker is freed and returned to the connection pool on the client. The AppServer agent is immediately available to handle a request from any client that sends one to the broker.

9. The client handles the results of the application service request and either invokes another request as in Step 4 or disconnects from the application service as in Step 10.

10. When the client no longer requires the application service, it disconnects from the application service and cleans up the connection pool and all resources associated with the logical connection.

The number of available simultaneous logical connections to a state-free application service is limited only by the performance required for each client that connects to the application service. The performance of each client connection depends on the availability of a free physical connection to an AppServer. The availability of free physical connections depends on:

- The number and availability of AppServers on the application service list
- The resources of each AppServer machine
- The number of agents each AppServer is configured to start up and manage at one time
- The number and types of client requests to the state-free application service

**Connection management**

By default, there are no physical connections between a state-free AppServer and a client—that is, all AppServer agents are available to handle requests from any client making a request for its application services. As long as a state-free AppServer has agents available (it has at least one agent that is not currently handling a client request), it can handle requests from any client on the network.

However, at any time during the handling of a client request, a client can become bound to the agent—that is, the AppServer agent handling the request becomes available to handle requests only from the client it is currently servicing. The function of this physical binding is very different from a bound connection in stateless operating mode (see the **Stateless operating mode** on page 27).
A client can bind to an agent in state-free operating mode only by invoking a **RUN PERSISTENT** request to instantiate a persistent procedure. When this happens, the AppServer broker that receives the request creates a physical connection between the client and an available agent where the persistent procedure is then instantiated.

For the client, this physical binding is valid only for requests to execute internal procedures or user-defined functions of the remote persistent procedure that it has instantiated on this agent. If the client makes any other requests for the specified application service, all other available AppServer instances or other available agents of a single AppServer can execute these requests. For the agent, this physical binding means that it is no longer available to handle requests from any other clients while it is bound to this single client.

**Caution:** Binding a single session-free AppServer agent to a single client, as described in this section, reduces the resources available to handle requests from all clients for the session-free application service. Therefore, be certain that your application requires this type of agent binding before using it for application development. As an alternative, consider using **RUN SINGLE-RUN** or **RUN SINGLETON**, both of which instantiate an external procedure without binding to the agent.

A bound agent becomes free (unbound) and is again available to handle requests from all clients when the bound client deletes the remote persistent procedure that it has created on the agent. Note that it is the successful instantiation (by means of **RUN PERSISTENT**) of a remote persistent procedure that forces an agent to become bound to the client. If a client fails in its attempt to instantiate a remote persistent procedure, no binding between the client and AppServer agent occurs.

In general, then, a state-free AppServer agent is available to handle a client request as long as it is not currently processing a client request and as long as it is not waiting on an exclusive bound connection to a single client. If a state-free AppServer has no AppServer agents available to handle requests, it returns a run-time error to the OpenEdge interface (client or Web service) that sent the request.

**Note:** In stateless operating mode, if the AppServer receives a request from a connected client and it has no available agents, it queues the request until an agent becomes available to execute it instead of returning an error, as does a state-free AppServer.

To help a state-free AppServer session perform common tasks for each request that it receives, you can configure Activate and De-activate procedures on the AppServer. For more information on connection and request management on a state-free AppServer, see Programming the AppServer on page 49.

**Performance considerations**

A state-free AppServer provides the highest degree of availability and responsiveness to clients compared to a comparable configuration using any other operating mode. Because multiple state-free AppServer instances and their agents can handle multiple requests from one or more clients, multiple requests even from the same client can execute in parallel, vastly increasing the potential response time for a given request. This capability provides the highest degree of scalability compared to other operating modes. That is, a relatively small number of AppServer agents can support many more client requests than even for stateless operating mode. As long as clients never use bound agents, a state-free AppServer can make maximum use of its available agent resources to handle client requests. This operating mode thus maximizes throughput for large numbers of clients, especially if no attempt is made to maintain any context between client requests.
Context management

An AppServer running in state-free operating mode can provide context management in the following ways:

- Using persistent storage managed by the application ("roll your own"), such as a database that is keyed to specific clients or common information used by all clients

- When a client binds an agent to itself by instantiating a remote persistent procedure (see Connection management on page 25).

- Using the client context identifier, a globally unique identifier used to manage client login session context. For more information on the client context identifier, see Managing state-free and stateless client context on page 69.

An agent does not reset any session context between client requests. However, there is no guarantee what client the agent will service next. So, allowing AppServer procedures when they are invoked to create persistent procedures or other objects that remain in the context of an agent after it services a client provides little benefit and is likely to create harmful memory leaks in the agent session. The exception to this is when a procedure is run as a single-run or singleton, which allows persistent procedures to be run in a state-free AppServer without binding. For more information on the single-run and singleton options, see Running and managing remote procedures on page 107.

For an AppServer running in state-free operating mode you can also configure Startup and Shutdown procedures that run in every AppServer session when it starts up and shuts down. You can use these procedures to establish and discard run-time context that the AppServer maintains in common across all client requests. For more information on the Startup and Shutdown procedures, see Programming the AppServer on page 49.

Session models and application services

As noted in previous sections, the most fundamental choice for setting up and programming an AppServer is the operating mode you want to use. The operating mode determines the session model that the AppServer and its application service must support. The session model, in turn, determines certain requirements for programming any AppServer client application. It also determines how the OpenEdge Web Services Adapter (WSA) or the OpenEdge Adapter for Sonic ESB® manages any OpenEdge Web services that an AppServer supports.

The session model represents a set of standard criteria that determine how an AppServer can interact with its clients as an application service or as a Web service, and it determines some fundamental features of the entire application. The AppServer supports two different session models, distinguished by the following criteria:

- **Session-managed** — The client sends its requests and receives the responses over a persistent connection to a single AppServer that supports the connected client. Requests from the same client can only be handled sequentially (single-threaded) by the AppServer both as an application service and as a Web service. The AppServer operating modes that support session-managed applications include:
  - State-reset
  - State-aware
  - Stateless
• **Session-free** — The client sends its requests and receives its responses from one or more AppServers that maintain no connections with their clients. All requests from any and all clients are managed in parallel (multi-threaded) whether the requests are for an application service or a Web service. For direct clients of session-free AppServers (application services), the request management function resides in the client session. For clients of OpenEdge Web services, the AppServer request management resides in the Web Services (or Sonic ESB) Adapter. The only AppServer operating mode that supports session-free applications is the state-free operating mode.

As described previously (see Understanding AppServer operating modes on page 23), you might have to program an application differently for it to work in a different AppServer operating mode. However, the differences between the two basic session models that AppServer operating modes support affect application design at such a fundamental level that OpenEdge provides explicit support for these session models in various phases of application development and administration. This is especially true for the handling of context management and asynchronous requests. So, for example, when defining a Web service or binding client sessions to an application service, OpenEdge requires you to specify the session model in order to manage the Web service or client binding properly.

### Session models and Open Client objects

The Open Client object model is the basis for defining an Open Client interface using the Open Client Proxy Generator. (For information on ProxyGen, see *OpenEdge Development: Open Client Introduction and Programming*). This object model includes the following types of objects, each of which is represented on the AppServer by specific ABL constructs:

- **AppObject** — A single collection of external procedures that can include both persistent and non-persistent procedures, each of which can be invoked from the client application using an Open Client method or Web service operation
- **SubAppObjects** — Additional collections of external procedures that can include both persistent and non-persistent procedures, each of which can be invoked from the client application using an Open Client method or Web service operation
- **ProcObjects** — Any persistent, single-run, or singleton procedures included in an AppObject or SubAppObject that provide one or more internal procedures or user-defined functions, each of which can be invoked from the client application using an Open Client method or Web service operation

These Open Client objects support each session model differently, and the choice of operating mode (hence, session model) affects the client programming required to access the AppServer procedures.

ProxyGen relies on the procedure prototypes of the application service to allow you to define the Open Client objects for an Open Client interface. The AppServer procedures themselves contain no code that is specific to the supported clients.

### Choosing a session model

The session model you choose determines how the AppServer services clients, and that choice depends on the type of application you want to support.
Session-managed model
The session-managed model is specifically designed for application services or Web services that support a single transaction across multiple requests from the same client, returning intermediate results with each request until the transaction is completed. Thus, the AppServer maintains context between requests for each client that it services, and each such client participates in a persistent connection to the AppServer. The connected AppServer is dedicated to serving that one client until the connection between them is terminated.

In this session model, all client requests are single-threaded, meaning that the AppServer does not handle another request from the same client until it has completed and responded to any pending request from that client.

Session-free model
The session-free model is specifically designed for application services or Web services that return a complete result, starting and ending any transactions, in a single request. Thus, the AppServer maintains no context for any client that it services. Requests from a session-free client are handled by any available AppServer (and AppServer agent) that supports the required application service. The session-free client (or the Web Services or OpenEdge Adapter for Sonic ESB on behalf of Web service clients) manages a pool of physical AppServer connections (the connection pool) that provides access to the session-free AppServers available to service client requests. Thus, a session-free AppServer executes requests from all clients as it receives them, and completes each request independently of any prior request (see State-free operating mode on page 30).

In this session model, all requests from a client are multi-threaded, meaning that multiple requests from a single client can be executed in parallel, as AppServer resources permit. The more AppServers (and AppServer agents) that are available (through a NameServer) to handle requests for a given application service (or Web service), the more requests that the application or Web service can handle simultaneously from a single client. Each session-free AppServer executes requests as fast as communications and AppServer agent resources permit. Thus, the same client can have multiple requests executed by an AppServer at one time, and each AppServer can execute requests from any number of clients at one time, both limited by the number and extent of AppServer agents configured for each AppServer.

Making the final choice
As an aid to choosing a session model, the following table shows major points of comparison between the session-free and session-managed models.

Table 1: Session-free and session-managed models compared

<table>
<thead>
<tr>
<th>Point comparison</th>
<th>Session-free</th>
<th>Session-managed</th>
</tr>
</thead>
<tbody>
<tr>
<td>AppServer operating mode</td>
<td>Supported by the state-free operating mode.</td>
<td>Supported by the state-aware, state-reset, and stateless operating modes.</td>
</tr>
<tr>
<td>Point comparison</td>
<td>Session-free</td>
<td>Session-managed</td>
</tr>
<tr>
<td>------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>AppServer application</td>
<td>Every application service or Web service request is independent of every</td>
<td>Every application service or Web service request is associated with a single</td>
</tr>
<tr>
<td>programming model</td>
<td>other request. No session context is maintained between the client and the</td>
<td>connected client and AppServer. Session context for that client can be</td>
</tr>
<tr>
<td></td>
<td>AppServers involved in the requests.</td>
<td>maintained by the AppServer across client requests.</td>
</tr>
<tr>
<td></td>
<td>The use of <code>RUN PERSISTENT</code> to instantiate remote persistent procedures is</td>
<td></td>
</tr>
<tr>
<td></td>
<td>not recommended. Instead, use the <code>SINGLE-RUN</code> or <code>SINGLETON</code> option (supported for ABL clients only).</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The use of connection-based interfaces (SubAppObjects and ProcObjects) is</td>
<td></td>
</tr>
<tr>
<td></td>
<td>not recommended. The use of connect and disconnect procedures in the AppServer configuration is not allowed.</td>
<td></td>
</tr>
<tr>
<td>Compatibility with different ABL clients</td>
<td>Compatible with all ABL clients, including Web service clients supported by the WSA (or OpenEdge Adapter for Sonic ESB) and Internet clients supported by the AppServer Internet Adapter (AIA).</td>
<td>Compatible with all ABL clients, including Web service clients supported by the WSA (or OpenEdge Adapter for Sonic ESB) and Internet clients supported by the AppServer Internet Adapter (AIA).</td>
</tr>
<tr>
<td>Point comparison</td>
<td>Session-free</td>
<td>Session-managed</td>
</tr>
<tr>
<td>-----------------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Client connection model</td>
<td>The client application does not physically connect to or disconnect from an AppServer. The client logically connects to an application service or Web service and simply sends requests to any available AppServer and handles the responses from wherever they are sent.</td>
<td>Before sending any other Web service request, the client application must first establish a physical connection to a single AppServer instance through an application service or Web service it wants to access. Once a physical connection is established, the AppServer can maintain session context for the client and application service, but for a Web service, both the Web service client and the Web service must maintain awareness of the physical connection using SOAP headers. The client developer must program the handling of these headers in the client application.</td>
</tr>
<tr>
<td>Communications between client and AppServer</td>
<td>Clients of the AppServer exchange messages transparently with any available AppServer resource using a pool of physical AppServer connections (connection pool) maintained by the WSA and OpenEdge Adapter for Sonic ESB for Web services and maintained by the client session for all other clients. Thus, AppServer resources handle all client requests in parallel, depending on the resources available.</td>
<td>Each client of the Web service creates a separate and exclusive connection over which all messages are exchanged with a particular AppServer resource until the connection is terminated. Thus, the AppServer resource handles each client request in sequence, executing each request only after the previous request from the client has completed.</td>
</tr>
</tbody>
</table>

Thus, if your application requires the AppServer to manage context for its clients, choose a session-managed operating mode (stateless, state-aware, or state-reset). Also, if you are Web-service-enabling an existing AppServer application that relies on a specific operating mode, you must continue to use that mode for the Web service unless you make the necessary changes in the application to use another operating mode.

If your application requires no context management on the AppServer and otherwise has no programming dependency on a session-managed operating mode, you can choose the session-free operating mode (state-free). For an existing session-managed AppServer application that meets these requirements (maintains no context), you can immediately change the AppServer operating mode to state-free with no code changes, thus converting the entire application to a session-free model with potentially greater performance benefits.

**Note:** If you do convert an existing AppServer application from the session-managed to the session-free model, be sure to set any AppServer instances that support the same application service to the state-free operating mode.
If you can use the session-free operating model, it has the following advantages over the session-managed model:

- Both application services and Web services scale much more readily compared to session-managed.
- The programming model for session-free applications—for the AppServer application and for the client application—is simpler than for session-managed applications.

For more information on programming:

- Session-free and session-managed AppServers, see Programming the AppServer on page 49.
- Session-free and session-managed ABL clients, see Programming ABL Client Applications on page 87.
- Session-free and session-managed Open Clients, see OpenEdge Development: Open Client Introduction and Programming.
- Session-free and session-managed Web service clients of OpenEdge Web services, see OpenEdge Development: Web Services.

**Note:** If your aim in choosing the session-free model is greater scalability and performance, and you also require secure connections, note that using either (or both) the AIA with HTTPS or an SSL-enabled AppServer tends to incur heavy performance penalties regardless of the session model that you choose. For more information, see the sections on security considerations in Design and Implementation Considerations on page 137.

Whatever session model your AppServer application supports, you must also perform these tasks for a Web service:

- Specify the same session model for any Web service definition based on the same application service. For more information on specifying the session model for a Web service definition using ProxyGen, see OpenEdge Developments: asaps_design-and-implementation-considerations_oe.xml: Open Client Introduction and Programming.
- Manage the Web service through the WSA (or OpenEdge Adapter for Sonic ESB) according to the same session model. For more information on managing Web services for a given session model, see the chapters on Web service management and deployment in OpenEdge Application Server: Administration.

### Context management and the session-free model

A session-free AppServer can provide limited context management using a client-instantiated persistent procedure (Open Client or Web service ProcObject instantiation) on the AppServer. However, Progress Software Corporation recommends avoiding the use of persistent procedures and ProcObjects in session-free applications. The programming is more complex and their use can degrade application performance.

A far more useful means of managing session-free context is to use a persistent storage mechanism, such as the OpenEdge RDBMS, to maintain the context for all clients of an application. In effect, this is no different than using a database in any session model, except that you might require more parameters on a remote request to identify the context (such as a customer order) to be maintained.
Deciding on the scope of individual requests

When designing an application service, the scope (or power) of each remote operation ultimately depends on the transaction requirements of the application. However in general, it is a good practice to make operation requests as powerful as possible in order to hide the details of the AppServer ABL from the client programmer. Also, the more power that remote operations have, the fewer requests must be made from the client to accomplish tasks over the network, thus reducing the opportunity for the network to be a bottleneck.

Session-managed operations

For session-managed applications, which support complex transactions, the operations can perform simple (less powerful), fine-grained procedures and functions because the ultimate result for a client does not have to be completed in a single request. Each procedure or function executed in a series of session-managed requests can help to assemble the context of a single transaction until a single committing operation completes the transaction after this context is complete. An example might be a transaction to assemble a bill of materials (BOM), where the transaction depends on what components are already in stock for the BOM, requiring the client to invoke different operations to complete the structure of the transaction. Once the structure is completed for any missing components, the BOM transaction can be committed using a final operation.

Session-free operations

For a session-free application, whether simple or complex (powerful), each operation must complete any transaction that it starts. Operations typical of this type of application include returning the current outdoor temperature given a zip code, or returning a zip code given a street address. A more powerful example might be an operation that returns all the available seats on airline flights to a given destination, followed by another operation that attempts to reserve a selected list of seats on any of the available carriers, based on a set of parameterized preferences, or specifically by seat number. In all cases, these operations complete their function in one request, and if a result is in some way unsuccessful, the request might simply be repeated with or without modification of the parameters involved.

Context can, however, be maintained for a session-free application if the client maintains the context between requests using input-output parameters provided by the application service procedures or Web service operations. An example might be an order entry Web service that allows a user to obtain status of a particular order, then examine the detail lines of that order. In this case the context might be a customer number passed in a getOrders procedure to get the status of all that customer's orders. This might be followed by a getOrderLines procedure for a selected order, with both the customer number and order number passed as parameters to get the detail lines of a selected order for the customer. In this case, the context is really being maintained in persistent storage, such as a database, that is shared by all clients of the application.

Load balancing and session models

If you deploy an application using load balancing, the NameServer distributes client load according to configured proportions across multiple AppServer instances that support the same application service.
Session-managed load balancing
If your application is session-managed, load balancing occurs at the time the client connects to the application service. That is, the NameServer distributes the load by connecting the client to an AppServer according to the configured proportion and all requests from that client go to the connected AppServer.

Session-free load balancing
If your application is session-free, load balancing occurs both at the time of the initial logical connection and for every remote request invoked on that logical connection. For example:

1. When it first connects to an application service, the client asks the NameServer what AppServer broker it should connect to as many times as there are initial connections specified for the binding and creates a physical connection to each broker as the NameServer specifies. The NameServer specifies the brokers for the initial connection request based on the load-balancing settings for the application service. The result of this physical connection process is the connection pool where the number of connections are distributed across the available AppServers approximately according to the configured load-balancing formula.

2. When it invokes a remote procedure request to a connected session-free application service, the client asks the NameServer what broker to use for the remote request. The NameServer returns the broker to use according to the configured load-balancing formula. If the client already has a physical connection available to the broker the NameServer specifies, the client uses that connection to run the request. If the client has no available connection to that broker, the client creates another connection to execute the request, up to the maximum number of physical connections specified for the binding. If creating a connection will exceed this connection maximum, the request is queued for the next available connection on any AppServer broker in the connection pool. If creating such a connection fails (because the broker is unavailable), OpenEdge raises a run-time error on that request.

So, for a session-free application service, load balancing is determined with finer granularity than for a session-managed application service.

Configuring load balancing
For more information on load balancing an AppServer application, see the information on AppServer administration in OpenEdge Application Server: Administration. For information on how to specify the initial and maximum connections for ABL clients connecting to a session-free application service, see Programming ABL Client Applications on page 87. For information on how Web service properties affect session configuration, see the information on Web service administration in OpenEdge Application Server: Administration.

Message compression support
You can compress messages exchanged between the client and the AppServer. Message compression results in lower bandwidth usage. Message compression can also improve message throughput by reducing network usage. This will benefit network environments such as dial-up communication and WAN configurations. ABL clients, Java Open Clients, and .NET Open Clients support message compression. You must enable compression on a client to compress messages exchanged with an AppServer.

You can enable message compression using:
• **ABL client:** Use the Network Message Compression (-mc) startup parameter to enable message compression for ABL clients. For more information on the Network Message Compression startup parameter, see *OpenEdge Deployment: Startup Command and Parameter Reference*.

• **Java Open Client and .NET Open Client:** Use the property `CompressionEnabled` to enable compression of messages between Java Open Client/.NET Open Client and the AppServer. Use `CompressionLevel` property to determine the level of compression of a message exchanged between the client and the AppServer. Use `CompressionThreshold` property to determine the minimum size (in bytes), that the message must achieve before it gets compressed. For more information on these properties, see *OpenEdge Development: Java Open Clients* and *OpenEdge Development: .NET Open Clients*.

## Understanding synchronous and asynchronous requests

The ABL client and AppServer support the following two types of remote procedure execution for both session-managed and session-free applications:

• **Synchronous request** — (Default) Where the client blocks and waits for the result of the remote request before continuing execution

• **Asynchronous request** — Where the client continues execution after initiating the request and processes the result whenever the AppServer makes it available

The ABL client also supports asynchronous requests to industry Web services, which work much like session-free asynchronous requests to an AppServer. For more information on ABL asynchronous requests to a Web service, see *OpenEdge Development: Web Services*.

**Note:** OpenEdge does not provide built-in support for asynchronous requests in Open Clients or for OpenEdge Web services. You can implement your own asynchronous request mechanism in Java Open Clients using threads (see *OpenEdge Development: Java Open Clients*).

## Process control flow

The following figure compares the process flow within an application using synchronous and asynchronous requests.
In this figure, one client (Client1) makes synchronous requests and another client (Client2) makes asynchronous requests to two AppServers (Server1 and Server2). For Client1, control flows in sequence between client and AppServer. For Client2, control flows in parallel between client and AppServer. Assuming that the clients and AppServers perform the same work, Client2 can conceivably complete the work in less time than can Client1. The advantage for Client2 comes with the parallelism that asynchronous requests can achieve. In this case, the actual advantage that Client2 affords depends on its ability to respond to the results from the AppServers at the moment they arrive, while doing its own share of the work until the results actually do arrive.

**Specifying the type of remote request**

The type of remote request (synchronous or asynchronous) is determined by options on the `RUN` statement. The AppServer in a session-managed application can handle both types of requests in the same AppServer session and over the same client connection. For a session-free application, OpenEdge also supports both types of request for any client binding to an application service.

You can initiate as many asynchronous requests as you want at any point during client execution. You can also mix synchronous and asynchronous requests in the same application. The one constraint is that you can initiate a synchronous request only if there are no pending (incomplete) asynchronous requests over the same AppServer connection (session-managed) or for the same application service binding (session-free). OpenEdge also provides a mechanism for you to check the status of asynchronous requests for a session-managed connection or session-free binding.
Note: As suggested earlier, the AppServer agent that executes a request does not know if the request is synchronous or asynchronous. The AppServer broker manages each request on behalf of the client, and dispatches the request for execution according to the AppServer operating mode (state-reset, state-aware, stateless, or state-free). Consequently, the client can execute synchronous and asynchronous requests without regard to the AppServer operating mode.

Handling asynchronous request results

OpenEdge allows your application to handle asynchronous request results using an event procedure that executes in response to a PROCEDURE-COMPLETE event. As with other ABL events, this event is handled in the context of the next PROCESS EVENTS statement or blocking I/O statement (such as WAIT-FOR) following the activation of the PROCEDURE-COMPLETE event.

The asynchronous event procedure returns any results from the request to its own context block, using appropriate input parameters that you define according to the signature of the remote procedure that you invoke asynchronously. Because the event procedure signature must correspond to the signature of the remote procedure, you typically must define a unique event procedure to handle the results for each remote procedure that you call asynchronously.

The timing and availability of asynchronous request results is, of course, variable. However, the order of the results returned for multiple asynchronous requests differs markedly, depending on the application session model and implementation, as described in the following sections.

Asynchronous request queuing

To allow a client to submit a sequence of asynchronous requests over the same session-managed connection or the same session-free binding, OpenEdge maintains a queuing system to manage such requests between the client and the AppServer or application service. OpenEdge handles multiple asynchronous requests in this queuing system differently, depending on the session model:

- **Session-managed connections** — The AppServer can execute only one asynchronous request at a time over a single client connection. So, results from session-managed asynchronous requests are returned to the client in the order they are sent.

- **Session-free bindings** — The application service can handle multiple asynchronous requests from the same client in parallel. So, results from session-free asynchronous requests are returned to the client in no predictable order.

Therefore, you just write your application to account for the difference in how results are returned for each session model.

Regardless of the session model, the asynchronous request queuing system is the same, and includes three queues:

- One *send queue* per AppServer connection, for queuing asynchronous requests to the AppServer.

- One *response queue* per AppServer connection, for queuing the results of each asynchronous request.

- The ABL *event queue* on the client for handling all ABL events, including user-interface and other events, such as PROCEDURE-COMPLETE. This event queue handles the PROCEDURE-COMPLETE event for all AppServer connections on a client.
Understanding the session-managed queuing model

For a given AppServer connection, each asynchronous request is executed in the order that it is submitted, and the event procedures for those requests are executed in the same order that the requests are executed. This order is maintained by the send and response queues associated with the connection, as well as the ABL event queue used to raise `PROCEDURE-COMPLETE` events.

The send queue queues the requests that are submitted for execution by the client. The response queue queues the responses received on a given AppServer connection for asynchronous requests that have completed execution, but whose event procedure has not yet been executed. Finally, the ABL event queue queues the `PROCEDURE-COMPLETE` event for each completed request as it is received from the response queue.

The following figure shows how this works, where 1a is the first asynchronous request (AsyncRequest 1a) to AppServer 1 and 2a is the first asynchronous request (AsyncRequest 2a) to AppServer 2, and so forth.

**Figure 2: Session-managed asynchronous request queuing**

AsyncRequest refers to the execution of an asynchronous `RUN` statement, and EventProc refers to the execution of an event procedure in response to the `PROCEDURE-COMPLETE` event handled from the ABL event queue. These requests are further identified by 1 for the AppServer 1 connection and 2 for the AppServer 2 connection, and a letter indicating the order of the request on the client. For simplicity, the example assumes that each AppServer has only one agent to execute requests.

On a given connection (AppServer 1 or AppServer 2), if an asynchronous request is submitted for execution when there is already a request executing on the AppServer, the new asynchronous request is queued on the send queue until all previously submitted asynchronous requests for that connection have executed. Such is the case for AsyncRequests 1d and 2e. The new asynchronous request is only sent to the AppServer for execution once all prior asynchronous requests have completed execution on the AppServer. In the previous figure, AsyncRequests 1b and 2c are currently executing on their respective AppServers.
As each request completes execution, the response to the request is queued on the appropriate response queue and the next request in the send queue is sent to the AppServer. In the previous figure, AsyncRequests 1c and 2d are next in line for execution. Each response remains in the response queue until the corresponding event procedure can be executed in the context of a \texttt{PROCESS EVENTS} or blocking I/O statement. In the previous figure, the \texttt{PROCEDURE-COMPLETE} events for AsyncRequests 1a, 2a, and 2b are all pending execution of their respective event procedures.

The response to an asynchronous request can be received by the client application at any time, but the event procedure associated with that request can be run only if the application is executing a \texttt{PROCESS EVENTS} or blocking I/O statement. Note that the order in which the client handles pending responses for different AppServer connections is undefined. In the previous figure, either the event procedure for AsyncRequest 1a or AsyncRequest 2a can execute first.

Thus, OpenEdge guarantees that the responses from asynchronous requests on the same AppServer connection are handled in order of their execution on the client. However, it does not guarantee the order in which responses are handled from multiple AppServer connections.

### Understanding the session-free queuing model

For a given logical connection to an application service, asynchronous requests are executed in no determined order, and the event procedures for those requests are executed in no determined order. Unlike requests sent over physical connections, which are handled sequentially, requests sent over logical connections are handled in parallel and completed in a manner that depends entirely on available AppServer resources.

The send queue queues the requests that are submitted for execution by the client in the order that the client submits them. The response queue queues the responses received on a given application service connection for requests that have completed execution. Finally, the ABL event queue queues the \texttt{PROCEDURE-COMPLETE} event for each completed request as it is received from the response queue. However, it is the availability of AppServer resources in the application service connection pool that determines when and how these requests are actually processed.

The following figure shows how this works, with nine asynchronous requests submitted on the client. In this example, the requests are assumed to be sent for a single application service binding that is supported by two AppServer instances (A and B).
For simplicity, the example assumes that each AppServer has only one agent to execute requests. However, the principle is the same for one AppServer running two agents to execute the requests. The main difference is that two App Servers are likely to provide higher general availability for the application service. AsyncRequest refers to the execution of an asynchronous run statement, and EventProc refers to the execution of an event procedure in response to the procedure-complete event handled from the ABL event queue. The requests are numbered in order of execution.

On a given application service binding, if an asynchronous request is submitted for execution when all AppServer resources are unavailable, the next asynchronous request is queued on the send queue until all previously submitted asynchronous requests for that connection have executed. Such is the case for AsyncRequests 8 and 9. The most recent asynchronous request (9) is only sent to an AppServer for execution once AsyncRequest 8 has been submitted to an available AppServer for execution. All prior asynchronous requests have completed execution (1, 2, 4, 5, and 6) or are executing on an AppServer (3 and 7).

Note that although all these requests have been submitted by the client in order from 1 to 9, AsyncRequest 3 is still being executed, and AsyncRequests 4 through 6 have already completed. If this was a session-managed application, AsyncRequests 4 through 7 would still be in the send queue, waiting for 3 to complete. Because the application is session-free, AsyncRequest 3 has taken so long to complete execution on AppServer B that AsyncRequests 4 through 6 have already completed execution on AppServer A and AsyncRequest 7 is still executing on AppServer A.

Assuming that 3 completes while 7 is still running, 3's results will appear in the response queue ahead of 7 and AsyncRequest 8 will likely begin executing on AppServer B. Note that when this series of requests began, AsyncRequests 1 and 2 would have executed at the same time on AppServers A and B, but 2 clearly completed before 1 and passed its results back ahead of it.

So, while results might return faster in a session-free application, the application must be written to handle them in any order. An example might be populating a temp-table with the results of each request ordered by an index, where the order of population is therefore irrelevant.
Note, again, in the previous figure, if AppServers A and B were actually two agents of a single AppServer, the order of completion might be exactly the same, especially if the two-AppServer scenario balances load evenly, because a state-free AppServer distributes its requests evenly to all of its available agents. The one feature that could very well change the order of completion between the two-AppServer and the one-AppServer scenarios is if the two-AppServer scenario used load balancing and had a balance of load set significantly different between the two AppServers.

**Asynchronous requests and internal procedures**

The parallelism of execution and the order of request completion shown in Figure 3 assume that each asynchronous request executes a remote external procedure non-persistently. If you invoke a remote external procedure persistently, then invoke requests to its internal procedures asynchronously, execution of these requests is handled sequentially on the same agent bound to the client by the remote persistent procedure (see Context management and the session-free model on page 38. The effect is essentially the same as if the requests were invoked on a session-managed connection (see Figure 2). However, any other asynchronous requests to non-persistent external procedures continue to execute in parallel to the extent that resources allow.

**Tuning the size of the send and response queues**

For each connection, OpenEdge allocates the memory for the send and response queues with a default size. If asynchronous requests are such that they exceed the send and response queue size, OpenEdge stores additional queued requests in a temp-table associated with the client session. Thus, performance can vary based on whether requests are stored in memory or in the temp-table.

To help you maximize the performance of asynchronous requests, OpenEdge provides the Async Queue Size (-asyncqueuesize) client startup parameter. This parameter allows you to specify the number of bytes that you want OpenEdge to allocate for the send and response queues on an AppServer connection.

The Async Queue Size parameter only applies when you create an AppServer connection on the client, and it applies equally to each such connection. Also, although OpenEdge creates the send and response queues for each connection, they are not used unless you make asynchronous requests to the AppServer. Thus, you can maximize AppServer performance by choosing a value based on the amount of data that you expect to flow between the client and the AppServer for these asynchronous requests. Similarly, if you are making no asynchronous requests, you can choose a low value to reduce the space allocated to queues that you are not planning to use.

For more information on this startup parameter, see the Async Queue Size (-asyncqueuesize) entry in OpenEdge Deployment: Startup Command and Parameter Reference.
Programming the AppServer

This chapter describes how to program procedures that execute on the AppServer. For details, see the following topics:

- Programming for an AppServer session model
- ABL for programming AppServer procedures
- Accessing the connection ID
- Accessing the name of the current remote procedure
- Using AppServer configuration procedures
- Controlling AppServer entry points
- Managing stateless connection context
- Managing state-free and stateless client context
- Understanding remote procedure handles
- Handling conditions and returning values
- Managing transactions
- Programming for Open Client applications
Programming for an AppServer session model

The most fundamental decision to make when designing and programming an AppServer application is what session model it will use, session-free or session-managed. This decision limits your choice of AppServer operating mode and the available features of ABL, some of which support particular operating modes. The decision of which session model to use depends on your particular application requirements, which cover all areas of the application, including network capabilities, available AppServer resources, and critical application functions.

Once you have determined the session model to use, you must design and build the AppServer procedures to support it. Typical applications (especially Open Client and Web service applications) start with development of an AppServer application that you then make available to the client application using the appropriate ABL, Open Client Toolkit, or Web service tools. You can also use parallel development techniques appropriate to the client type to develop both sides of the application at the same time. Whatever approach you use, the session model that you choose fundamentally determines the options for application development that are available to you. For more information on the differences between these two operating modes, see AppServer and Client Interaction on page 23.

Note: This chapter addresses only programming issues specific to AppServer procedures. It assumes that you are generally familiar with writing and executing ABL procedures. For more information on using ABL, see OpenEdge Getting Started: ABL Essentials.

Session-free programming

You can develop a session-free (state-free operating mode) application much like a stateless session-managed application. However, a session-free application must follow these additional AppServer programming requirements and recommendations:

• Every request (external procedure) runs independently of every other request on the AppServer. No request can create global or persistent data that any subsequent request can ever expect to find, because each request can execute on a different AppServer or AppServer agent.

• You cannot specify a connect or disconnect procedure, and none is ever executed on the AppServer.

• You can specify Activate and Deactivate procedures to execute before and after each request, but you cannot use them to maintain any contextual data that is not read and written to persistent storage (a database).

• Do not use persistent procedures of any kind, unless you cannot avoid it. (For ABL clients, as an alternative to `RUN PERSISTENT`, consider using the `SINGLE-RUN` or `SINGLETON` option.) Using any persistent procedures (Open Client and Web service ProcObjects) that are called and instantiated directly by the client forces a bound connection to the AppServer, which limits application performance and scalability (see Affecting application scalability on page 51). Persistent procedures also complicate the programming of session-free client applications, which must maintain connection context with every call to an internal procedure or user-defined function. Also, using the AppServer session to create a persistent procedure internally in response to some other client invocation has limited, if any, value. No external procedure invoked by the client can expect to access any persistent resources created by an AppServer session in response to another client invocation, because the latest remote external procedure is likely to execute in a different AppServer agent than the one in which any of these AppServer-created persistent resources reside.
Affecting application scalability

If you instantiate a remote persistent procedure in a session-free application, any network resource that provides the bound connection for the client is unavailable for other clients until that remote persistent procedure is deleted (or released for an Open Client or Web service ProcObject). If enough clients simultaneously create and use instances of this persistent procedure, the application might have no more physical connections available in its connection pool to serve additional clients.

Thus, if all clients that use the application service are always required to create an instance of this remote persistent procedure, the application service is scalable only for the number of clients that can simultaneously create that remote persistent procedure. Even if clients are not required to create this persistent procedure for all uses of the application service, any clients that do so reduce the availability of network resources to other clients and limit the scalability of the application service accordingly.

Thus, to allow a session-free application service to scale fully for the network resources available, do not allow any clients to instantiate remote persistent procedures. For Open Client and Web service applications, you can do this by not defining any ProcObjects in the Open Client interface.

SESSION handle attributes and state-free operating mode

The LOCAL-VERSION-INFO, CURRENT-REQUEST-INFO, and CURRENT-RESPONSE-INFO attributes on the SESSION handle support client context management for the state-free operating mode.

The functionality of the following SESSION handle attributes is modified or not supported with the state-free operating mode:

• SERVER-CONNECTION-BOUND-REQUEST
• SERVER-CONNECTION-BOUND
• SERVER-CONNECTION-CONTEXT
• SERVER-CONNECTION-ID

For more information, see ABL for programming AppServer procedures on page 52.

Session-managed programming

The requirements for programming an AppServer for session-managed applications differ among the available operating modes, state-reset, state-aware, and stateless. However, all of them provide some means to manage session context internally on behalf of clients that are physically connected to the session-managed AppServer. The remaining sections in this chapter indicate where a feature's function or availability depends on the operating mode. However, in general, the main differences affect how you can manage client connection context for each mode. For more information on the session-managed application model, in general, and how the different operating modes support it, see AppServer and Client Interaction on page 23.
# ABL for programming AppServer procedures

The following table lists the ABL elements that either are valid only in an AppServer session or have special application in AppServer programming. The remaining sections in this chapter explain how to use these elements.

## Table 2: ABL for programming AppServer procedures

<table>
<thead>
<tr>
<th>ABL element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENT-REQUEST-INFO</td>
<td>An object reference attribute on the SESSION system handle that references a Progress.Lang.OERequestInfo object, which describes the current client request being executed.</td>
</tr>
<tr>
<td>CURRENT-RESPONSE-INFO</td>
<td>An object reference attribute on the SESSION system handle that references a Progress.Lang.OERequestInfo object, which describes the response to the client request being executed.</td>
</tr>
<tr>
<td>DEFAULT-COMMIT</td>
<td>A LOGICAL attribute on a transaction object handle that specifies how the transaction object is to complete the transaction if an automatic transaction terminates with no prior SET-COMMIT( ) or SET-ROLLBACK( ) being invoked.</td>
</tr>
<tr>
<td>DELETE OBJECT handle</td>
<td>A statement that you can use to delete certain session objects, including local and remote persistent procedures.</td>
</tr>
<tr>
<td>DELETE PROCEDURE procedure-handle</td>
<td>A statement that you can use to delete both local and remote procedure objects (persistent procedures).</td>
</tr>
<tr>
<td>EXPORT ( [list] )</td>
<td>A method on the SESSION system handle that creates and modifies the list of remote procedures provided by the AppServer.</td>
</tr>
<tr>
<td>FUNCTION ... IN SUPER</td>
<td>A statement that specifies the prototype definition for a user-defined function defined within a super procedure.</td>
</tr>
<tr>
<td>IS-OPEN</td>
<td>A LOGICAL attribute on a transaction object handle that returns TRUE if a database transaction is active. This is identical to the ABL TRANSACTION function.</td>
</tr>
<tr>
<td>LOCAL-VERSION-INFO</td>
<td>An object reference attribute on the SESSION system handle that references a Progress.Lang.OEVersionInfo object, which identifies the current instance of the OpenEdge AVM, including its OpenEdge version.</td>
</tr>
<tr>
<td>ABL element</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>Parameters for procedures and user-defined functions</td>
<td>Procedure and user-defined function parameters can be defined as any ABL parameter data type, including class-based object types. (For more information see Passing class-based objects as parameters between an AppServer and an ABL client on page 112.) However, note that for any procedure or user-defined function that you include in an OpenEdge Web service or Sonic ESB Adapter service, you cannot pass a static or dynamic ProDataSet parameter that includes parent-id data-relations (for example, data-relations as defined using the <code>PARENT-ID-RELATION</code> option of the <code>DEFINE DATASET</code> statement). Otherwise, ProxyGen fails to generate any proxies for the specified service definition.</td>
</tr>
<tr>
<td>PERSISTENT</td>
<td>A logical attribute on procedure handles and the <code>THIS-PROCEDURE</code> system handle that is <code>true</code> if the specified procedure is running persistently as a procedure object.</td>
</tr>
<tr>
<td>PROCEDURE ... IN SUPER</td>
<td>A statement that begins the prototype definition for an internal procedure defined within a super procedure.</td>
</tr>
</tbody>
</table>
| REMOTE | A logical attribute on the `SESSION` system handle that is `true` if the current session is running in the context of an AppServer.  
A logical attribute on procedure handles and the `THIS-PROCEDURE` system handle that is `true` if the specified procedure is running at the top level in the context of an AppServer (as the result of a remote procedure call by a client application). Always `false` if `PROXY` is `true`. |
<p>| SERVER-CONNECTION-BOUND | A logical attribute on the <code>SESSION</code> system handle that is <code>true</code> if the AppServer agent is bound to a particular client session in stateless and state-free operating modes by client invocation of a remote persistent procedure or Open Client/Web service ProcObject. By default, it is <code>false</code>. Valid only if the <code>REMOTE</code> attribute is <code>true</code>. |
| SERVER-CONNECTION-BOUND-REQUEST | A logical attribute on the <code>SESSION</code> system handle that, when set to <code>true</code>, requests that the AppServer agent be bound to the current client connection identified by the <code>SERVER-CONNECTION-ID</code> attribute. When set to <code>false</code>, requests that the AppServer agent be unbound from the currently bound client connection pending deletion of all remote persistent procedures running in the session. Valid only if the <code>REMOTE</code> attribute is <code>true</code> and the operating mode is stateless. In state-free operating mode, any attempt to set this attribute raises a <code>WARNING</code> condition in the AppServer agent, which writes a message to the AppServer log file, and the value remains unchanged. You |</p>
<table>
<thead>
<tr>
<th>ABL element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SERVER-OPERATING-MODE</td>
<td>A CHARACTER attribute on the SESSION system handle that returns the operating mode specified by the operatingMode property set for this AppServer in the ubroker.properties file. Valid only if the REMOTE attribute is TRUE.</td>
</tr>
<tr>
<td>SERVER-CONNECTION-ID</td>
<td>A CHARACTER attribute on the SESSION system handle that returns the run-time connection ID of the current client connection assigned to this AppServer session. Valid only if the REMOTE attribute is TRUE. In state-free operating mode, this value has no meaning.</td>
</tr>
<tr>
<td>SERVER-CONNECTION-CONTEXT</td>
<td>A CHARACTER attribute on the SESSION system handle that contains an application-determined value that you can set. OpenEdge passes this value to each AppServer agent that executes a request on behalf of the client connection identified by the SERVER-CONNECTION-ID attribute. Valid only if the REMOTE attribute is TRUE. In state-free operating mode, any attempt to set this attribute raises a WARNING condition in the AppServer agent, which writes a message to the AppServer log file, and the value remains unchanged. You can handle the WARNING condition by including the NO-ERROR option in the statement that attempts to set the value, and checking ERROR-STATUS:NUM-MESSAGES for a value greater than zero. In state-free operating mode, this attribute always has the Unknown value (?).</td>
</tr>
<tr>
<td>SET-COMMIT( )</td>
<td>A method on a transaction object handle that tells the transaction object to commit any automatic transaction when the current request completes and returns execution to the client.</td>
</tr>
<tr>
<td>SET-ROLLBACK( )</td>
<td>A method on a transaction object handle that tells the transaction object to roll back any automatic transaction</td>
</tr>
<tr>
<td>ABL element</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>when the current request completes and returns execution to the client.</td>
</tr>
<tr>
<td>SINGLE-RUN</td>
<td>A LOGICAL attribute on procedure handles and the THIS-PROCEDURE system handle that is TRUE if the specified procedure is running as a single-run object.</td>
</tr>
<tr>
<td>SINGLETON</td>
<td>A LOGICAL attribute on procedure handles and the THIS-PROCEDURE system handle that is TRUE if the specified procedure is running as a singleton object.</td>
</tr>
<tr>
<td>TRANSACTION</td>
<td>A HANDLE attribute on procedure handles and the THIS-PROCEDURE system handle that provides a handle to the current transaction object.</td>
</tr>
<tr>
<td>TRANS-INIT-PROCEDURE</td>
<td>A HANDLE attribute on a transaction object handle that, if an automatic transaction is active, returns the procedure handle to the transaction initiating procedure that started the transaction.</td>
</tr>
</tbody>
</table>

Accessing the connection ID

As described in ABL for programming AppServer procedures on page 52, OpenEdge creates a unique ID for each connection to an AppServer. For session-managed applications, there is a connection ID mechanism especially for an AppServer running in stateless operating mode. This is the SERVER-CONNECTION-ID attribute on the SESSION handle, a read-only CHARACTER attribute for which OpenEdge generates a unique value for each connection to a session-managed AppServer. For information on how the AppServer generates the value for a client connection ID, see the information on the AppServer connection-based model and physical connection IDs in OpenEdge Getting Started: Application and Integration Services.

For both session-managed and session-free applications, you can use the ClientContextId property on the Progress.Lang.OERequestInfo object to identify a single client context. This client context identifier can be used to identify a single user login session, especially for multiple AppServer sessions running in a stateless or state-free operating mode. By default, OpenEdge provides a globally unique value for the ClientContextId property that can also be managed by the application across multiple AppServer sessions.

For more information on using the SERVER-CONNECTION-ID, see Managing stateless connection context on page 66. For more information on using the ClientContextId property, see Managing state-free and stateless client context on page 69.
Accessing the name of the current remote procedure

An OpenEdge AppServer runs certain configuration procedures when a client runs external or internal procedures. AppServer configuration procedures are ABL procedures that you specify when you configure an AppServer; the procedures run typically before and after invoking remote procedures during an AppServer agent's lifetime.

One set of configuration procedures includes the Activate and Deactivate procedures. These procedures encapsulate logic that executes automatically for certain client requests (remote procedures) on an AppServer running in stateless and state-free operating modes.

With the help of the read-only `ProcedureName` property on the `Progress.Lang.OERequestInfo` class, you can access the name of the internal or external procedure invoked by the ABL client. You can use the `ProcedureName` property to identify the client-invoked procedure in any procedure currently running on the AppServer, including configuration procedures, such as the Activate and Deactivate procedures. You can use the instance of `OERequestInfo` referenced by the `CURRENT-REQUEST-INFO` attribute on the AppServer agent's `SESSION` system handle. For more information on this attribute, see the Managing state-free and stateless client context on page 69.

The `ProcedureName` property captures the name of the procedure along with the relative or absolute path as specified in the client code that runs the remote procedure. If no path is specified with the external procedure filename in the client's `RUN` statement, the property captures only the procedure filename.

The following table lists the procedure path or filename for various requests invoked by a client, and provides a description of each `ProcedureName` property setting as a result.

<table>
<thead>
<tr>
<th>Table 3: RUN requests and <code>ProcedureName</code> property values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RUN request on the ABL client</strong></td>
</tr>
<tr>
<td>RUN test/foo.p ON SERVER hdl.</td>
</tr>
<tr>
<td>RUN C:/OpenEdge/WRK/test/foo.p ON SERVER hdl</td>
</tr>
<tr>
<td>RUN bar.p ON SERVER hdl</td>
</tr>
</tbody>
</table>

**Note:** For execution of an internal procedure defined in a remote persistent, single-run, or singleton procedure, `ProcedureName` has the format `External-Procedure-Name&Internal-Procedure-Name`, where `External-Procedure-Name` is the name of the remote external procedure and `Internal-Procedure-Name` is the name of its executing internal procedure.
Accessing a procedure name on the AppServer

You can access the path or filename of a remote procedure running on the AppServer that an ABL client has invoked using the following code fragment:

```
DEFINE VARIABLE rqInfo AS Progress.Lang.OERequestInfo NO-UNDO.
DEFINE VARIABLE callerProcedureName AS CHARACTER NO-UNDO.
rqInfo = SESSION:CURRENT-REQUEST-INFO.
callerProcedureName = rqInfo:ProcedureName.
```

Suppose external procedures on the AppServer define at least one of two internal procedures, `UsefulRoutine1` and `UsefulRoutine2`, that implement some logic in two different ways, as in the following external procedure fragment:

```
/*
 * h-UsefulProc.p
 */
PROCEDURE UsefulRoutine1:
 /*Does something useful */
END PROCEDURE.
PROCEDURE UsefulRoutine2:
 /*Does something useful in a different way*/
END PROCEDURE.
```

An ABL client might invoke these internal procedures in persistent procedures currently running on this AppServer as shown in the following client code fragment:

```
DEFINE VARIABLE happsrv AS HANDLE NO-UNDO.
DEFINE VARIABLE hUseful AS HANDLE NO-UNDO.
DEFINE VARIABLE hProc AS HANDLE NO-UNDO.
CREATE SERVER happsrv.
IF happsrv:CONNECT(" -H localhost -S 3090 -DirectConnect -AppService asbroker1","","") THEN
 DO:
  RUN h-UsefulProc.p ON happsrv PERSISTENT SET hUseful.  hProc =
  happsrv:FIRST-PROCEDURE.
  DO WHILE VALID-HANDLE(hProc):
   IF LOOKUP("UsefulRoutine1", hProc:INTERNAL-ENTRIES) NE 0 THEN
     RUN UsefulRoutine1 IN hProc.
   ELSE
     RUN UsefulRoutine2 IN hProc.  hProc = hProc:NEXT-SIBLING.
   END.
  DELETE PROCEDURE hUseful.
  happsrv:DISCONNECT().
  END.
ELSE
  MESSAGE "Failed to connect" VIEW-AS ALERT-BOX.
END.
```

For the client remote request that runs `h-UsefulProc.p` persistently, the value of the `ProcedureName` property is "h-UsefulProc.p".

For a client remote request that runs one of the specified internal procedures in `hProc` when the remote procedure handle refers to any instance of the `h-UsefulProc.p` persistent procedure currently instantiated on the AppServer, the value of the `ProcedureName` property is one of the following, depending on the internal procedure that the client is executing:

- "h-UsefulProc.p&UsefulRoutine1"

OpenEdge Application Server: Developing AppServer Applications
Using AppServer configuration procedures

AppServer configuration procedures are ABL procedures that you can specify when you configure an AppServer, and that run at specific times during an AppServer agent's lifetime. The three procedure types are described in the following sections:

- Startup and Shutdown procedures on page 58
- Connect and Disconnect procedures on page 59
- Activate and Deactivate procedures on page 61

Each of these procedures is run at specific times during the lifetime of an AppServer agent with the idea of simplifying the process of deploying an application with the AppServer.

Startup and Shutdown procedures

Startup and Shutdown procedures encapsulate logic that executes during the startup and shutdown of an AppServer. In effect, the logic in these procedures executes in each AppServer agent as it starts up and shuts down. Functionality that might go in a Startup or Shutdown procedure includes connecting or disconnecting databases, loading or storing the contents of temp-tables, and instantiating or deleting certain local persistent procedures.

Startup procedure

A Startup procedure executes as an AppServer agent starts up. You can specify the name of an AppServer Startup procedure using the OpenEdge Explorer or OpenEdge Management or by setting the `srvrStartupProc` property in the AppServer properties file (`ubroker.properties`).

The Startup procedure takes a character string as an input parameter. For example:

```
DEFINE INPUT PARAMETER startup-data AS CHARACTER NO-UNDO.
```

You can set the value for this parameter in the properties file using the `srvrStartupProcParam` property, and you can set it to any arbitrary value. If you do not specify a value for this property, the parameter is set to the `Unknown value (?)` when the AppServer agent executes the Startup procedure.

The Startup procedure is automatically executed as a persistent procedure within each AppServer agent when the AppServer agent first starts. The persistent procedure is automatically deleted just before the agent terminates. If you do not need the context established by the Startup procedure, you can delete the persistent procedure within an AppServer agent context by executing the `DELETE PROCEDURE` (or `DELETE OBJECT`) statement using the appropriate procedure handle.

If the Startup procedure completes with no error, the AppServer starts up successfully. If the Startup procedure completes with an error return value (`RETURN ERROR` statement), AppServer startup fails and the AppServer agent is terminated.
Shutdown procedure

A Shutdown procedure executes just before the AppServer agent is shut down. A Shutdown procedure takes no parameters. You can specify the name of an AppServer Shutdown procedure using the OpenEdge Explorer or OpenEdge Management or by setting the `srvrShutdownProc` property in the AppServer properties file (`ubroker.properties`).

Unlike the Startup procedure, the Shutdown procedure is run as a non-persistent procedure, and any errors propagated by the Shutdown procedure are simply ignored. The AppServer agent terminates immediately after the Shutdown procedure executes.

Usage requirements

You can specify Startup and Shutdown procedures for a state-aware or stateless AppServer. You cannot specify Startup and Shutdown procedures for a state-reset AppServer.

Note that both the Startup and Shutdown procedures are optional. There is no requirement to specify either or both of them. Additionally, the content of these procedures is completely application specific, and the AppServer architecture places no constraints on it. Any ABL logic that is valid within an AppServer agent is valid for these procedures.

Note: In Progress Version 8, you used the Connect procedure to program all startup activities on an AppServer agent. With the introduction of reusable and stateless servers much of the logic that you might have put into the Connect procedure for the Version 8 AppServer is now more appropriate for the Startup procedure. Similarly, much of the logic in a Version 8 Disconnect procedure might be more appropriate for the Shutdown procedure.

Connect and Disconnect procedures

Connect and Disconnect procedures encapsulate logic that executes whenever a client application establishes and terminates a connection with an AppServer. Functionality that might go in a Connect or Disconnect procedure includes authenticating client users, connecting or disconnecting client-specific databases, loading or storing the contents of client-specific temp-tables, or any other common actions required when each individual client connection begins or terminates.

Note: In state-free operating mode, these configuration procedures have no function.

Connect procedure

A Connect procedure executes in whatever AppServer agent handles the client connection request, before the connection request is accepted. If the Connect procedure completes with no error, the connection request from the client application is accepted. If the Connect procedure completes with an error return value (`RETURN ERROR` statement), the connection request is rejected.

The Connect procedure can also return a user-defined string to the client. This is accomplished by using either the `RETURNstring` statement or the `RETURN ERRORstring` statement in the connect procedure. The client can access this string if the Connect procedure executes and returns an error or if the Connect procedure succeeds. This feature allows you to return more detailed error messages or provide diagnostic information about a successful connection.

How the client can access the user-defined error depends on the client type:
You can specify the name of an AppServer Connect procedure using the OpenEdge Explorer or OpenEdge Management or by setting the `srvrConnectProc` property in the AppServer properties file (`ubroker.properties`). If you specify this procedure, it runs as part of the connection process initiated when an ABL client application executes the `CONNECT( )` method on a server handle or when an Open Client executes a corresponding connection method.

You can use this procedure to authenticate a client application’s request for a connection, or to perform any other initialization steps that you want executed each time a client application initiates an explicit connection request. For example, you might use it to store connection context using an application-specific context database or using the `SERVER-CONNECTION-CONTEXT` attribute on the `SESSION` handle (see Managing stateless connection context on page 66). Using the `SERVER-CONNECTION-ID` attribute, you can uniquely identify the connection. You can then gather the context identified with the connection and store it in the context database or marshall it into the `SERVER-CONNECTION-CONTEXT` attribute. Any AppServer procedure that executes on behalf of the connection can restore and resave this context as appropriate. For more information, see Activate and Deactivate procedures on page 61.

If you specify a Connect procedure, you must define it with three character input parameters. The following is an example of how you can define these input parameters:

```plaintext
DEFINE INPUT PARAMETER cUserId AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER cPassword AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER cAppServerInfo AS CHARACTER NO-UNDO.
```

When the AppServer Connect procedure runs, the values of these three parameters are the same as the corresponding parameters specified via the `CONNECT( )` method. If a value is not specified for a parameter to the `CONNECT( )` method, the AppServer Connect procedure receives the Unknown value (?) for that parameter.

OpenEdge does not examine the values passed from the `CONNECT( )` method to an AppServer Connect procedure. The name of each character input parameter might suggest the content you define for it. For example, the user-id can contain user name information, the password can contain password information, and the app-server-info can contain other user-defined information needed by the AppServer Connect procedure to complete a connection. However, the requirement to specify a particular value for the `CONNECT( )` method is based only on the implementation of your AppServer Connect procedure. After examining the values provided to the AppServer Connect procedure, you can reject the connection request from the client application by terminating the procedure with the `RETURN ERROR` statement.

### Disconnect procedure

A Disconnect procedure executes in whatever AppServer agent handles a client disconnection request, but before the connection is terminated. You can use the AppServer Disconnect procedure to specify logic that you want executed at this time. For example, you might want to use it to disconnect from a client-specific database or from other AppServers that were connected on behalf of the client.
You can specify the name of an AppServer Disconnect procedure using the OpenEdge Explorer or OpenEdge Management or by setting the `srvrDisconnProc` property in the AppServer properties file (`ubroker.properties`). This procedure takes no parameters.

For an AppServer running in state-reset or state-aware operating mode, the Disconnect procedure runs after the AppServer returns the response to the client disconnection request to the client application. For a stateless AppServer, the procedure runs just before the AppServer returns this disconnection response, where it is typically used to delete connection-specific context. Any raised conditions that you do not handle in this procedure are ignored by OpenEdge.

### Usage requirements

You can specify Connect and Disconnect procedures for all three AppServer operating modes (state-reset, state-aware, and stateless).

If the AppServer operating mode is state-reset or state-aware, the Connect procedure runs as a persistent procedure. If the AppServer operating mode is stateless, the Connect procedure runs as a non-persistent procedure. The Disconnect procedure runs as a non-persistent procedure in all operating modes.

If the Connect procedure is executed as a persistent procedure, and you do not need its context, you can delete it by executing the `DELETE PROCEDURE` (or `DELETE OBJECT`) statement on the procedure handle associated with the Connect procedure. If the AppServer operating mode is state-reset, the persistent Connect procedure is automatically deleted when a new connection is assigned to the AppServer agent.

Note that if the AppServer is running in stateless operating mode, there is no guarantee that the AppServer agent that runs the Connect procedure for the client application will be the same one that actually processes requests for the client application. Thus, any context that a Connect procedure establishes for a stateless connection must be stored in a file, database, `SERVER-CONNECTION-CONTEXT` attribute of the `SESSION` handle (see Managing stateless connection context on page 66), or some other resource that is accessible through ABL.

Note that both the Connect and Disconnect procedures are optional. There is no requirement to specify either or both of them. Additionally, the content of these procedures is completely application specific, and the AppServer architecture places no constraints on it. Any ABL logic that is valid within an AppServer agent is valid for these procedures.

### Activate and Deactivate procedures

Activate and Deactivate procedures encapsulate logic that executes automatically for certain client requests on an AppServer running in stateless and state-free operating modes. (For information on the stateless operating mode, see the sections on operating modes in AppServer and Client Interaction on page 23) When a client application sends remote procedure requests to a stateless AppServer, each request might execute in a different AppServer agent instance. To maintain application continuity between AppServer agents, you might need to establish some application-specific resources or context before and discard the resources and context after each request. The Activate and Deactivate procedures help to manage these resources and context more easily.

**Note:** Context management using the `SERVER-CONNECTION-CONTEXT`, `SERVER-CONNECTION-ID`, and `SERVER-CONNECTION-BOUND-REQUEST` attributes applies only to the stateless operating mode. In state-free operating mode, these attributes have no meaning.
Activate procedure

The Activate procedure executes immediately before a remote procedure request when the connection is in the unbound state. A typical use of Activate procedures is to retrieve connection context using an application-specific context database or the SERVER-CONNECTION-CONTEXT attribute on the SESSION handle (see Managing stateless connection context on page 66). Using the SERVER-CONNECTION-ID attribute, you can uniquely identify (key) the context in the context database. You can then retrieve the connection-identified context from the context database or unmarshall it from the SERVER-CONNECTION-CONTEXT attribute value. You can create the initial context for the connection using the Connect procedure. For more information, see Connect and Disconnect procedures on page 59.

In either state-free or stateless operating mode, you can also retrieve client connection context that might be keyed on the value of the ClientContextId property of the Progress.Lang.OERequestInfo class. From the Activate (or any other AppServer) procedure, you can access this property on the OERequestInfo object that is returned by the CURRENT-REQUEST-INFO attribute on the SESSION handle. For more information, see Managing state-free and stateless client context on page 69.

You can specify the name of an AppServer Activate procedure using the OpenEdge Explorer or OpenEdge Management or by setting the srvrActivateProc property in the AppServer properties file (ubroker.properties). If you specify this procedure, it accepts no parameters and runs as a non-persistent procedure on any AppServer agent that executes a remote procedure request for an unbound connection.

**Note:** The Activate procedure only runs prior to remote procedure requests. It does not run in response to connection or disconnection requests.

For the purposes of error handling, the Activate procedure is considered part of the remote procedure request. If the Activate procedure completes with no ABL termination condition (ERROR, QUIT, or STOP), the remote procedure executes immediately afterward. If the Activate procedure completes with a termination condition, it returns to the client application as if the remote procedure had generated the termination condition. Any return values (using the RETURN ERROR string statement) are passed to the client and the original remote procedure request is not executed.

Deactivate procedure

The Deactivate procedure executes immediately after remote procedure and delete procedure requests when the connection is in the unbound state. A typical use of the Deactivate procedures is to store connection context using an application-specific context database or using the SERVER-CONNECTION-CONTEXT attribute on the SESSION handle. For more information, see Managing stateless connection context on page 66. Using the SERVER-CONNECTION-ID attribute, you can uniquely identify (key) the context in the context database. You can then gather the context identified with the connection and store it in the context database or marshall it into the SERVER-CONNECTION-CONTEXT attribute.

In either state-free or stateless operating mode, you can also store connection context keyed on the value of the ClientContextId property of the Progress.Lang.OERequestInfo class. From the Activate (or any other AppServer) procedure, you can access this property on the OERequestInfo object that is returned by the CURRENT-REQUEST-INFO attribute on the SESSION handle. For more information, see Managing state-free and stateless client context on page 69.
You can specify the name of an AppServer Deactivate procedure using the OpenEdge Explorer or OpenEdge Management or by setting the `svrDeactivateProc` property in the AppServer properties file (`ubroker.properties`). If you specify this procedure, it accepts no parameters and runs as a non-persistent procedure after the request executes, but before any response is returned to the client. Also, the Deactivate procedure runs only if the connection is unbound when the request completes execution.

A stateless connection can be in the unbound or the bound state when a request is initiated and can change to the opposite state before the request completes execution. If the connection ends up in the bound state during execution of the request, the Deactivate procedure does not run when the request completes. However, if the connection ends up in the unbound state, the Deactivate procedure runs even if the original request completes with some ABL termination condition (ERROR, QUIT, or STOP).

After a delete procedure request, the Deactivate procedure runs only if the connection is unbound. The connection can become unbound only if both of the following are true:

- The deleted procedure was the last instantiated remote persistent procedure for the connection
- The `SERVER-CONNECTION-BOUND-REQUEST` attribute has a value of `FALSE`

For more information on how stateless connections transition between bound and unbound states, see Managing stateless connection context on page 66.

**Note:** The Deactivate procedure only runs following remote procedure and delete procedure requests. It does not run in response to new connection or disconnection requests.

For the purposes of error handling, only return values from the request (or any executed Activate procedure) are passed to the client. Any return values generated by the Deactivate procedure, itself, are ignored.

**Usage requirements**

As described in the previous sections, Activate and Deactivate procedures run only on behalf of connections in the unbound state. If a remote procedure request executes over a bound connection and the connection becomes unbound during execution of this procedure, the Deactivate procedure runs when the request completes with or without error.

Note that both the Activate and Deactivate procedures are optional. There is no requirement to specify either or both of them. Additionally, the content of these procedures is completely application specific, and the AppServer architecture places no constraints on it. Any ABL logic that is valid within an AppServer agent is valid for these procedures.

**Controlling AppServer entry points**

You can control AppServer entry points for an AppServer agent at run time using the `EXPORT( )` method on the `SESSION` handle. AppServer entry points are the pathnames of procedures in the AppServer `PROPATH` that a client application can execute as remote procedures (persistently or non-persistently). The `EXPORT( )` method establishes entry points by allowing you to set and maintain an `export list` that contains the pathnames of authorized remote procedures. As such, this is a mechanism that you can use to enforce secure access to the AppServer.
The scope of the export list set by the `EXPORT()` method is the AppServer agent in which the method executes. As a result, when and where you need to set the export list depends both on your AppServer operating mode and the requirements of your application. If you never set an export list using the `EXPORT()` method, client applications can call any ABL procedure on the AppServer `PROPATH` as a remote procedure.

This section describes how the AppServer uses an export list to limit client access to the AppServer and how you can use the `EXPORT()` method to set them. For more information on how the AppServer operating mode can affect export list settings, see the information on AppServer security in Design and Implementation Considerations on page 137.

### Export list operation

When an AppServer agent receives a remote procedure request from a client application, the requested procedure is first filtered by the export list. Only if the procedure is declared within this list can it be run as a remote procedure request. Otherwise, the remote procedure request is rejected and an error is returned to the client application.

### Setting and resetting the export list

You can invoke the `EXPORT()` method at any time, as many times as you need to, within an AppServer session. When you invoke the method, you have the option to pass it a comma-separated list of procedure names. Any format for a procedure name that is valid for the ABL `RUN` statement is also valid when you call the `EXPORT()` method. Each time you invoke the method, any procedure names that you specify are added to the current export list. If you invoke the method without a parameter (with no list of procedure names), the export list is reset to empty, as if the `EXPORT()` method were never called.

**Note:** If you do specify a procedure name list, the `EXPORT()` method does not validate the procedure names that you provide. Validation of a procedure name occurs only during the remote procedure request, after the name has been filtered by the export list.

To set export lists consistently for all client connections requires a different sequence of `EXPORT()` method settings depending on the operating mode of your AppServer. For more information on setting export lists for different AppServer operating modes, see Design and Implementation Considerations on page 137.

### Calling the `EXPORT()` method

There is no requirement to execute the `EXPORT()` method within an AppServer session. If the `EXPORT()` method is not called, or the export list is at any point reset to empty, then all remote procedure requests are accepted for execution.
Setting an initial export list

The following code example shows how you might invoke the `EXPORT( )` method from within an AppServer Connect procedure. In this example, two procedures, `order.p` and `stock.p`, are added to the export list, as shown:

```plaintext
DEFINE INPUT PARAMETER cUserId AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER cPassword AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER cAppServerInfo AS CHARACTER NO-UNDO.
DEFINE VARIABLE lReturn AS LOGICAL.
/* Authenticate user and call EXPORT appropriately for this user. */
ASSIGN lReturn = SESSION:EXPORT("order.p,stock.p").
```

When this code executes, it adds the two procedures to the export list. A remote procedure request to an AppServer agent that executes this AppServer Connect procedure executes itself only if the request is for `order.p` or `stock.p`. Otherwise, the remote procedure request returns an error.

Export list entries can also include names with an asterisk (*). The asterisk is a wildcard character to indicate that any group of characters is acceptable as a replacement for the asterisk.

For example, you can reference all procedures in a directory such as `stocksys/*.p`, some procedures in a directory such as `acctpay/qry*.p`, or all procedures in a library such as `order.pl<<*>`, as shown:

```plaintext
lReturn = SESSION:EXPORT("stocksys/*.p,acctpay/qry*.p,orders.pl<<*>")
```

After this code executes, all procedures located in the `stocksys` directory, some procedures located in the `acctpay` directory, and all procedures located in the orders procedure library are added to the export list.

If you create an export list, keep in mind that a client application is unaware that the procedures that it is submitting are being validated against this list. When the AppServer agent rejects a procedure for not being on this list, it returns an error to the client application.

Resetting the export list

At any time while servicing a connection, you can modify any existing export list by adding to the list or resetting the list. Thus, for example, if the list contains "`order.p,stock.p`", you can add "`payment.p`", as shown:

```plaintext
lReturn = SESSION:EXPORT("payment.p")
```

This, then, expands the list to "`order.p,stock.p,payment.p`".

If at any point, you want to completely reset the list, do the following:

```plaintext
lReturn = SESSION:EXPORT(). /* Reset to empty */
lReturn = SESSION:EXPORT("special-order.p,special-stock.p")
```
At this point, the AppServer agent only accepts remote procedure calls to `special-order.p` and `special-stock.p`.

### Resolving remote procedure names

If a procedure is on the export list, or if no export list is set, the AppServer agent determines which procedure to run using the `PROPATH` setting and standard procedure name resolution rules that apply to the `RUN` statement.

Also, the following issues can affect how procedure names are resolved:

- The directory in which an AppServer agent starts might affect how a procedure name is resolved because remote procedure requests and the `PROPATH` setting might contain unqualified (relative) pathnames.
- An AppServer agent runs in the same directory as the AppServer broker that starts it. Therefore, carefully consider what directory the AppServer broker starts in because procedure names might be resolved relative to this directory. For more information on setting up AppServer brokers, see *OpenEdge Application Server: Administration*.

For more information on the `PROPATH` environment variable, see *OpenEdge Application Server: Administration*. For more information on the standard name resolution rules, see the `RUN` statement entry in *OpenEdge Development: ABL Reference*.

### Managing stateless connection context

On a state-reset or state-aware AppServer, all remote procedures executed over the same connection run in the same AppServer agent. The state-aware AppServer session is dedicated to the client connection and its state at the initiation of a client request is exactly as it was when the previous request on the client connection finished.

Stateless AppServer sessions generally require more complex management than sessions on a state-reset or state-aware AppServer. The trade-off is potentially higher throughput for the effort. Because any AppServer agent in a stateless AppServer pool can handle any remote procedure request for any unbound connection, the context created by one remote procedure request is generally unavailable to the next request in the connection sequence. To help you manage context in a stateless environment, the AppServer provides the following mechanisms:

- `SERVER-CONNECTION-CONTEXT` attribute
- `SERVER-CONNECTION-ID` attribute
- `SERVER-CONNECTION-BOUND-REQUEST` and `SERVER-CONNECTION-BOUND` attributes

Other than in Using the `SERVER-CONNECTION-BOUND` attribute on page 68, most of the remaining information on context management applies only to AppServers running in stateless operating mode.

To support context management for both state-free and (unbound) stateless operating modes, OpenEdge provides the `ClientId` property on the `Progress.Lang.OERequestInfo` class. For more information, see Managing state-free and stateless client context on page 69.
Using the SERVER-CONNECTION-CONTEXT attribute

When a client application establishes a connection with an AppServer, the AppServer broker creates a connection context area in memory for the connection. An AppServer procedure can access this context area using the SERVER-CONNECTION-CONTEXT attribute of the AppServer SESSION handle. This is a readable and writable attribute of type CHARACTER. The initial value of SERVER-CONNECTION-CONTEXT is the Unknown value (?).

When any remote procedure executes, it can set the SERVER-CONNECTION-CONTEXT attribute to any application-specific value. Each time that OpenEdge assigns a client request to an AppServer agent, OpenEdge ensures that the SERVER-CONNECTION-CONTEXT attribute is set to the last value assigned in a previous AppServer agent that serviced the same client connection. This value is also available to any Connect procedure, Activate procedure, Deactivate procedure, or Disconnect procedure that you configure for the AppServer. Thus, each AppServer agent that services a client connection can pass context information to the next.

For more information on the Connect, Activate, Deactivate, and Disconnect procedures, see the Using AppServer configuration procedures on page 58.

Using the SERVER-CONNECTION-ID attribute

For more complicated context than you might want to store in the SERVER-CONNECTION-CONTEXT attribute, you might choose to use a context database. In this case, you can use the SERVER-CONNECTION-ID attribute as a primary key to help you store and retrieve the context for each client connection. A context database is an application database that is connected to and accessible from each AppServer agent on the AppServer.

Because the SERVER-CONNECTION-ID is a globally unique ID, OpenEdge ensures that no two connections on a computer network have the same value. Before a request is executed within an AppServer session, OpenEdge also ensures that the SERVER-CONNECTION-ID is set to the ID for the connection on which the request was received. This value is also available to any Connect procedure, Activate procedure, Deactivate procedure, or Disconnect procedure that you configure for the AppServer.

Using the SERVER-CONNECTION-BOUND-REQUEST attribute

As explained earlier (see AppServer and Client Interaction on page 23), a connection from client to a stateless AppServer is in one of two states:

• **Unbound** — Where remote procedure requests over the connection can be handled by any available AppServer agent running on the AppServer

• **Bound** — Where remote procedure requests over the connection are handled by a single AppServer agent dedicated to the connection

By default, a connection on a stateless AppServer is unbound.

Making a connection bound

An unbound connection can become bound in two ways:
• A client application can call a remote persistent procedure (see Programming ABL Client Applications on page 87) or instantiate a ProcObject, in the case of an Open Client or Web service client. This call runs and instantiates a persistent procedure in the AppServer agent that handles the remote procedure request. The connection thus becomes bound, dedicating this AppServer agent to handle all future client requests over the bound connection.

• An AppServer agent handling a request from a client can run a procedure that sets the SERVER-CONNECTION-BOUND-REQUEST attribute on the SESSION handle to TRUE. The connection thus becomes bound, dedicating the AppServer agent that sets this attribute to handle all future client requests over the bound connection.

Note that it is the successful instantiation of a remote persistent procedure that forces the connection to transition from the unbound state to the bound state. If a client fails in its attempt to instantiate a remote persistent procedure on an unbound connection, the connection remains unbound.

Making a connection unbound

You can request that a connection become unbound by setting the SERVER-CONNECTION-BOUND-REQUEST attribute on the SESSION handle to FALSE. However, the connection only becomes unbound as long as there are no remote persistent procedures still active in the AppServer agent.

Once a connection transitions from the bound to the unbound state, the AppServer agent can handle the next request sent by any connected client. Conversely, the next request sent by the client on the previously bound connection can be handled by any AppServer agent.

If remote persistent procedures are active on a bound connection, setting the SERVER-CONNECTION-BOUND-REQUEST attribute to FALSE has no effect until all remote persistent procedures in the AppServer agent are explicitly deleted. When all remote persistent procedures are deleted, the connection either remains in the bound state or becomes unbound depending on the last setting of the SERVER-CONNECTION-BOUND-REQUEST attribute.

Using the SERVER-CONNECTION-BOUND attribute

A stateless connection can be in a bound state because of active remote persistent procedures even if the SERVER-CONNECTION-BOUND-REQUEST attribute is set to FALSE. To easily determine if a connection is bound, even if the SERVER-CONNECTION-BOUND-REQUEST attribute is set to FALSE, check the value of the SERVER-CONNECTION-BOUND attribute. When set to TRUE for an AppServer agent running on a stateless AppServer, the read-only SERVER-CONNECTION-BOUND attribute indicates that the connection is currently bound, either from having active remote persistent procedures or from having the SERVER-CONNECTION-BOUND-REQUEST attribute set to TRUE.

Note: If the AppServer is running in state-aware or state-reset operating mode, the SERVER-CONNECTION-BOUND attribute for each of its AppServer agents is always set to TRUE.
Managing state-free and stateless client context

In the state-free or the (especially unbound) stateless operating modes of the AppServer, any available AppServer agent handles each remote request from an AppServer client. The agent can use a client context identifier provided by OpenEdge as a key to retrieve a previously initialized client context from a context store in order to execute the request on behalf of the originating client. On completion of the request, the agent can then update the client context in the context store for access by the client when the request returns, or pass the client context identifier to any other AppServer that might be sent a request from the agent for the same request. Agents across one or more AppServers can thus use this same client context identifier to access the context store on behalf of a given client.

OpenEdge generates an initial, globally unique value for this client context identifier when it first creates the server object that a client uses to connect to an AppServer. The client can then assign its own value for this identifier if necessary, possibly one that has been previously initialized for another AppServer connection.

For example, this identifier is typically used to identify the current user login session that a client initially establishes through an AppServer that implements the authentication service for the client. This value is then propagated in subsequent requests sent between the client and any AppServer involved in requests for that same login session.

Passing the client context identifier between client and server

OpenEdge provides a class, Progress.Lang.OERequestInfo, and its properties to hold the value of the client context identifier and related context information. OpenEdge provides instances of OERequestInfo for reference by attributes on both the server object handle that an ABL client uses to make remote requests to an AppServer, and on the SESSION system handle on the AppServer agent that processes the request. These attribute references include:

- On a client's server object handle (or a returned asynchronous request object handle):
  - REQUEST-INFO — Referencing context information that is sent with a client request to the AppServer associated with this handle
  - RESPONSE-INFO — Referencing context information that is returned to the client with the response from the most recent request sent to the AppServer associated with this handle

- On the AppServer agent's SESSION handle:
  - CURRENT-REQUEST-INFO — Referencing context information that is received from the current client request to the AppServer. This is the same information that the client sends from the REQUEST-INFO attribute on the server object handle associated with the request.
  - CURRENT-RESPONSE-INFO — Referencing context information that is returned from the AppServer in response to the current client request. This is the same information that the client receives in the RESPONSE-INFO attribute on the server object handle associated with the request.
The `ClientContextId` property on an `OERequestInfo` object holds the client context identifier value that is generated and passed between a client and AppServer. Additional properties identify other information about the request, including the remote procedure call (`ProcedureName`), a unique ID that OpenEdge generates for each request (`RequestId`), and OpenEdge release information (`VersionInfo`) about the AppServer client that initiates the request or the AppServer agent that returns the response, depending on the attribute that provides the information.

As noted previously, the `ClientContextId` property value typically serves as a primary key to the client context information maintained in a context store for a user login session, which might be initialized by an authentication service that handles the user login. Subsequently, when the client executes a remote procedure on the AppServer, it again propagates the client context identifier (and related information) from the `REQUEST-INFO` attribute on the server handle, and the AppServer agent can then retrieve the identifier from its `SESSION` handle using the `CURRENT-REQUEST-INFO` attribute. Before the remote procedure returns, it can modify the context identifier referenced by the `CURRENT-RESPONSE-INFO` attribute on the `SESSION` handle, and the client can retrieve the updated identifier returned using the `RESPONSE-INFO` attribute on the server handle (or on the returned asynchronous request object handle for an asynchronous request).

For more information on these attributes and examples that access the `ClientContextId` property, see the reference entries for these attributes in *OpenEdge Development: ABL Reference*. For more information on using the `ProcedureName` property, see *Accessing the name of the current remote procedure* on page 56.

### Generating a client context identifier

The client context identifier is typically a globally unique character string. For performance, the maximum length of the string should be less than 36 characters. The client context identifier can also be a blank (""") value or the `Unknown` value (?).

The following code fragment shows a recommended ABL assignment statement for generating a unique value for the client context identifier:

```abl
DEFINE VARIABLE ccid AS CHARACTER NO-UNDO.
ccid = SUBSTRING(BASE64-ENCODE (GENERATE-UUID), 1, 22).
```

This is only one of the methods you can use to generate a client context identifier, but it does generate a globally unique value.

The default method of generating the client context identifier is to allow the AppServer clients (ABL, Java, and .NET) to automatically generate the value and send it to the AppServer. Optionally, the AppServer client or the AppServer agent can create their own client context identifier for advanced application logic.
For example, the client context identifier in an ABL session needs to be reset in order to indicate the end of the client user login session. This is typically done when the server invalidates the current client login session by invoking the LOGOUT() method on the client-principal object that maintains the session, and by removing the client login context from the context store. The server then typically sets the ClientContextId property on the appropriate Progress.Lang.OERequestInfo class instance (available through the CURRENT-RESPONSE-INFO attribute on the SESSION handle) to the Unknown value (?). This tells the client that its current login session has been terminated, and any subsequent (presumably invalid) requests from the client to the same AppServer connection will contain a client context identifier with the Unknown value (?). For more information on client-principal objects and using the ClientContextId property to manage context for user login sessions, see Implementing multi-tier security models on page 72 in this manual and the sections on authenticating the user identity for multi-tier applications in OpenEdge Development: Programming Interfaces.

Managing context from the AppServer with OERequestInfo objects

The entire application can manage the value of the client context identifier in the ClientContextId property. For example, an AppServer agent might control the client context identifier for a client using the OERequestInfo instance returned to the client in the AppServer response, or act as a client to another AppServer using the OERequestInfo instance sent to that other AppServer in its own remote request.

In addition to the client context identifier, instances of OERequestInfo provide additional context information that depends on whether the AppServer client or agent is originating the instance.

OpenEdge context information accessible by an AppServer agent provides:

- **OpenEdge version information for the current AppServer agent** — The agent session can access the LOCAL-VERSION-INFO attribute on its SESSION system handle to find its OpenEdge Release information. This object reference attribute returns a Progress.Lang.OEVersionInfo class instance with OEMajorVersion, OEMinorVersion, and OEMaintVersion properties that provide this information. This Release information is also returned in the response to each client request.

- **OpenEdge AppServer version information returned to a client** — This identifies the OpenEdge Release information for the AppServer returned to the client from the AppServer agent in response to the current client request. This is the same information provided to the agent using its own LOCAL-VERSION-INFO attribute. The agent provides this information to the client using the VersionInfo property on the OERequestInfo instance referenced by the CURRENT-RESPONSE-INFO attribute on the agent's SESSION handle. The client can then access this information using the RESPONSE-INFO attribute on the server object handle it uses to make a remote request.

- **OpenEdge version information of the client application** — This identifies the OpenEdge Release of the client application that executed the current request. The agent can read this information using the VersionInfo property on the OERequestInfo instance referenced by the CURRENT-REQUEST-INFO attribute on the agent's SESSION handle.

- **Client context identifier for the current request** — This is the value of the ClientContextId property on the OERequestInfo instance referenced by the server handle's REQUEST-INFO attribute when the client makes a remote request. The AppServer agent can retrieve this property value on the OERequestInfo instance referenced by the CURRENT-REQUEST-INFO attribute on its SESSION handle.
• **Identifier for the current request only** — The client automatically generates a globally unique identifier for the current request in the RequestId property on the OERequestInfo instance referenced by the server handle's REQUEST-INFO attribute. The AppServer agent can read this property value on the OERequestInfo instance referenced by its SESSION handle's CURRENT-REQUEST-INFO or CURRENT-RESPONSE-INFO attribute. This value is read-only and cannot be changed by any application code.

• **Client context identifier in response to the current request** — Before completing its current request, the AppServer can set a different client context identifier value as a part of its response to the client by setting the ClientContextId property available through the CURRENT-RESPONSE-INFO attribute of its SESSION handle. Otherwise, by default, the existing property value available through its CURRENT-REQUEST-INFO attribute is used. The client can read this value after the remote request completes using the RESPONSE-INFO attribute on the server handle.

For more information on these class properties and handle attributes, see *OpenEdge Development: ABL Reference*.

For state-free logical connections (bindings), the typical way to manage context for a client session is for the application service to rely on a context database. In state-free bindings, a client request can be executed, not only on any agent of an AppServer, but on any one of many AppServers that support the application service for a single user login session. You can also use this same configuration for an AppServer running in the stateless operating mode.

Because none of the context management attributes provided for stateless operating mode work in state-free mode (except for the SERVER-CONNECTION-BOUND attribute), the context database must:

• Be shared by all AppServer instances that support the application service

• Use an entirely application-defined means (such as the ClientContextId property) to identify and distinguish client sessions that use the application service and to key the context database for this purpose

Together with this context database, you can use the Activate and Deactivate configuration procedures to restore context for every client request or provide the necessary context database management solely within the context of remote requests.

For more information on managing context from the ABL client, see *Accessing client context regardless of session model* on page 105.

---

**Implementing multi-tier security models**

An AppServer agent can act as a client of another AppServer to manage security on behalf of an originating client in a multi-tier distributed application. In addition to accessing client context as described in the previous section (see *Managing context from the AppServer with OERequestInfo objects* on page 71), this “agent as client” can also execute remote procedure calls using the following security models:

• **Delegation** — The agent establishes its application context in a remote login procedure by propagating the client context identifier from the originating client to the AppServer (authentication server).

• **Impersonation** — The agent executes remote procedure calls in the context of the originating client by propagating that client’s context identifier with each remote procedure call to another AppServer, thus allowing the AppServer to establish the proper client context for the call.
• **Single sign-on (SSO)** — The agent sets the client context identifier of a server handle for an AppServer B based on the context established by executing the remote login procedure on the server handle for an AppServer A (the authentication server). When the agent connects to AppServer B, the AppServer’s Connect procedure then sets the session identity in an SSO operation using a sealed client-principal returned from a context store keyed on the client context identifier that was propagated in the prior call to the remote login procedure on AppServer A.

• **WebSpeed agent** — This replaces the AppServer agent along with a Web browser serving as the originating client and the WebSpeed application possibly calling out to a separate authentication server (AppServer). In response to a browser login request, the Web server propagates the client context identifier from the authentication server to the browser using cookies or by embedding the context identifier in resource locators managed by the WebSpeed application. The browser then returns the same client context identifier with each subsequent request to the WebSpeed application in the same user login session.

For more information on client-principal objects and implementing basic security for multi-tier applications, see the sections on authentication in *OpenEdge Development: Programming Interfaces*.

---

**Understanding remote procedure handles**

The **THIS-PROCEDURE** system handle returns the current procedure handle value from within any executing block of an external procedure. The **THIS-PROCEDURE** handle created within an AppServer session for a remote procedure request is a **remote procedure handle**. Its **REMOTE** attribute is, therefore, set to **TRUE**. If the **REMOTE** attribute is **FALSE**, then the current procedure exists as the result of a **local procedure** call (that is, a procedure call initiated and executed within the current ABL session context).

**Remote procedure handles**

When an ABL client application executes a remote procedure, two procedure handles are created: one within the client application session and another separate handle within the AppServer session where the procedure is created. OpenEdge internally maintains a mapping between the two handles. The handle within the client application is a **proxy procedure handle**, and its **PROXY** attribute is set to **TRUE**. The handle’s **PERSISTENT**, **SINGLE-RUN**, or **SINGLETON** attribute is set to **TRUE** for a persistent, single-run, or singleton procedure, respectively. The corresponding handle within the AppServer agent is a **remote procedure handle**, and its **REMOTE** attribute is set to **TRUE**. As with the proxy handle, the **PERSISTENT**, **SINGLE-RUN**, or **SINGLETON** attribute is set to **TRUE** for a persistent, single-run, or singleton procedure, respectively.

Unlike the values of procedure handles and the **THIS-PROCEDURE** handle that reference the same local procedure context, the proxy procedure handle and the remote procedure handle are truly separate handles. For example, setting the **PRIVATE-DATA** attribute on a remote procedure handle has no effect on the **PRIVATE-DATA** attribute of the corresponding proxy procedure handle in the client application.

For more information on the relationship between remote and proxy procedure handles, see the information on procedure handles in *Design and Implementation Considerations* on page 137.
Deleting remote procedures

As with local persistent procedures, a remote persistent procedure context remains active within an AppServer session until it is deleted using the `DELETE OBJECT` or `DELETE PROCEDURE` statement. You can thus delete a remote persistent procedure in the AppServer session by deleting its remote procedure handle. If the delete occurs in the context of a remote procedure request, the deletion is pending until the request completes and returns to the client. When the remote persistent procedure is finally deleted, both its proxy procedure handle on the client and its remote procedure handle in the AppServer agent are deleted together.

In the case of a remote single-run or singleton procedure, `DELETE PROCEDURE` only deletes the proxy handle on the client. This is because a single-run procedure is deleted automatically by the AVM, and a singleton procedure can be running on more than one AppServer agent, where it can be deleted using ABL code running on the agent.

Handling conditions and returning values

Conditions raised in an AppServer session have varied effects, depending on the condition. You can also return values to the client application from a remote procedure, with or without raising an `ERROR` condition, or throw error objects from an AppServer to an ABL client. The following sections describe how each of these standard ABL mechanisms can affect an AppServer session and any client remote procedure request.

Raising the ERROR condition and returning values

You can return application exceptions, with or without associated values, to client applications by executing the `RETURN ERROR` statement. The effect is similar to a local procedure return. When you execute this statement in any remote procedure (persistent or non-persistent), the procedure:

1. Terminates
2. Raises the `ERROR` condition on the `RUN` statement in the client (or an equivalent exception in an Open Client)
3. Returns any specified value to the client application

An `ERROR` condition raised in an Activate procedure for a stateless AppServer session has the same effect as if it were raised in the remote procedure for the request.

To return a value without raising the `ERROR` condition, use the `RETURN` statement without the `ERROR` option.

**Note:** The `LONGCHAR` and `MEMPTR` data types cannot be returned from a user-defined function across an AppServer.
Throwing error objects to the client

Error objects can be thrown from an AppServer to an ABL client, and those errors are handled in the same way as errors thrown from a locally called procedure. The objects themselves are serialized when thrown from the AppServer and then reconstructed on the client, and they are subject to the same rules for serialization as all other class-based objects passed between an AppServer and ABL Client. In particular, any user-defined class (such as a subclass derived from Progress.Lang.AppError) must be defined as SERIALIZABLE. See Passing class-based objects as parameters between an AppServer and an ABL client on page 112 for more information.

For more information on structured error handling and error objects, see OpenEdge Development: Error Handling.

Raising the ENDKEY condition

Like an ABL client, an AppServer session raises the ENDKEY condition if an attempt to find a database record fails or an INPUT FROM statement reaches the end of an input stream. Like a batch client, there are no user-interface events that can raise the ENDKEY condition.

Handling the STOP condition

If a client application raises the STOP condition while a remote procedure is active on a connected AppServer, OpenEdge also raises the STOP condition in the context of the AppServer agent handling the remote procedure request.

Note: A client can raise the STOP condition in the context of an executing remote procedure by invoking the STOP( ) method on the server object handle.

A STOP condition raised within a remote procedure does not necessarily terminate procedure execution, and it never causes the AppServer session to restart (as in an ABL client session). You can use the ON STOP phrase within the remote procedure to intercept any STOP condition raised during execution and handle it as your application requires. Otherwise, the STOP condition propagates to the client application. For information on unhandled STOP conditions, see Effects of unhandled conditions on page 77.

A STOP condition raised in an Activate procedure for a stateless AppServer session has the same effect as if it were raised in the remote procedure for the request.

Raising a timed stop

The STOP-AFTER phrase on DO, FOR, or REPEAT statement establishes a timer on the block. If the timer expires before execution of the block completes, the AVM raises the STOP condition. At this point, normal STOP condition processing occurs and you can use the ON STOP phrase to handle the STOP.

The rules for executing synchronous remote procedure calls within a timed code block are essentially the same as those for local procedures. Remote procedures may be called within timed blocks or from blocks within timed blocks.
When a timed remote procedure call is executed on an OpenEdge AppServer, the time limit for the procedure is enforced in the client AVM. If the timeout is exceeded, the STOP condition is raised by the client. In this context, a STOP message is sent to the AppServer AVM after the specified time limit had exceeded, which is similar to what happens when an interactive user presses the STOP key.

The following examples illustrate making timed synchronous remote procedure calls on an AppServer:

```c
/* clntNested3.p */
DEFINE VARIABLE clntTimeLimit AS INTEGER NO-UNDO INITIAL 5.
DEFINE VARIABLE hSrv AS HANDLE NO-UNDO.
DEFINE VARIABLE cParm AS CHARACTER NO-UNDO.

CREATE SERVER hSrv.
cParm = FILL("X", 10). /* 10 "X" characters will be sent to server */
hSrv:CONNECT("-URL AppServerDC://hostName/svc").
DO STOP-AFTER clntTimeLimit ON STOP UNDO, LEAVE:
    RUN foo.
END.

PROCEDURE foo:
    DEFINE VARIABLE cnt AS INTEGER NO-UNDO.
    FOR EACH Customer NO-LOCK:
        cnt = cnt + 1.
    END.
    cParm = FILL("X", cnt).
    RUN srvrNested3.p ON SERVER hSrv (INPUT-OUTPUT cParm).
END PROCEDURE.
```

In this example, the client program (clntNested3.p) connects to the AppServer, and runs the local procedure foo within a timed code block, specifying STOP-AFTER 5. The procedure foo counts the number of customer records, and then fills the character variable cParm with "X" characters. The remote procedure srvrNested3.p is then called. Because this remote procedure call is made from within the timed procedure foo, srvrNested3.p is implicitly run as a timed procedure; the time limit for this call is the time remaining for the timed block calling foo (5 seconds, less the time required to count the number of customer records). The client AVM begins timing this when the remote procedure call is made. If the call to srvrNested3.p does not return in the client AVM within the 5 second time limit, the client AVM will issue a STOP message to the AppServer. When the AppServer receives the STOP message, the STOP condition will be raised in the AppServer AVM, as shown:

```c
/* srvrNested3.p */
DEFINE INPUT-OUTPUT PARAMETER p1 AS CHARACTER NO-UNDO.

DEFINE VARIABLE srvrTimeLimit AS INTEGER NO-UNDO INITIAL 10.
DEFINE VARIABLE spinLimit AS INT64 NO-UNDO INITIAL 15000.
p1 = FILL("Y", 30). /* 30 "Y" characters will be sent to the client */
DO STOP-AFTER srvrTimeLimit:
    RUN spinHere (spinLimit).
END.

RETURN "30 Ys".
PROCEDURE spinHere:
    DEFINE INPUT PARAMETER spinLimit AS INT64 NO-UNDO.
    DEFINE VARIABLE loopFlag AS LOGICAL NO-UNDO.
```
DEFINE VARIABLE endTime AS INT64 NO-UNDO.

ASSIGN
    loopFlag = TRUE
    endTime = ETIME(FALSE) + spinLimit.

DO WHILE loopFlag:
    IF (ETIME(FALSE) > endTime) THEN
        loopFlag = FALSE.
    END.
END.
END PROCEDURE.

In srvrNested3.p on the AppServer agent, the INPUT-OUTPUT parameter p1 is filled with 30 "Y" characters, and the spinHere procedure is called from within a timed code block, with the time limit of 10 seconds (the value of srvrTimeLimit). In this example, spinHere is supposed to return after 15 seconds (the value of spinLimit). The procedure will ostensibly time out after 10 seconds, and the STOP condition will be raised. However, the client AVM will issue the STOP message to the server after 5 seconds, the STOP condition will be raised in the client after 5 seconds (approximately).

Because srvrNested3.p employs default STOP handling (that is, the srvrNested3.p program does not specify an ON STOP phrase), the STOP condition is not handled by the server AVM. The timeout is consequently propagated back to the client AVM.

It is important to note that the remote procedure call could also be made within an explicit timed block. As with the local case, the actual time limit value used by the RUN would be the smaller of the implicit and the explicit time limit values.

Note: Another way to time limit procedure calls is to set the ubroker property called srvrExecutionTimeLimit will in the [UBroker] section of the ubroker.properties file. This is a non-negative integer property that specifies the maximum time in seconds that a remote procedure may execute on the given AppServer. This timeout value applies to all remote procedures that execute on the AppServer.

Handling the QUIT condition

You can use the ON QUIT statement within a remote procedure to intercept any QUIT condition raised during execution and handle it as your application requires. Otherwise, the QUIT condition propagates to the client application. For information on unhandled QUIT conditions, see Effects of unhandled conditions on page 77.

A QUIT condition raised in an Activate procedure for a stateless AppServer session has the same effect as if it were raised in the remote procedure for the request.

Effects of unhandled conditions

An unhandled ERROR or ENDKEY condition raised in an AppServer agent has no effect on the client application to which it is connected. The handling of each condition conforms to standard ABL rules. For more information about these rules, see OpenEdge Getting Started: ABL Essentials.

Two ABL conditions (STOP and QUIT), if not handled using the ON STOP and ON QUIT statements in an AppServer session, have definite effects on any client application for which the AppServer agent is handling a request.
Any unhandled \texttt{STOP} causes the remote procedure request (persistent or non-persistent) to terminate and propagate the \texttt{STOP} condition to the client application.

Any unhandled \texttt{QUIT} condition causes the remote procedure request to terminate and return successfully without any propagated condition to the client. However, OpenEdge also terminates the client connection to the AppServer.

**Managing transactions**

You can manage transactions in an AppServer session much the same way that you can manage them in an ABL client session, using transaction blocks and statements that update tables in a database. However, because client and AppServer sessions are entirely separate, including their transaction contexts, the client normally has no direct ability to influence the course of an AppServer transaction, which begins and ends within the scope of a single remote procedure request. To allow finer-grained control of transactions running on the AppServer, OpenEdge provides an additional transaction type that allows a transaction on the AppServer to span multiple client requests.

**Types of AppServer transactions**

An AppServer can allow a client to invoke two different types of transactions in an AppServer agent. The two types of transaction are distinguished by scope (duration) and include:

- The normal ABL transaction
- The automatic transaction

**Normal ABL transactions**

A normal ABL transaction has the scope of part or all of a single remote procedure call. That is, the transaction begins and ends during execution of a single remote procedure request. Thus, all control of the transaction is handled by the AppServer agent within a single request.

Normal ABL transactions take full advantage of ABL built-in transaction management features, such as ABL-integrated retry and undo (rollback) mechanisms. Normal ABL transactions yield the smallest possible transaction unit, providing the most efficient handling of record conflicts and transaction resolution. For most applications, this is the only type of transaction that you need to use. It is also the only type of transaction available to previous versions of the AppServer. For more information on managing normal ABL transactions on an AppServer, see the Implementing normal ABL transactions on page 79.

**Automatic transactions**

An \textit{automatic transaction} allows the transaction to span more than one remote procedure call. That is, once an automatic transaction is started through some remote procedure call, you can explicitly terminate it in the same procedure call or in a subsequent procedure call running in the same AppServer session. To control this type of transaction, the AppServer provides attributes and methods that you can use to specify how and when you want the transaction to end.
With automatic transactions, transactions tend to grow large, with greater potential for lock conflicts and other concurrency problems. (For more information on small and large transactions, see OpenEdge Getting Started: ABL Essentials.) Even so, automatic transactions can be extremely useful where the application requires the transaction to span multiple AppServer requests. This is especially true where client response is entirely programmatic and does not depend on user interaction for a transaction to complete.

An example of a useful application might be where a transaction commit depends on a real-time status from a process control or financial trading system. Depending on this status, the AppServer might require additional information from the client to resolve the transaction. As long as the client does not have to wait to make this information available to the AppServer, the impact of the automatic transaction on system performance can be minimal. For more information on managing automatic transactions, see Implementing automatic transactions on page 79.

Implementing normal ABL transactions

You can implement a normal ABL transaction in any AppServer procedure that the client runs either directly or indirectly. The ABL for starting a normal ABL transaction in an AppServer session is identical to ABL required to start transactions in an ABL client session. In any case, a normal ABL transaction starts and ends entirely within the context of the AppServer procedure that implements it. Because this type of transaction completes within the scope of a single procedure call, you can implement a normal ABL transaction on an AppServer running in any operating mode (state-reset, state-aware, or stateless).

Implementing automatic transactions

To implement an automatic transaction, you provide a specially-coded remote procedure (transaction initiating procedure) that initializes a transaction object in the AppServer agent where it executes. This transaction object is a permanent object in the session context that provides methods and attributes on a transaction object handle that allow the AppServer session to commit, rollback, or set and check the status of an automatic transaction. Because an automatic transaction terminates when the context of the transaction initiating procedure is deleted, you must execute the transaction initiating procedure persistently to make practical use of the transaction that it initiates.

Initializing the automatic transaction object

To initialize the transaction object, you must create a transaction initiating procedure by specifying the TRANSACTION-MODE statement as the first executable statement in the procedure file. For example:

```
TRANSACTION-MODE AUTOMATIC [ CHAINED ]
```

When the client calls this transaction initiating procedure as a remote persistent procedure, an automatic transaction starts in the context of the AppServer session and can remain open after the remote procedure call returns to the client. The context for this open automatic transaction is the entire AppServer session, not (as you might think) the context internal to the transaction initiating procedure instantiated in the AppServer session. Thus, any remote procedure (persistent or non-persistent) that is subsequently executed participates in this open automatic transaction.
Controlling the transaction

You can access the transaction object handle for the initialized transaction object using the TRANSACTION attribute on the procedure of any procedure executing in the AppServer session by using the THIS-PROCEDURE system handle, as shown:

```
DEFINE VARIABLE hTran AS HANDLE NO-UNDO.
hTran = THIS-PROCEDURE:TRANSACTION.
```

You can then access the transaction object methods and attributes to control the transaction, as described in the following table.

**Table 4: Transaction handle methods and attributes**

<table>
<thead>
<tr>
<th>Method/Attribute</th>
<th>Return/Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT-COMMIT</td>
<td>LOGICAL</td>
<td>A writable attribute whose setting tells the transaction object how to complete any open automatic transaction if the transaction terminates with no prior invocation of the SET-COMMIT( ) or SET-ROLLBACK( ) method (for example, if the transaction initiating procedure is deleted). Setting it to TRUE ensures that the transaction is committed. Setting it to FALSE ensures that the transaction is rolled back. The default value is FALSE.</td>
</tr>
<tr>
<td>IS-OPEN</td>
<td>LOGICAL</td>
<td>A read-only attribute that returns TRUE if a transaction is active. The function of this attribute is identical to the ABL TRANSACTION function. See also the TRANS-INIT-PROCEDURE attribute.</td>
</tr>
<tr>
<td>SET-COMMIT( )</td>
<td>LOGICAL</td>
<td>Tells the transaction object to commit any active automatic transaction after the current request returns execution to the client. If no automatic transaction is active, the method returns FALSE. Note that you cannot invoke this method after invoking a prior SET-ROLLBACK( ) within the same client request.</td>
</tr>
</tbody>
</table>
Managing transactions

<table>
<thead>
<tr>
<th>Method/Attribute</th>
<th>Return/Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET-ROLLBACK( )</td>
<td>LOGICAL</td>
<td>Tells the transaction object to roll back any active automatic transaction after the current request returns execution to the client. If no automatic transaction is active, the method returns FALSE. Note that you can invoke this method after invoking a prior SET-COMMIT( ) within the same client request.</td>
</tr>
<tr>
<td>TRANS-INIT-PROCEDURE</td>
<td>HANDLE</td>
<td>A read-only attribute that returns the procedure handle to the transaction initiating procedure for any active automatic transaction. If no automatic transaction is active this attribute returns an invalid handle. (Validate with the VALID-HANDLE function.)</td>
</tr>
</tbody>
</table>

**Terminating automatic transactions**

An automatic transaction remains active as long as the context of the transaction initiating procedure remains active and you do not otherwise terminate the transaction using a transaction handle method. Thus, you can terminate an automatic transaction using the following two techniques:

- **Explicit termination** — You can explicitly terminate an automatic transaction on the AppServer with a reference to the SET-COMMIT( ) method or the SET-ROLLBACK( ) method, as shown:

  ```
  hTran = THIS-PROCEDURE:TRANSACTION.
  hTran:SET-COMMIT().
  ```

  ```
  hTran = THIS-PROCEDURE:TRANSACTION.
  hTran:SET-ROLLBACK().
  ```

As long as an automatic transaction is open, you can execute any internal procedure of the transaction initiating procedure from any other procedure running in the AppServer session. However, if no transaction is open, any such attempt to call these internal procedures from within the AppServer agent context returns an error. Without an open transaction, only a client application can call a remote internal procedure of the transaction initiating procedure (see **Restarting automatic transactions** on page 82 for more information).

**Note:** You can determine whether an automatic transaction is open in an AppServer session by either checking for an error after executing an internal procedure of the transaction initiating procedure or by checking the value of the TRANS-INIT-PROCEDURE attribute.

- **Implicit termination** — You can also terminate the automatic transaction from either the client or the AppServer session by deleting the proxy procedure handle (on an ABL client) or remote persistent procedure handle (on the AppServer) of the transaction initiating procedure. If you
delete the transaction initiating procedure, the transaction commits or rolls back depending on the value of the transaction handle DEFAULT-COMMIT attribute.

Restarting automatic transactions

After an automatic transaction terminates, how an automatic transaction can restart depends on how it terminates. If the transaction terminates by deleting the transaction initiating procedure, only a client application can restart an automatic transaction by again remotely instantiating a transaction initiating procedure.

If a transaction initiating procedure is still active in the AppServer session, you can support the restarting of an automatic transaction using one of the following techniques, depending on how you code the transaction initiating procedure:

- **Automatic (chained) restarting** — If you specify the CHAINED option for the TRANSACTION-MODE AUTOMATIC statement, a new automatic transaction starts immediately after the previous automatic transaction commits or rolls back. Thus, with the CHAINED option an automatic transaction is always open in the AppServer session until the transaction initiating procedure, itself, is made inactive (is deleted).

- **Manual restarting** — If you do not specify the CHAINED option for the TRANSACTION-MODE AUTOMATIC statement, the client application can cause another automatic transaction to restart by calling any internal procedure of the active transaction initiating procedure. Thus, a remote call to an empty internal procedure can start a new automatic transaction.

Caution: In general, try to keep automatic transactions on an AppServer as short as possible. The multi-request feature of automatic transactions tend to encourage long transactions, and long transactions can make large or important segments of a database inaccessible to other ABL sessions for an indefinite period. This can create database deadlocks and traffic jams that can seriously retard the performance of multiple sessions. For more information on techniques to shorten automatic transactions, see the information on transaction management considerations in Design and Implementation Considerations on page 137.

Automatic transaction example

By including appropriate calls to internal procedures of the transaction initiating procedure in your remote procedures, you can encapsulate the management of a multi-request automatic transaction in a manner completely hidden from the client. However, the following example is relatively simple in that the client calls all of the internal procedures of the transaction initiating procedure directly, thus more openly exposing the transaction mechanism to the client.

The following two procedures include a transaction initiating procedure and an ABL client procedure that accesses it.

The transaction initiating procedure (a-txtest.p) implements an unchained automatic transaction. This procedure provides three internal procedures that allow the client to explicitly start (startme), commit (stopme), and roll back (abortme) transactions.
a-txtest.p

```abl
/* This statement makes this procedure a transaction-initiating procedure */
TRANSACTION-MODE AUTOMATIC.

PROCEDURE startme: /* This empty internal procedure starts a transaction */
END.

PROCEDURE stopme: /* This internal proc arranges for a commit */
  DEFINE VARIABLE hTrans AS HANDLE.
  hTrans = THIS-PROCEDURE:TRANSACTION.
  hTrans:SET-COMMIT().
END.

PROCEDURE abortme: /* This internal proc arranges for a rollback */
  DEFINE VARIABLE hTrans AS HANDLE.
  hTrans = THIS-PROCEDURE:TRANSACTION.
  hTrans:SET-ROLLBACK().
END.
```

The client procedure (a-txclnt.p) establishes the initial transaction object by instantiating a-txtest.p on the AppServer. After calling two database update procedures, the client commits the transaction with a call to stopme. It then begins a new transaction by calling startme, and after calling the same two database update procedures, rolls back the transaction by calling abortme.

As noted in the comments a-txtest.p establishes an unchained transaction object. If the procedure instead establishes a chained transaction object, each time the current transaction terminates (stopme, abortme), the AppServer agent automatically starts a new transaction with no need for the client to call a procedure (startme) to initiate it.

a-txclnt.p

```abl
DEFINE VARIABLE ph AS HANDLE.

/* Create a transaction initiating procedure and start the transaction */
RUN a-txtest.p ON SERVER h TRANSACTION DISTINCT PERSISTENT SET ph.
RUN custupdate.p ON SERVER h TRANSACTION DISTINCT.
RUN orderupdate.p ON SERVER h TRANSACTION DISTINCT.

/* Causes transaction to commit including both custupdate and orderupdate */
RUN stopme IN ph.

/* Start another transaction by running an internal proc of a-txtest.p. This starts a new transaction, which is unnecessary if the TRANSACTION-MODE statement in a-txtest.p specifies the CHAINED option. */
RUN startme IN ph.
RUN custupdate.p ON SERVER h TRANSACTION DISTINCT.
RUN orderupdate.p ON SERVER h TRANSACTION DISTINCT.

/* Causes transaction to rollback including both custupdate and orderupdate */
RUN abortme IN ph.
DELETE PROCEDURE ph.
```

The RUN statements in this example use ABL syntax for executing remote procedures of various types. For more information, see Programming ABL Client Applications on page 87.
Note that while these examples show a close correlation between transaction directives and remote procedure requests, there is no requirement that this be so. You can completely hide any sense that the client is ultimately initiating and managing a transaction by hiding all calls to the internal procedures of the transaction initiating procedure within the logic of your application API. For example, you can encapsulate the remote procedure calls in `a-txclnt.p` in higher-level procedures, such as `start-updates.p`, `rerun-updates.p`, and `end-updates.p`.

### Programming for Open Client applications

In general, programming AppServer procedures for Open Client (Java, .NET, and Web service client) applications is not much different than programming for ABL clients. However, for each AppServer persistent procedure that you expect to be instantiated as an Open Client object, you must also specify the prototypes defined by all super procedures, including user-defined functions, that your persistent procedure uses. The reason for this is that the Open Client Toolkit, which generates the proxy objects and Web service interface for use by Open Client and Web service applications, analyzes the r-code to help determine class definitions for proxy objects. This r-code analysis includes any super procedure references required to define equivalent method overrides for each class.

You can specify super procedure prototypes in one of two ways, depending on how you write your procedures:

- Using the ProtoGen utility available from the Application Development Environment (ADE)
- Using a text editor or the Procedure Editor

#### Using the ProtoGen utility

The ProtoGen utility analyzes the source code of any super procedure that you specify and generates an include file containing the prototypes for all of its internal procedures and user-defined functions. You can access this utility from the PRO*Tools palette, which is available from the Tools menu in the ADE.

One of the more common applications of this utility is to generate prototype include files for persistent procedures that you build in the AppBuilder, especially if you build them using SmartObject technology.

To include super procedure prototypes in AppBuilder procedures:

1. Code each super procedure sufficiently to specify all `PROCEDURE` and `FUNCTION` statements that it requires.

2. From the ADE, choose PRO*Tools from the Tools menu, and click the ProtoGen button, as shown:

3. In the Prototype Generator dialog box, enter the filename of the super procedure and the name of the include file you want to generate from it.
4. Generate the include file and, using the AppBuilder, insert a reference to the include file within every persistent procedure and SmartObject that relies on the specified super procedure prototypes.

5. Compile and generate the r-code for each persistent procedure and SmartObject.

Once you compile and generate the r-code, the Open Client developer can use the Open Client Toolkit to generate the corresponding proxy object. For more information on using ProtoGen, see the ADE online help and for using the AppBuilder, see OpenEdge Development: AppBuilder.

**Using code editors**

For any persistent procedures that you write by hand, you can also use the ProtoGen utility to generate include files from any super procedure prototypes you might need. However, you might prefer or need to code these prototypes by hand, especially in a character environment.

To code the necessary prototypes, edit a persistent procedure that requires them and include a PROCEDURE statement for each internal procedure prototype and a FUNCTION statement for each user-defined function prototype using the IN SUPER option in each case. For internal procedures, you must also follow the PROCEDURE statement with any DEFINE PARAMETER statements to specify the parameters for the internal procedure and terminate the procedure block with an END statement, like a standard procedure block. For user-defined functions, you can include the parameter and return value prototypes within the FUNCTION statement itself. For example:

```
PROCEDURE addMessage IN SUPER:
    DEFINE INPUT PARAMETER pcText AS CHARACTER NO-UNDO.
    DEFINE INPUT PARAMETER pcField AS CHARACTER NO-UNDO.
    DEFINE INPUT PARAMETER pcTable AS CHARACTER NO-UNDO.
END PROCEDURE.

FUNCTION setPosition RETURNS LOGICAL
    (INPUT pfRow AS DECIMAL,
     INPUT pfCol AS DECIMAL) IN SUPER.
```

This example defines prototypes for an internal procedure, `addMessage`, and a user-defined function, `setPosition`, that are defined in some super procedure. Note that there is no indication of what super procedure these prototypes refer to. For the Open Client Toolkit, only the prototypes are required; the IN SUPER option only tells OpenEdge that the procedure or function definition lies elsewhere. As long as the toolkit knows the prototypes, it can generate the correct proxy object definitions. OpenEdge then locates the appropriate code to execute at run time for each super procedure prototype that the Open Client application references.

Again, remember to compile and generate the r-code for each persistent procedure before making it available for access by the Open Client Toolkit.
Programming ABL Client Applications

This chapter describes how to program ABL procedures that execute in an ABL session that runs as a client to an AppServer.

For details, see the following topics:

• Programming for the application session model
• ABL for programming AppServer client procedures
• Accessing AppServer resources from a client
• Connecting an application service or AppServer instance
• Accessing the connection ID on a session-managed client
• Accessing client context regardless of session model
• Running and managing remote procedures
• Disconnecting from an AppServer instance
• Handling conditions and return values
• Managing asynchronous requests
Programming for the application session model

If you are writing a client for a new AppServer application (application service), the decision of what session model to use most immediately impacts the AppServer and its configuration and programming (see Programming the AppServer on page 49). Once this decision is made for a new application service or if you are writing the client for an existing application service, the session model imposes fundamental limitations on your options for programming the ABL client.

There is a way to pass a client context identifier between sessions. You typically rely completely on the API defined for the application service to manage this context. Progress Software Corporation recommends that you never attempt to run a remote external procedure persistently when writing for a session-free application, and it is likely to be a violation of the conventions defined for the API to do so. For more information on the reasons for this, see Programming the AppServer on page 49.

Session-managed application services provide various means to manage context depending on the AppServer operating mode. Where there are any responsibilities involved for the ABL client, this chapter describes them as appropriate for the AppServer operating mode. Otherwise, this chapter describes the differences in functionality available explicitly for each operating mode or session model, as appropriate. For more information on the differences between the two application session models, session-free and session-managed, see AppServer and Client Interaction on page 23.

Note: This chapter addresses programming issues specific to ABL client procedures of an AppServer or application service. It assumes that you are generally familiar with writing and executing ABL procedures. For information on using ABL, see OpenEdge Getting Started: ABL Essentials. While there is some reference to Open Clients, for complete information on programming Java, .NET, and Web service Open Clients, see OpenEdge Development: Open Client Introduction and Programming.

ABL for programming AppServer client procedures

The following table lists the ABL elements that are either valid only for accessing an AppServer connection or have special application in AppServer client programming. The remaining sections in this chapter explain how to use these elements.

Table 5: ABL for programming AppServer client procedures

<table>
<thead>
<tr>
<th>ABL element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASYNC-REQUEST-COUNT</td>
<td>One of the following:</td>
</tr>
<tr>
<td></td>
<td>1) An INTEGER attribute on the server object handle that returns the number</td>
</tr>
<tr>
<td></td>
<td>of active asynchronous requests submitted to this server.</td>
</tr>
<tr>
<td></td>
<td>2) An INTEGER attribute on the procedure handle that returns the number</td>
</tr>
<tr>
<td></td>
<td>of currently outstanding asynchronous requests for this procedure. Can be</td>
</tr>
<tr>
<td></td>
<td>non-zero only if the PROXY and PERSISTENT attributes are both set to TRUE.</td>
</tr>
<tr>
<td>ABL element</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Asynchronous request object handle</td>
<td>A type of handle that maintains the status of an asynchronous request in an ABL client application. This handle provides methods and attributes that allow you to check the status of a remote procedure (internal or external) that is called asynchronously.</td>
</tr>
<tr>
<td>CANCELLED</td>
<td>A <strong>LOGICAL</strong> attribute on the asynchronous request object handle that indicates if the asynchronous request was cancelled using either the <code>CANCEL-REQUESTS( )</code> method or the <code>DISCONNECT( )</code> method on the associated server handle.</td>
</tr>
<tr>
<td>CANCEL-REQUESTS( )</td>
<td>A method on the server object handle that for a session-managed application raises a <strong>STOP</strong> condition in the context of the currently running asynchronous request and purges the send queue of any asynchronous requests that have not been executed.</td>
</tr>
<tr>
<td></td>
<td>For a session-free application, it causes a <strong>STOP</strong> condition to be raised for all currently running asynchronous requests, and purges the send queue of any asynchronous requests that have not been executed.</td>
</tr>
<tr>
<td>CLIENT-CONNECTION-ID</td>
<td>A <strong>CHARACTER</strong> attribute on the server object handle that for a session-managed application returns the connection ID for the physical AppServer connection associated with this server handle.</td>
</tr>
<tr>
<td>COMPLETE</td>
<td>A <strong>LOGICAL</strong> attribute on the asynchronous request object handle that indicates if the asynchronous request is completed and its result is processed on the client.</td>
</tr>
<tr>
<td>CONNECT(</td>
<td>A method on the server object handle that physically connects and associates an AppServer instance or logically connects an application service with the server handle.</td>
</tr>
<tr>
<td>[ connection-parameters ]</td>
<td></td>
</tr>
<tr>
<td>[ , userid]</td>
<td></td>
</tr>
<tr>
<td>[ , password]</td>
<td></td>
</tr>
<tr>
<td>[ , appserver-info ]</td>
<td></td>
</tr>
<tr>
<td>CONNECTED( )</td>
<td>A method on the server object handle that returns <strong>TRUE</strong> for a session-managed application if an AppServer is currently connected and associated with the server handle. For a session-free application it returns <strong>TRUE</strong> if the <code>CONNECT( )</code> method has been successfully executed for an application service associated with this handle and at least one AppServer resource is available for the client to access this application service.</td>
</tr>
<tr>
<td>ABL element</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>CREATE SERVER <code>server-handle</code></td>
<td>A statement that creates a server object handle and stores it in a <code>HANDLE</code> variable.</td>
</tr>
<tr>
<td>DEFINE INPUT PARAMETER <code>parameter</code></td>
<td>A statement for defining <code>INPUT</code> parameters within an event procedure. Each <code>INPUT</code> parameter must correspond to an <code>OUTPUT</code> or <code>INPUT-OUTPUT</code> parameter of the associated asynchronous remote procedure.</td>
</tr>
<tr>
<td>DELETE OBJECT <code>handle</code></td>
<td>A statement that you can use to delete certain objects, including server objects, and persistent procedures (local and remote), and asynchronous request objects.</td>
</tr>
<tr>
<td>DELETE PROCEDURE <code>procedure-handle</code></td>
<td>A statement that you can use to delete both local and remote procedure objects (persistent procedures).</td>
</tr>
<tr>
<td>DISCONNECT( )</td>
<td>A method on the server object handle that disconnects from and removes all reference to the AppServer currently associated with the server handle. Any running or pending asynchronous requests submitted by this client are also cancelled.</td>
</tr>
<tr>
<td>ERROR</td>
<td>A <code>LOGICAL</code> attribute on the asynchronous request object handle that indicates that an <code>ERROR</code> condition was returned from the AppServer as a result of processing the request.</td>
</tr>
<tr>
<td>EVENT-PROCEDURE</td>
<td>A <code>CHARACTER</code> attribute on the asynchronous request object handle that contains the name of the internal procedure to be run as the event procedure for this asynchronous request.</td>
</tr>
<tr>
<td>EVENT-PROCEDURE-CONTEXT</td>
<td>A <code>HANDLE</code> attribute on the asynchronous request object handle that contains the procedure handle of the active procedure context where the event procedure for this asynchronous request is defined.</td>
</tr>
<tr>
<td>FIRST-ASYNC-REQUEST( )</td>
<td>A method on the server object handle that returns the first entry in the list of all current asynchronous request handles for the specified AppServer.</td>
</tr>
<tr>
<td>FIRST-PROCEDURE</td>
<td>A <code>HANDLE</code> attribute on the server object handle that returns the first entry in the list of remote persistent procedures running on the connected AppServer.</td>
</tr>
<tr>
<td>FIRST-SERVER</td>
<td>A <code>HANDLE</code> attribute on the <code>SESSION</code> system handle that returns the handle to the first entry in the chain of server handles for the session.</td>
</tr>
<tr>
<td>FUNCTION ... IN <code>procedure-handle</code></td>
<td>A statement that defines a forward reference to a remote user-defined function prototype. When <code>procedure-handle</code> is a proxy procedure handle, the function is defined on the connected AppServer in the specified procedure object.</td>
</tr>
<tr>
<td>ABL element</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LAST-ASYNC-REQUEST( )</td>
<td>A method on the server object handle that returns the last entry in the list of all current asynchronous request handles for the specified AppServer.</td>
</tr>
<tr>
<td>LAST-PROCEDURE</td>
<td>A HANDLE attribute on the server object handle that returns the last entry in the list of remote persistent procedures running on the connected AppServer.</td>
</tr>
<tr>
<td>LAST-SERVER</td>
<td>A HANDLE attribute on the SESSION system handle that returns the handle to the last entry in the chain of server handles for the session.</td>
</tr>
<tr>
<td>LOCAL-VERSION-INFO</td>
<td>An object reference attribute on the SESSION system handle that references a Progress.Lang.OEVersionInfo object, which identifies the current instance of the OpenEdge AVM, including its OpenEdge version.</td>
</tr>
<tr>
<td>NAME</td>
<td>A CHARACTER attribute on the server object handle that uniquely identifies the AppServer for use in the Application Debugger. For a session-managed application, it returns the connection ID for the AppServer associated with the server handle. For a session-free application, it returns the connection ID of the first AppServer connection created in the connection pool with this server handle when the application service is first bound using the CONNECT( ) method.</td>
</tr>
<tr>
<td>NEXT-SIBLING</td>
<td>One of the following:</td>
</tr>
<tr>
<td></td>
<td>1) A HANDLE attribute on the server object handle that returns the next entry in the list of server handles created for the current ABL session.</td>
</tr>
<tr>
<td></td>
<td>2) A HANDLE attribute on the proxy procedure handle that returns the next entry in the list of proxy procedure handles.</td>
</tr>
<tr>
<td></td>
<td>3) A HANDLE attribute on the asynchronous request object handle that returns the next entry in the list of asynchronous request handles for asynchronous remote procedures submitted for execution on the same AppServer.</td>
</tr>
<tr>
<td>PERSISTENT</td>
<td>A LOGICAL attribute on procedure handles and the THIS-PROCEDURE system handle that is TRUE if the specified procedure is running persistently as a procedure object.</td>
</tr>
<tr>
<td>PERSISTENT-PROCEDURE</td>
<td>A HANDLE attribute on the asynchronous request object handle that returns the proxy procedure handle to the remote persistent procedure that contains the internal procedure executed for the specified asynchronous request. If the request is for an external, rather than an internal, procedure, this attribute returns an invalid handle.</td>
</tr>
<tr>
<td>ABL element</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>PREV-SIBLING</td>
<td>One of the following:</td>
</tr>
<tr>
<td></td>
<td>1. A <strong>HANDLE</strong> attribute on the server object handle that returns the previous entry in the list of server handles created for the current ABL session.</td>
</tr>
<tr>
<td></td>
<td>2. A <strong>HANDLE</strong> attribute on the proxy procedure handle that returns the previous entry in the list of proxy procedure handles.</td>
</tr>
<tr>
<td></td>
<td>3. A <strong>HANDLE</strong> attribute on the asynchronous request object handle that returns the previous entry in the list of asynchronous request handles for asynchronous remote procedures submitted for execution on the same AppServer.</td>
</tr>
<tr>
<td>PROCEDURE-COMPLETE</td>
<td>The event returned for an asynchronous request object handle that indicates the associated remote procedure has completed execution and, as a result, causes execution of the specified event procedure as specified by the <strong>EVENT-PROCEDURE</strong> and <strong>EVENT-PROCEDURE-CONTEXT</strong> attributes.</td>
</tr>
<tr>
<td>PROCEDURE-NAME</td>
<td>A <strong>CHARACTER</strong> attribute on the asynchronous request object handle that provides the name of the remote procedure executed to instantiate this asynchronous request handle.</td>
</tr>
<tr>
<td>PROCESS EVENTS</td>
<td>A statement that you can use to handle any pending <strong>PROCEDURE-COMPLETE</strong> events for asynchronous requests. You can also use any blocking I/O statement, such as the <strong>WAIT-FOR</strong> statement.</td>
</tr>
<tr>
<td>PROXY</td>
<td>A <strong>LOGICAL</strong> attribute on procedure handles that is <strong>TRUE</strong> if the procedure handle is a proxy handle for a persistent, single-run, or singleton procedure running remotely in the context of an AppServer.</td>
</tr>
<tr>
<td>Proxy procedure handle</td>
<td>A type of procedure object handle that references the context of a remote persistent, single-run, or singleton procedure, providing access to its remote internal procedures and user-defined functions.</td>
</tr>
<tr>
<td>QUIT</td>
<td>A <strong>LOGICAL</strong> attribute on the asynchronous request object handle that indicates a <strong>QUIT</strong> condition was returned from the AppServer as a result of processing the request.</td>
</tr>
<tr>
<td>REQUEST-INFO</td>
<td>An object reference attribute on a server object handle (or on the returned asynchronous request object handle for an asynchronous remote request), which references a Progress.Lang.OERequestInfo instance. This class instance provides information about the next remote procedure request to be run on the AppServer connected using the associated server object handle.</td>
</tr>
<tr>
<td>ABL element</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>RESPONSE-INFO</td>
<td>An object reference attribute on a server object handle (or on the returned asynchronous request object handle for an asynchronous remote request), which references a Progress.Lang.OERequestInfo instance. This class instance provides information about the remote procedure response most recently received from the AppServer connected using the associated server object handle.</td>
</tr>
<tr>
<td>RUN ... [PERSISTENT</td>
<td>The statement that executes an external (remote) procedure on a connected AppServer specified by server-handle, in a transaction that is distinct from the client. With the PERSISTENT option, it instantiates the procedure as a persistent procedure object. With the SINGLE-RUN or SINGLETON option, intended for use with stateless and state-free AppServers, the procedure is instantiated on the server without causing the client to be bound to the AppServer or application service. With the ASYNCHRONOUS option (not valid with SINGLE-RUN or SINGLETON), the procedure executes asynchronously with respect to the client. This option also optionally specifies the associated asynchronous request object handle and event procedure to handle results of the request. If server-handle is the SESSION system handle, the external procedure executes synchronously in the local session.</td>
</tr>
<tr>
<td>ON [SERVER] server-handle</td>
<td></td>
</tr>
<tr>
<td>[TRANSACTION DISTINCT]</td>
<td></td>
</tr>
<tr>
<td>[ASYNCHRONOUS ...]</td>
<td></td>
</tr>
<tr>
<td>RUN ... IN procedure-handle</td>
<td>The statement that executes an internal (remote) procedure on a connected AppServer, where procedure-handle specifies a remote persistent, single-run, or singleton procedure that is instantiated on the AppServer and that defines the internal procedure. With the ASYNCHRONOUS option, the internal procedure executes asynchronously with respect to the client. This option also optionally specifies the associated asynchronous request handle and event procedure to handle results of the request. If procedure-handle is the handle to a local persistent procedure, the internal procedure executes synchronously in the local session.</td>
</tr>
<tr>
<td>[ASYNCHRONOUS ...]</td>
<td></td>
</tr>
<tr>
<td>SELF</td>
<td>A system handle that, in the context of an event procedure, returns the asynchronous request object handle of the completed request for which the event procedure is executing.</td>
</tr>
<tr>
<td>SERVER</td>
<td>One of the following:</td>
</tr>
<tr>
<td></td>
<td>1. A HANDLE attribute on a remote persistent procedure handle that returns the server object handle to the AppServer on which the specified remote persistent procedure runs. Valid only if the PROXY and PERSISTENT attributes are both TRUE.</td>
</tr>
<tr>
<td></td>
<td>2. A HANDLE attribute on the asynchronous request object handle that returns the server handle of the AppServer where this asynchronous request was submitted for execution. If the request is run in the local session (using the SESSION system handle), the attribute is set to SESSION.</td>
</tr>
<tr>
<td>ABL element</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Server object handle</td>
<td>A type of handle that provides a connection to an AppServer in an ABL client application. This handle provides server object methods and attributes that allow you to connect and manage an AppServer.</td>
</tr>
<tr>
<td>SSL-SERVER-NAME</td>
<td>A CHARACTER attribute on the socket object handle that returns the SSL server's X.500 Subject name field after any SSL session has been established.</td>
</tr>
<tr>
<td>SINGLE-RUN</td>
<td>A LOGICAL attribute on procedure handles and the THIS-PROCEDURE system handle that is TRUE if the specified procedure is running as a single-run object.</td>
</tr>
<tr>
<td>SINGLETON</td>
<td>A LOGICAL attribute on procedure handles and the THIS-PROCEDURE system handle that is TRUE if the specified procedure is running as a singleton object.</td>
</tr>
<tr>
<td>STOP</td>
<td>A LOGICAL attribute on the asynchronous request object handle that indicates a STOP condition was returned from the AppServer as a result of processing the request.</td>
</tr>
<tr>
<td>TYPE</td>
<td>One of the following: 1) A CHARACTER attribute on the server object handle that returns the handle type, which is &quot;SERVER&quot; for a server object handle. 2) A CHARACTER attribute on the asynchronous request object handle that provides the handle type, which is &quot;ASYNC-REQUEST&quot; for an asynchronous request object handle.</td>
</tr>
<tr>
<td>WAIT-FOR ...</td>
<td>A statement that you can use to handle any pending PROCEDURE-COMPLETE events for asynchronous requests. You can also use PROCESS EVENTS or any other blocking I/O statement, such as the PROMPT-FOR statement.</td>
</tr>
</tbody>
</table>

Accessing AppServer resources from a client

In order to run remote procedures on an AppServer, a client application must first establish a connection with the required AppServer (session-managed) or application service (session-free). When done with the connection, the client application can then disconnect from the AppServer or application service.

Steps for accessing AppServer resources

To allow your client application to access and use an AppServer or application service:

1. Create a server object handle.
2. Use the CONNECT( ) method on the server object handle to physically connect an AppServer instance or logically connect an application service.
3. Invoke remote procedure requests using the `RUN` statement and remote user-defined function requests, as provided by the application service. Delete any proxy procedure object handles to remote persistent procedures as they are no longer needed.

4. When all remote requests to the AppServer or application service have completed, use the `DISCONNECT()` method on the server object handle to disconnect the client application from the AppServer or application service.

5. Delete the server object handle.

The remainder of this chapter provides more information and code examples for each step in this process.

### Features of client-AppServer interactions

Several features describe the relationship between a client application and the AppServer environment. For example:

- A client application can have connections to multiple AppServers or application services simultaneously.

- A client application can connect directly to an AppServer (practical only when session-managed) or indirectly using a NameServer.

- For a session-managed application, the choice of AppServer for each connection is determined by the NameServer based on the application service name provided by the client. The NameServer chooses the connection from among all AppServers that support the specified application service. The actual location of the AppServer is completely transparent to the client. For a connection using the AppServer Internet Adapter (AIA), the client accesses the AIA using a URL that includes the application service as an attribute and the AIA either uses a NameServer or connects to the AppServer directly, depending on how it is configured. Using an AIA, the actual locations of both the NameServer and the AppServer are completely transparent to the client.

- For a session-free application, the NameServer provides one or more AppServer connections to be maintained by the client in a connection pool based on the application service name and other parameters provided by the client. The actual locations of these AppServer connections are completely transparent to the client. For an OpenEdge Web service, the connection pool is maintained by the WSA or the OpenEdge Adapter for Sonic ESB (if the Web service is deployed to the Sonic ESB).

- How the AppServer allocates AppServer agents to service requests that it receives depends on the AppServer operating mode, which is largely transparent to the client.

**Note:** You might have to write the client application differently to access and manage responses from an application service, depending on the operating mode.

- An AppServer can manage transactions in a number of ways. The transaction management mechanism used by the AppServer is completely transparent to the client. The client has no direct control over the transaction mechanics used by the AppServer.
Connecting an application service or AppServer instance

The connection mechanism to access an application service depends largely on the session model and the operating mode that supports it. So, a connection to:

- A session-managed application service always resolves to a single connection between one AppServer and the client. This is the mechanism of a single physical connection.

- A session-free application service resolves to a pool of one or more AppServer connections (connection pool) that support client access to the application service. This is the mechanism of a single logical connection.

For more information on these two different types of connections and how the operating mode affects them, see AppServer and Client Interaction on page 23.

For both session models, you use the \texttt{CONNECT( )} method on a server object handle to create the appropriate connection.

To connect an ABL client to an AppServer or application service:

1. Create a server object handle.
2. Execute the \texttt{CONNECT( )} method on the server object handle, passing the parameters necessary to identify and configure the connection.

Creating a server object handle

To create a server object handle, you must define a \texttt{HANDLE} variable and execute the \texttt{CREATE SERVER} statement to store the new server object handle. For example:

```
DEFINE VARIABLE hAppSrv AS HANDLE NO-UNDO.
CREATE SERVER hAppSrv.
```

You can then use this handle to physically connect to an AppServer instance or logically connect to an application service.

Establishing a connection with the \texttt{CONNECT( )} method

The \texttt{CONNECT( )} method takes up to four character string arguments in the following order, all of which are optional, depending on the configuration of your AppServer environment:

1. \texttt{connection-parameters}
2. \texttt{userid}
3. \texttt{password}
4. \texttt{appserver-info}
You use the `connection-parameters` argument to locate and configure the connection to one or more AppServer instances that support the required business function (application service).

The `userid`, `password`, and `appserver-info` arguments apply only to session-managed configurations and are passed from the client application to the AppServer, which passes them to the Connect procedure (if defined). If a Connect procedure is not defined for the connected AppServer, these arguments are discarded. The requirement to specify these arguments depends entirely on the application service defined for the connected AppServer.

An AppServer can reject a connection request if:

- The Connect procedure fails
- There are not enough system resources (for example, AppServer agents) available to service the connection
- The specified session model for the connection is incorrect

A runtime error is generated when a connection is rejected by an AppServer.

In session-managed mode, the AppServer Connect procedure has the ability to return a user-defined string if it uses the `RETURNString` or `RETURN ERRORString` statements. To access the string, simply use the ABL `RETURN-VALUE` function in your client-side connection code. A null value indicates that the Connect procedure did not return a string.

### Connection parameters argument

The `connection-parameters` argument specifies a comma-separated list of parameters necessary to establish the AppServer connection. These parameters include two types:

- A basic set used to connect to an AppServer instance or application service, regardless of the session model
- A set for specifying and managing the session model of the connection

The following table describes the basic connection parameters\(^1\) to connect to an AppServer instance or application service, regardless of the session model.

<table>
<thead>
<tr>
<th>Connection parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-AppService application-service</code></td>
<td>If you connect through a NameServer, the name of an application service supported by the specified NameServer (defaults to the default service for the specified NameServer). If you connect directly to an AppServer, this parameter is ignored.</td>
</tr>
<tr>
<td>`-H [ host_name</td>
<td>IP-address ]`</td>
</tr>
</tbody>
</table>

---

\(^1\) Previous versions of the AppServer allow you to include a network protocol using the `-N` parameter, which must always specify TCP. While still allowed, it is optional and always defaults to TCP/IP.
### Connection parameter

<table>
<thead>
<tr>
<th>Description</th>
<th>Connection parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>The UDP port number for a NameServer, or, if you connect directly, the TCP/IP port number for an AppServer connection. You can specify either an explicit port number or a service name. If you use a service name, the method uses the port number associated with that name in the TCP/IP services file (defaults to 5162).</td>
<td>`-S [service-name</td>
</tr>
<tr>
<td>If specified, the <code>-H</code> and <code>-S</code> parameters are interpreted as the network address and TCP/IP port number of an AppServer connection. Otherwise, the <code>-H</code> and <code>-S</code> parameters are interpreted as the network address and UDP port number of a NameServer.</td>
<td><code>-DirectConnect</code></td>
</tr>
<tr>
<td>If specified, the connection is direct to the AppServer using Secure Sockets Layer (SSL) tunneling. (Used in conjunction with the <code>-AppService, -H, and -S</code> parameters).</td>
<td><code>-ssl</code></td>
</tr>
<tr>
<td><strong>Note:</strong> Be very sure you need SSL before using this option. SSL incurs more or less heavy performance penalties, depending on resources and load.</td>
<td></td>
</tr>
<tr>
<td>If specified, the connection does not reuse the SSL session ID when reconnecting to the same SSL-enabled server (either a Web server with HTTPS or an SSL-enabled AppServer).</td>
<td><code>-nosessionreuse</code></td>
</tr>
<tr>
<td>If specified, turns off host verification for an SSL-enabled connection, either using HTTPS with the AIA or using a direct connection to an SSL-enabled AppServer. Without this parameter, the client compares the host name specified for the connection with the Common Name specified in the server digital certificate, and raises an error if they do not match. With this parameter specified, the client never raises the error. For more information on digital certificates, see OpenEdge Getting Started: Core Business Services - Security and Auditing.</td>
<td><code>-nohostverify</code></td>
</tr>
<tr>
<td>Connection parameter</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>-pf filename</code></td>
<td>A text file containing any of the other AppServer connection parameters described in this table or . If this file contains any other OpenEdge startup parameters, the method ignores them.</td>
</tr>
<tr>
<td><code>-URL Web-or-AppServer-path</code></td>
<td>An HTTP (or HTTPS-based) URL to an AIA (for an Internet-secure AppServer connection) or an AppServer-based URL (with or without SSL tunneling for an SSL-enabled AppServer connection). The same URL format is also used to connect Open Clients to an AppServer. For more information see Connecting to AppServers Using a URL on page 187</td>
</tr>
</tbody>
</table>

**Note:** Be very sure you need SSL (either, and especially both, an HTTPS or SSL-enabled AppServer) before using this option. SSL at any point in a networked application incurs more or less heavy performance penalties, depending on resources and load.

**Note:** Connections to an Internet-secure (HTTPS) or SSL-enabled AppServer require the management of public keys on the client (SSL client) and private keys on the server (SSL server). For an Internet-secure AppServer, the SSL server is the Web server that hosts the AIA. For an SSL-enabled AppServer, the SSL server is the AppServer itself. For information on configuring a Web server for HTTPS, see your Web server documentation. For information on using SSL to secure an AppServer, see the sections on SSL security in Design and Implementation Considerations on page 137. For information on configuring an AppServer for SSL tunneling, see OpenEdge Application Server: Administration. For more information on SSL and managing private key and digital certificate stores for OpenEdge SSL clients and servers, see OpenEdge Getting Started: Core Business Services - Security and Auditing.

The following table describes connection parameters for specifying and managing the session model of the connection.
### Table 7: AppServer session model connection parameters;

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Session model</th>
<th>Description</th>
</tr>
</thead>
</table>
| `-sessionModel sessionModel` | Session-free or Session-managed | Session model supported by the AppServer operating mode, specified by one of the following values:  
- Session-managed  
- Session-free  
The value is **not** case-sensitive. This parameter is required for session-free applications and is optional for session-managed applications.  
This value must match the AppServer operating mode or the CONNECT( ) method fails.  
**Default:** Session-managed |
| `-connectionLifetime nSeconds` | Session-free        | The maximum number of seconds that a given connection can be reused before it is destroyed. Connections whose lifetime exceeds the specified value are destroyed as they become available.  
An available connection is one that is not currently reserved to run a request. Bound connections associated with remote persistent procedures are not available for reuse until the persistent procedure is deleted. So, bound connections remain available as long as necessary, even if they exceed the specified value.  
**Default:** 300 |
| `-initialConnections nConnections` | Session-free | The number of connections established when the CONNECT( ) method executes on a given server handle. The value must be greater than zero. If the specified number of connections cannot be created, the CONNECT( ) method fails and any successfully-created connections are closed.  
**Default:** 1 |
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Session model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-maxConnections nConnections</td>
<td>Session-free</td>
<td>The maximum number of connections that can be created for a given server handle to execute non-persistent external procedures. The value must be greater than or equal to zero. If this value is zero, there is no limit to the number of connections that can be created. <strong>Note:</strong> For calls to persistent procedures, their internal procedures, and user-defined functions, the client has no limit on the number of connections that can be created. <strong>Default:</strong> 0</td>
</tr>
<tr>
<td>-nsClientMaxPort portNum</td>
<td>Session-free or Session-managed</td>
<td>The maximum value for the UDP port number used by the client when communicating with the Name Server. If this value is zero, OpenEdge chooses the NameServer client port randomly. This value should be greater than or equal to the value of the -nsClientMinPort parameter. <strong>Default:</strong> 0</td>
</tr>
<tr>
<td>-nsClientMinPort portNum</td>
<td>Session-free or Session-managed</td>
<td>The minimum value for the UDP port number used by the client when communicating with the Name Server. If this value is zero, OpenEdge chooses the NameServer client port randomly. <strong>Default:</strong> 0</td>
</tr>
<tr>
<td>-nsClientPicklistExpiration nSeconds</td>
<td>Session-free</td>
<td>The maximum amount of time, in seconds, that the client retains an AppServer pick list for an application service. <strong>Default:</strong> 300</td>
</tr>
<tr>
<td>-nsClientPicklistSize nPicks</td>
<td>Session-free</td>
<td>The number of AppServer picks to request from the NameServer each time it looks up the available AppServer connections for a given application service name. <strong>Default:</strong> 1</td>
</tr>
</tbody>
</table>
### Session Model Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Session model</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-nsClientPortRetry</td>
<td>Session-free or</td>
<td>The maximum number of attempts that the client makes to get a valid local UDP port number when attempting to communicate with the NameServer. Default: 0</td>
</tr>
<tr>
<td>nRetries</td>
<td>Session-managed</td>
<td></td>
</tr>
<tr>
<td>-nsClientDelay</td>
<td>Session-free or</td>
<td>The interval, in milliseconds, that the client waits between attempts to get a valid UDP port number when attempting to communicate with the NameServer. Default: 0</td>
</tr>
<tr>
<td>nMilliseconds</td>
<td>Session-managed</td>
<td></td>
</tr>
</tbody>
</table>

Note that the actual AppServer that the client connects to is controlled by the NameServer based on the application service (-AppService) name specified by the client. The OpenEdge interface in cooperation with the NameServer connects the client application to one of the AppServer instances that supports the specified application service. If you do not specify an application service, the NameServer uses whatever AppServer registers itself as the default service, if any. For more information on load balancing, see the information on load balancing and session models in AppServer and Client Interaction on page 23.

If the application service is unknown to the NameServer, the client application receives an error. Otherwise, the connection proceeds and any configured Connect procedure executes for the connected AppServer (session-managed only). For more information on application services and NameServers, see the AppServer chapter in OpenEdge Getting Started: Application and Integration Services.

#### Session-managed application arguments

The userid, password, and appserver-info arguments are optional. However, the requirement to specify these arguments, and to provide specific values for any of them, depends on whether the AppServer Connect procedure is being used on the AppServer you are connecting.

**Note:** These arguments are ignored for a session-free application service, because AppServer Connect procedures do not execute on a session-free AppServer.

If the AppServer Connect procedure is defined for the AppServer, then you must know how the AppServer Connect procedure is implemented to determine how to specify these values. If you do not specify a value for an argument, the Unknown value (?) is passed to the AppServer Connect procedure for that argument. OpenEdge only passes these values from the `CONNECT()` method to the AppServer Connect procedure. It is the procedure that actually evaluates the values that are passed.

**Note:** When determining what arguments to pass to the `CONNECT()` method, understand that an AppServer accepts a connection request only if any configured Connect procedure executes successfully.

If the AppServer Connect procedure is not defined for the AppServer, you do not need to pass the three application arguments to the `CONNECT()` method. If you happen to provide them, they are ignored.
For more information on Connect procedures, see Programming the AppServer on page 49.

Connection examples

The following code examples show how a client application specifies the connection and application arguments to connect an AppServer or application service.

Connecting to a session-managed AppServer

In this session-managed connection example, the -AppService, -H, and -S parameters are passed as the connection-parameters argument to the CONNECT( ) method. Specific userid and password values are also passed as connection arguments. A value is not supplied for the appserver-info argument. For example:

```
DEFINE VARIABLE hAppSrv AS HANDLE NO-UNDO.
DEFINE VARIABLE lReturn AS LOGICAL NO-UNDO.

CREATE SERVER hAppSrv.
lReturn = hAppSrv:CONNECT
    ("-AppService inventory -H zeus -S 5162","SMITH","STARSHIP").
IF NOT lReturn THEN DO:
    DELETE OBJECT hAppSrv NO-ERROR.
    RETURN ERROR "Failed to connect to Inventory AppServer: " + RETURN-VALUE.
END.
...
lReturn = hAppSrv:DISCONNECT().
DELETE OBJECT hAppSrv NO-ERROR.
```

This code tries to connect an AppServer that supports the application service, inventory. It sends its connection request to a NameServer that runs on a machine with the host name, zeus and is listening on UDP port 5162.

When the CONNECT( ) method executes, the NameServer provides the location of an AppServer broker to the client that supports the inventory application service. The NameServer chooses the AppServer broker randomly from among all AppServers that support the application service and according to load balance settings (if any) established for the application service. The client then connects to the broker, passing the last two method arguments as the first two parameters of the AppServer Connect procedure. The client passes a value of "SMITH" for userid and "STARSHIP" for password. It passes the Unknown value (?) for appserver-info because a value is not supplied for it.

If the Connect procedure fails, it could specify a user-defined string. Appending the RETURN-VALUE function to a generic error message in your client-side connection code would capture any user-defined string returned by the AppServer Connect procedure.

**Note:** The previous code example, and all remaining code examples in this chapter, are presented in simplified form for discussion purposes. In actual practice, you might use the Parameter File (-pf) parameter as the connection-parameters argument and variables to store the userid and password (probably encrypted).
Connecting to a session-free AppServer

In this session-free connection example, the `connection-parameters` argument to the `CONNECT()` method contains the `-AppService`, `-H`, `-S`, `-sessionModel`, and `-initialConnections` parameters, as shown:

```abl
DEFINE VARIABLE hInventory AS HANDLE NO-UNDO.
DEFINE VARIABLE lReturn AS LOGICAL NO-UNDO.

CREATE SERVER hInventory.
lReturn = hInventory:CONNECT("-AppService inventory -S 5163 -H 4glservices -sessionModel Session-free -initialConnections 25").
IF NOT lReturn THEN DO:
   DELETE OBJECT hInventory NO-ERROR.
   RETURN ERROR "Failed to connect to Inventory application service".
END.
IF ERROR-STATUS:ERROR THEN DO:
   lReturn = hInventory:DISCONNECT().
   DELETE OBJECT hInventory NO-ERROR.
   RETURN ERROR RETURN-VALUE.
END.
...
lReturn = hInventory:DISCONNECT().
DELETE OBJECT hInventory NO-ERROR.
```

This code tries to connect the `inventory` application service by setting up an initial connection pool containing 25 physical connections to AppServers that support this application service. It sends its connection request to a NameServer that runs on a machine with the host name, `4glservices` and is listening on UDP port 5163. When the `CONNECT()` method executes, the NameServer returns the location of an AppServer broker that supports the `inventory` application service, once for each of the 25 requested physical connections. The client then creates the 25 physical connections to these brokers, distributed randomly and according to load balance settings (if any) established for the application service.

Accessing the connection ID on a session-managed client

On a session-managed ABL client, a connection ID for an AppServer connection is available as the value of the `CLIENT-CONNECTION-ID` attribute on the server object handle for the connection. This is a read-only attribute of type `CHARACTER` that the AppServer generates for every session-managed client connection. For an Open Client application, OpenEdge provides appropriate common methods to access the connection ID (see `OpenEdge Development: Open Client Introduction and Programming`).

For a client, the connection ID of a given AppServer connection remains the same until the client disconnects from the AppServer. If the same client reconnects to the same AppServer, the connection ID of the new connection (and thus the value of the `CLIENT-CONNECTION-ID` attribute for that connection) is different from the connection ID of any previous connection. The `CLIENT-CONNECTION-ID` attribute works only on an ABL client running in a session-managed application.
The CLIENT-CONNECTION-ID value is also shared with the SERVER-CONNECTION-ID attribute on the AppServer SESSION handle. This value can therefore be used by both the client and AppServer to maintain a consistent client context especially with an AppServer running in stateless operating mode. This value can also be useful to provide a common audit trail ID for any AppServer and client in a session-managed application.

For information on how the AppServer generates this value, see Accessing the connection ID on page 55. For information on how to use this value to manage connection context, see Managing stateless connection context on page 66. For information on how to use this value in audit trails, see Audit trails on page 161.

Note: For an ABL client in a session-free application, the CLIENT-CONNECTION-ID attribute always has the Unknown value (?) because there is no sustained physical connection to the AppServer in a session-free application. In order to maintain client connection context for an application regardless of session model, use the ClientContextId property on the Progress.Lang.OERequestInfo class. For more information on using this property, see Accessing client context regardless of session model on page 105.

Accessing client context regardless of session model

OpenEdge supports the ClientContextId property on the Progress.Lang.OERequestInfo class that you can use to identify client context for both session-free and session-managed applications, especially in multi-tier application environments. OpenEdge provides instances of this class that you can reference using attributes of both the ABL client’s server object handle and the AppServer agent’s SESSION system handle.

OpenEdge creates one instance of OERequestInfo when the client creates a server object handle and assigns the REQUEST-INFO attribute to the object reference. OpenEdge also initializes the ClientContextId property on this instance with a globally unique value that the client application can change, if necessary, before each remote request using the same server handle.

When the client makes a remote request on this server handle, the values of this property and other properties of OERequestInfo are sent to the AppServer agent, where they initialize the same properties in a separate instance of OERequestInfo that is referenced by the CURRENT-REQUEST-INFO attribute on the agent’s SESSION handle. When the agent completes its handling of the request, it returns the OERequestInfo property values to the client that are set in the instance referenced by the CURRENT-RESPONSE-INFO attribute. Some of these values are different from those available through the CURRENT-REQUEST-INFO attribute, but the ClientContextId property value is the same unless your agent code changes it prior to returning to the client.

When the remote request returns a response to the client, the client code can read the property values returned from the AppServer in an OERequestInfo instance referenced by the RESPONSE-INFO attribute on the server object handle (or on the asynchronous request object handle returned for an asynchronous request).

For each subsequent client request on the same server handle, OpenEdge initializes the ClientContextId property on the REQUEST-INFO attribute instance from the ClientContextId value on the RESPONSE-INFO attribute, unless the client application changes the value prior to the request. In this way, a ClientContextId value can be propagated between the same client and AppServer with little or no code intervention.
Typically you use the *client context identifier* value set for the ClientContextId property to key a context store that is initialized at the start of a user login session and shared between the client and all AppServers that participate in the same login session. For more information on using the ClientContextId and other properties of the Progress.Lang.OERequestInfo class, especially on the AppServer, see *Managing state-free and stateless client context* on page 69. The following section describes use of these properties on the ABL client.

**Managing context from the ABL client with OERequestInfo objects**

OpenEdge context information accessible by an ABL client provides:

- **OpenEdge version information for the current ABL client** — The client session can access the LOCAL-VERSION-INFO attribute on its SESSION system handle to find its OpenEdge Release information. This object reference attribute returns a Progress.Lang.OEVersionInfo class instance and its OEMajorVersion, OEMinorVersion, and OEMaintVersion properties provide the version information.

- **OpenEdge AppServer version information on the current ABL client** — This identifies the OpenEdge Release information for an AppServer that handles a remote request for the client. The client can access the Release information for a connected AppServer using the VersionInfo property on the OERequestInfo instance returned by the server object handle's RESPONSE-INFO attribute.

- **Client context identifier for the next request** — This is the value of the ClientContextId property on the OERequestInfo instance referenced by the server handle's REQUEST-INFO attribute when the client makes a remote request. The client application can change this value before each request, typically when the client is also an AppServer agent in order to propagate the originating client context identifier to yet another AppServer in a multi-tier application. Otherwise, by default, the value of the property available through the server handle's RESPONSE-INFO attribute is used from the most recent AppServer response. The next AppServer agent can retrieve this property value on the OERequestInfo instance referenced by the CURRENT-REQUEST-INFO attribute on its SESSION handle.

- **Client context identifier for the most recent AppServer response** — The client can access the client context identifier returned by the AppServer for the most recently executed request as the value of the ClientContextId property on the OERequestInfo instance referenced by the RESPONSE-INFO attribute on the server object handle (or the asynchronous object handle returned for an asynchronous request). This value can be different from the ClientContextId property value sent by the client for the most recently executed request if the AppServer changed the value for its response.

- **Identifier for each request** — The client automatically generates a globally unique identifier for each request in the RequestId property on the OERequestInfo instance referenced by the server handle's REQUEST-INFO attribute. The AppServer agent can read this property value on the OERequestInfo instance referenced by its SESSION handle's CURRENT-REQUEST-INFO or CURRENT-RESPONSE-INFO attribute. This value is read-only and cannot be changed by any application code.

- **Identifier for the most recent request** — The identifier for the previous request executed on a client's server object handle is the RequestId property value on the OERequestInfo instance referenced by the RESPONSE-INFO attribute on the server object handle (or the asynchronous object handle returned for an asynchronous request).

For more information on these class properties and handle attributes, see *OpenEdge Development: ABL Reference.*
In addition to maintaining context for a client and AppServer connection, you can use the client context identifier as the key to implementing several multi-tier security models. For more information, see Implementing multi-tier security models on page 72.

Running and managing remote procedures

Once a client application connects to an AppServer, the client application can invoke requests to do the following:

- Run remote procedures as persistent, single-run, singleton, or non-persistent on the AppServer using the **RUN** statement.
- Run internal procedures and user-defined functions in remote external procedures.
- Run both remote external and remote internal procedures asynchronously.

**Note:** A user-defined function cannot run asynchronously. An external procedure called with **RUN SINGLE-RUN** or **RUN SINGLETON** cannot run asynchronously. However, internal procedures contained by such a procedure can run asynchronously.

- Delete remote persistent procedures using the **DELETE PROCEDURE** or **DELETE OBJECT** statement.

Running remote procedures

A remote procedure, whether it runs persistently or non-persistently, is scoped only to the AppServer session in which it runs. The order of execution for a remote procedure, relative to the client, depends on whether you execute it synchronously or asynchronously.

Synchronous remote procedures

If you run a remote procedure synchronously, the client application blocks until the remote procedure request completes and returns to the client. Execution then resumes in the client after the **RUN** statement that invoked the remote procedure.

Asynchronous remote procedures

If you run a remote procedure asynchronously, the client application continues execution immediately after the **RUN** statement that invokes the remote procedure completes. The remote procedure executes on the AppServer whenever the AppServer is available to execute it. You can then access the results of the remote request in an event procedure that executes in response to a **PROCEDURE-COMPLETE** event. The client handles the event in the context of a **PROCESS EVENTS** or other blocking I/O statement, similar to a user-interface event, and executes the event procedure like a user-interface trigger.

**RUN statement options for remote procedures**

You can use various options on the **RUN** statement to execute remote procedures:
Run a synchronous external procedure, using the ON [SERVER] handle-variable [TRANSACTION DISTINCT] option, as shown:

```
DEFINE VARIABLE hAppSrv AS HANDLE NO-UNDO.
CREATE SERVER hAppSrv.
  hAppSrv:CONNECT(...).
  . . .
RUN order.p ON SERVER hAppSrv TRANSACTION DISTINCT.
```

With a small change, this example can execute a remote procedure or a local procedure, depending on whether `hAppSrv` contains a server handle value or the value of the `SESSION` system handle. Thus, with the same statement, you can run `order.p` remotely (with a server handle) or locally (with the `SESSION` handle), and determine the choice at run time, as shown:

```
DEFINE VARIABLE hAppSrv AS HANDLE NO-UNDO.
CREATE SERVER hAppSrv.
  IF NOT hAppSrv:CONNECT(...) THEN
    hAppSrv = SESSION.
  . . .
RUN order.p ON SERVER hAppSrv.
```

Run a remote persistent procedure, using the ON SERVER and PERSISTENT SET handle-variable options, as shown:

```
DEFINE VARIABLE hAppSrv AS HANDLE NO-UNDO.
DEFINE VARIABLE hOrder AS HANDLE NO-UNDO.
CREATE SERVER hAppSrv.
  hAppSrv:CONNECT(...).
  . . .
RUN order.p PERSISTENT SET hOrder ON SERVER hAppSrv TRANSACTION DISTINCT.
```

A synchronous remote persistent procedure executes similarly to the same persistent procedure run locally. The AppServer session creates the context for the persistent procedure as it starts to execute, and that context persists after it returns until the end of the AppServer connection or until the persistent procedure is explicitly deleted (see Deleting remote persistent procedures on page 114. Once a remote persistent procedure context is instantiated, you can execute any remote internal procedure or user-defined function that is defined within that remote persistent procedure.

The execution context for a remote persistent procedure is managed almost exclusively within the AppServer session where the persistent procedure is instantiated. However, the persistent procedure handle (`hOrder` in the example), as a proxy procedure handle, provides some additional access to the remote context from the client, allowing the client to delete the persistent procedure remotely. For more information on proxy procedure handles, see Understanding proxy procedure handles on page 113.
• Run a remote external procedure on a stateless or state-free AppServer, using the **ON SERVER** option and either **SINGLE-RUN** **SET** handle-variable (as shown below) or **SINGLETON** **SET** handle-variable

```abl
DEFINE VARIABLE hAppSrv AS HANDLE NO-UNDO.
DEFINE VARIABLE serverHandle AS HANDLE NO-UNDO.
CREATE SERVER hAppSrv.
serverHandle:CONNECT(...).
.
RUN aproc.p SINGLE-RUN SET procHandle ON SERVER serverHandle.
RUN internalProcA IN procHandle (INPUT argIn1, OUTPUT argout1).
RUN internalProcB IN procHandle. (INPUT argIn2, OUTPUT argout2).
DELETE PROCEDURE procHandle.
```

Although the preceding example illustrates the use of the **SINGLE-RUN** option, **SINGLETON** can be substituted with no other syntax differences. The purpose of both options is to allow the client to access internal procedures and user-defined functions found in the specified external procedure, without causing the client to be bound to the AppServer as it is when **RUN PERSISTENT** is used. Single-run and singleton operations offer the potential for greater AppServer throughput (since the lack of binding increases the availability of agents) and improved application performance (because the number of messages that the client must send to the AppServer is minimized).

In the case of both **SINGLE-RUN** and **SINGLETON**, the remote external procedure is not instantiated, and no message is sent to the AppServer, until the first internal procedure or user-defined function is called (RUN internal procA in the preceding example). On each such call to an internal procedure or user-defined function, the AVM sends the AppServer a message with the names of both the external procedure and the internal procedure or function; these are the only messages generated. Note that if AppServer access is being controlled by an export list, procedures run using the **SINGLE-RUN** and **SINGLETON** options should be added using the **EXPORT( )** method. See Controlling AppServer entry points on page 63 for more information.

The difference between **SINGLE-RUN** and **SINGLETON** lies in how the AVM deals with the remote external procedure instance(s) created on the server. In the single-run case, a new instance of the external procedure is created with each call into that procedure, and the instance is deleted after execution of the call. In the singleton case, once the external procedure is instantiated on the first call to an internal procedure or user-defined function, that instance remains in AppServer memory until it is explicitly deleted or the session ends. This singleton instance is used for all subsequent calls into the remote procedure running on this AppServer agent, whether the calls come from the client that originally caused the instantiation or from a different client.

The following restrictions apply to the use of the **SINGLE-RUN** and **SINGLETON** options:

- The remote external procedure cannot run asynchronously; the use of the **ASYNCHRONOUS** option with the **SINGLE-RUN** or **SINGLETON** option results in an error. However, its internal procedures can run asynchronously.
- The external procedure cannot have input or output parameters.
- **DELETE PROCEDURE** only deletes the proxy handle on the client for a procedure run with the **SINGLE-RUN** or **SINGLETON** option. This is because a single-run procedure is deleted automatically by the AVM, and a singleton procedure can be running on more than one AppServer agent, where it can be deleted using ABL code running on the agent.
Run a synchronous internal procedure or user-defined function using the **INhandle-variable** option, as shown:

```
DEFINE VARIABLE hAppSrv AS HANDLE NO-UNDO.
DEFINE VARIABLE hProc AS HANDLE NO-UNDO.

FUNCTION fnOrders RETURNS LOGICAL IN hProc.
CREATE SERVER hAppSrv.
hAppSrv:CONNECT(...).
RUN order.p PERSISTENT SET hProc ON SERVER hAppSrv TRANSACTION DISTINCT.

RUN GetOrders IN hProc.
IF fnOrders THEN
   DISPLAY "Orders Found!"
```

In this example, both the `GetOrders` procedure and `fnOrders` user-defined function are defined in the remote persistent procedure specified by the proxy procedure handle, `hProc`.

For stateless and state-free AppServers, the same functionality is available by using the **INhandle-variable** option with RUN SINGLE-RUN or RUN SINGLETON (provided the external procedure associated with `handle-variable` does not have parameters). These options are preferable to RUN PERSISTENT because they do not bind the client to the AppServer or application service.

Note that running a synchronous remote internal procedure or invoking a remote user-defined function is syntactically identical to running it locally. OpenEdge automatically determines the right AppServer connection and remote procedure or function to execute from the specified proxy procedure handle.

- **Pass** `INPUT`, `OUTPUT`, and `INPUT-OUTPUT` variable, `TEMP-TABLE`, or `ProDataSet` parameters, **but not** `BUFFER` parameters.
• Run a procedure (internal or external) asynchronously, using the ASYNCHRONOUS option, as shown:

```
DEFINE VARIABLE hAppSrv AS HANDLE NO-UNDO.
DEFINE VARIABLE hAsync AS HANDLE NO-UNDO.
DEFINE VARIABLE hAsync2 AS HANDLE NO-UNDO.
DEFINE VARIABLE hProc AS HANDLE NO-UNDO.

DEFINE TEMP-TABLE ttOrder...

FUNCTION fnHighVolume RETURNS LOGICAL IN hProc.

CREATE SERVER hAppSrv.
   hAppSrv:CONNECT(...).

RUN order.p PERSISTENT SET hProc ON SERVER hAppSrv TRANSACTION DISTINCT
   ASYNCHRONOUS SET hAsync.

DO WHILE NOT hAsync:COMPLETE:
   PROCESS EVENTS.
   IF hAsync:COMPLETE THEN
      RUN GetOrders IN hProc ASYNCHRONOUS SET hAsync2
         EVENT-PROCEDURE GetOrderRecords IN THIS-PROCEDURE
            (INPUT-OUTPUT TABLE ttOrder).
   ELSE /* Order module not ready, so work on other stuff while waiting. */
     ...
   END.

WAIT-FOR CLOSE OF THIS-PROCEDURE.

PROCEDURE GetOrderRecords:
   DEFINE INPUT PARAMETER TABLE FOR ttOrder.
   DEFINE VARIABLE lSee AS LOGICAL NO-UNDO.

   IF fnHighVolume THEN
      DISPLAY "Orders > 100: Call in extra help!".
      MESSAGE "Do you want to see the orders?" UPDATE lSee.
      IF lSee THEN ... /* Display ttOrder. */
      RETURN.
   END.
```

This example asynchronously instantiates a remote persistent procedure, order.p, and when available, asynchronously calls the GetOrders remote internal procedure defined in order.p to return a TEMP-TABLE parameter with order information. While waiting for the persistent procedure, order.p, to become available, the procedure does an unspecified amount of work. This part of the example is procedural, using the PROCESS EVENTS statement to handle the PROCEDURE-COMPLETE event each time through the loop. The status of the request is checked by testing the COMPLETE attribute of the asynchronous request handle (hAsync).

After the example runs GetOrders asynchronously, the example blocks using the WAIT-FOR statement to handle the PROCEDURE-COMPLETE event for the GetOrders request. When GetOrders completes, the associated event procedure, GetOrderRecords, executes. GetOrderRecords returns the TEMP-TABLE parameter and executes the remote user-defined function fnHighVolume (also defined in order.p) to provide warning of heavy order traffic. Note that the remote user-defined function is (and can only be) called synchronously.
Note: An external procedure called with `RUN SINGLE- Run` or `RUN SINGLETON` cannot run asynchronously. That is, including the `ASYNCHRONOUS` option with `SINGLE-Run` or `SINGLETON` is an error. However, internal procedures defined in a single-run or singleton procedure can run asynchronously. The syntax shown in the preceding example for asynchronously running `GetOrders` is also valid in the context of a single-run or singleton procedure.

Clearly, the handling of asynchronous requests is more complex than for synchronous requests. While this example is partly procedural, you typically handle all asynchronous requests in an event-driven context using a blocking I/O statement, such as the `WAIT-FOR` statement. For more information on how to manage asynchronous requests, see Managing asynchronous requests on page 122.

However, note that you cannot use compile-time (preprocessor) arguments when running a remote procedure.

Passing class-based objects as parameters between an AppServer and an ABL client

Objects can be passed as `INPUT`, `OUTPUT`, and `INPUT-OUTPUT` parameters between an AppServer and an ABL client. Because parameters are passed by value between an AppServer and client, the object on the receiving side is a new instance, and the original may be garbage collected if it is not referenced elsewhere. A class must be defined on both client and server sides for objects of that class to be passed as parameters, and the data members, properties, events, and method signatures must match exactly on both sides. Otherwise, the AVM will raise a runtime error on the client-side `RUN` statement. (Any logic associated with class members, like the `GET` and `SET` methods of a property, can differ on the client and AppServer sides without causing an error, as long as their definitions match.)

An object passed as a parameter is serialized and then reconstructed from the serialized data on the receiving side. All non-static data members, properties, ProDataSets, and temp-tables of an object are serialized. Deserialization of an object on the receiving happens starting with the root class and continuing down through the most-derived class. No instance constructors are called, and `GET` and `SET` methods for properties are not invoked during serialization or deserialization. The values of the properties are simply transferred.

The following restrictions apply to objects being passed as parameters (or thrown as errors) between an AppServer and client:

- In the case of a user-defined class, the object's class and all of the classes in its hierarchy must be marked as `SERIALIZABLE`. For more information on marking a class as `SERIALIZABLE`, see the `CLASS` statement entry in OpenEdge Development: ABL Reference.

- The content of static data members is not serialized, and the state of queries, buffers, open files, streams, and event subscriptions, for example, are not maintained.

- All of the object's data members that are defined as class-based objects must be of a class type that is also marked `SERIALIZABLE`. (This restriction does not apply to static data members that are defined as objects, because static data members are not serialized.)

- Handle-based variables are serialized, but no information for reconstructing handle-based objects on the receiving side is serialized.

- `MEMPTRs` assigned by an ABL application are serialized, but `MEMPTRs` from an external source (such as a DLL or shared library) are not serialized.

- Statically defined temp-tables and ProDataSets in user-defined classes are serialized, except for `REFERENCE-ONLY` tables.
The REJECTED, ERROR, ERROR-STRING, and DATA-SOURCE-MODIFIED attributes for temp-tables in an object are maintained as part of the deserialization process. Similarly, the REJECTED and ERROR attributes for ProDataSets are maintained during serialization.

Not all built-in classes are serializable. See the CLASS statement entry in OpenEdge Development: ABL Reference for a full list of serializable built-in classes.

.NET and ABL-extended .NET objects cannot be passed as parameters.

Temp-tables and ProDataSets that contain class-based objects can be passed between AppServer and client as long as the objects also satisfy these restrictions.

Remote procedure behavior

In addition to the behavior specified by the RUN statement options, a remote procedure:

- Executes in the AppServer session as if it were the first procedure executed in the session (top level), using the normal ABL rules for starting and terminating transactions. For more information about the normal ABL rules for procedure execution, see OpenEdge Getting Started: ABL Essentials.

Note: Remote procedures can also participate in automatic transactions, which are unique to AppServer sessions. For information, see the sections on transaction management in Programming the AppServer on page 49.

- Can create other objects in the AppServer session, such as other persistent procedures. These objects are only available locally within the AppServer agent in which they are created. They cannot be directly shared between an ABL client application and the AppServer session.

Understanding proxy procedure handles

When you execute a remote persistent, single-run, or singleton procedure in an ABL client application, two procedure handles are created: one within the client application session and another separate handle within the AppServer session where the remote procedure is created. OpenEdge internally maintains a mapping between the two handles:

- The handle within the client application is a proxy procedure handle, and its PROXY attribute is set to TRUE. The handle also has PERSISTENT, SINGLE-RUN, and SINGLETON attributes. Of these, the one that corresponds to the option used with the RUN statement that called the remote procedure is set to TRUE, and the other two are set to FALSE.

- The corresponding handle within the AppServer agent is a remote procedure handle, and its REMOTE attribute is set to TRUE. The values of its PERSISTENT, SINGLE-RUN, and SINGLETON attributes match those of the proxy procedure handle.

For more information on the relationship between proxy and remote procedure handles, see the information on procedure handles in Design and Implementation Considerations on page 137.

Accessing proxy procedure handles

You can obtain access to the proxy procedure handle in the client using the SET option of the RUN statement, which returns the proxy procedure handle value to the HANDLE variable you specify.
You can also access all proxy procedure handles that are currently active for an AppServer connection by returning the value of the FIRST-PROCEDURE attribute or the LAST-PROCEDURE attribute on the server handle for the connection. You can then use the NEXT-SIBLING or PREV-SIBLING attributes on the resulting procedure handle to navigate the list of active proxy procedure handles.

For example:

```abl
DEFINE VARIABLE hProxy AS HANDLE NO-UNDO.
DEFINE VARIABLE hServer AS HANDLE NO-UNDO.
/* Create server handle for hServer and connect to an AppServer. Call remote persistent procedure on the AppServer. */
hProxy = hServer:FIRST-PROCEDURE. /* First proxy handle */
/* Display the pathname of the file that contains the remote procedure active on the AppServer for each proxy procedure handle. */
DO WHILE NOT hProxy = hServer:LAST-PROCEDURE:
   DISPLAY hProxy:FILE-NAME.
   hProxy = hProxy:NEXT-SIBLING.
END.
DISPLAY hProxy:FILE-NAME. /* Procedure pathname for last proxy handle */
```

### Comparing proxy and local procedure handles

Unlike the values of a procedure handle variable and the THIS-PROCEDURE system handle that reference the same local procedure context, the proxy procedure handle in a client session and the corresponding remote persistent procedure handle in the AppServer agent are truly separate handles. For example, setting the PRIVATE-DATA attribute on a proxy procedure handle has no effect on the PRIVATE-DATA attribute of the corresponding remote procedure handle.

However, note that the OpenEdge mapping between a proxy procedure handle and its corresponding remote procedure handle allows you to execute remote internal procedures and user-defined functions using the proxy handle. For more information, see Running and managing remote procedures on page 107.

### Deleting remote persistent procedures

As with local persistent procedures, a remote persistent procedure context remains active within an AppServer session until it is deleted using the DELETE OBJECT or DELETE PROCEDURE statement. All remote persistent procedures are also deleted when the client application disconnects from the AppServer where the procedure is active (see Disconnecting from an AppServer instance on page 118). You can thus delete a remote persistent procedure from an ABL client session by:

- Deleting its proxy procedure handle using the DELETE OBJECT or DELETE PROCEDURE statement.
- Disconnecting the AppServer by using the DISCONNECT( ) method on the corresponding server handle. This deletes all remote persistent procedures in the AppServer session.

If the delete occurs in the context of another remote procedure request to the AppServer, the deletion is pending until the request completes and returns to the client. When the remote persistent procedure is finally deleted, both its proxy procedure handle on the client and its remote procedure handle in the AppServer agent are deleted together.
In the case of a remote single-run or singleton procedure, DELETE PROCEDURE only deletes the proxy handle on the client. This is because the management of single-run and singleton procedures is handled entirely by the AVM.

Remote procedure code examples

This section presents AppServer-related code examples. The first example presents the code for a non-persistent procedure. The remaining two examples show two different ways to write code for the same remote persistent procedure.

Example 1: Remote non-persistent procedure example

This example shows how to run a remote, non-persistent procedure, checkstk.p, on an AppServer identified as hAppSrv. The TRANSACTION DISTINCT option is used on the RUN statement to specify that the client application's transaction is not propagated to hAppSrv. The two temp-table parameters that are defined, ttOrderLine and ttItem, are passed as INPUT and OUTPUT parameters, respectively, to the checkstk.p procedure, as shown:

```
DEFINE VARIABLE hAppSrv AS HANDLE NO-UNDO.
DEFINE VARIABLE lReturn AS LOGICAL NO-UNDO.
DEFINE TEMP-TABLE ttOrderLine LIKE Order-Line.
DEFINE TEMP-TABLE ttItem LIKE Item.

CREATE SERVER hAppSrv.
lReturn = hAppSrv:CONNECT("-AppService inventory -S 5162 -H zeus",
                       "SMITH", "STARSHIP").
IF NOT lReturn THEN DO:
   DELETE OBJECT hAppSrv NO-ERROR.
   RETURN ERROR "Failed to connect to AppServer".
END.
IF ERROR-STATUS:ERROR THEN DO:
   DELETE OBJECT hAppSrv NO-ERROR.
   RETURN ERROR RETURN-VALUE.
END.
.
RUN checkstk.p ON hAppSrv TRANSACTION DISTINCT
   (Customer.Cust-Num, TABLE ttOrderLine,OUTPUT TABLE ttItem) NO-ERROR.
IF ERROR-STATUS:ERROR THEN DO:
   lReturn = hAppSrv:DISCONNECT().
   DELETE OBJECT hAppSrv NO-ERROR.
   RETURN ERROR RETURN-VALUE.
END.
.
lReturn = hAppSrv:DISCONNECT().
DELETE OBJECT hAppSrv NO-ERROR.
```

When the RUN statement is executed, a remote procedure request to run checkstk.p is sent to the AppServer instance that is connected via the hAppSrv server handle. Once the RUN statement is completed, this code checks to see if checkstk.p completed with an ERROR condition as would occur by executing the RETURN ERROR statement. If checkstk.p did execute the RETURN ERROR statement, then this code also returns with an ERROR condition. By using the RETURN-VALUE function, the code also returns the value returned by checkstk.p.
Example 2: Remote persistent procedure example

The following code example shows how to instantiate a remote persistent procedure, order.p, on an AppServer:

```
DEFINE VARIABLE hAppSrv AS HANDLE NO-UNDO.
DEFINE VARIABLE hOrder AS HANDLE NO-UNDO.
DEFINE VARIABLE iOrderNum AS LIKE Order.Order-Num NO-UNDO.
DEFINE VARIABLE lReturn AS LOGICAL NO-UNDO.

DEFINE TEMP-TABLE ttOrder LIKE Order.
DEFINE TEMP-TABLE ttOrderLine LIKE Order-Line.

CREATE SERVER hAppSrv.
lReturn = hAppSrv:CONNECT("-AppService inventory -S 5162 -H zeus",
"SMITH", "STARSHIP").

IF NOT lReturn THEN DO:
    DELETE OBJECT hAppSrv NO-ERROR.
    RETURN ERROR "Failed to connect to AppServer".
END.
IF ERROR-STATUS:ERROR THEN DO:
    DELETE OBJECT hAppSrv NO-ERROR.
    RETURN ERROR RETURN-VALUE.
END.

RUN order.p ON hAppSrv TRANSACTION DISTINCT PERSISTENT SET hOrder.

RUN GetExistingOrder IN hOrder
    (iOrderNum, OUTPUT TABLE ttOrder, OUTPUT TABLE ttOrderLine) NO-ERROR.
    IF ERROR-STATUS:ERROR THEN DO:
        lReturn = hAppSrv:DISCONNECT().
        DELETE OBJECT hAppSrv NO-ERROR.
        RETURN ERROR RETURN-VALUE.
    END.
    ...
    lReturn = hAppSrv:DISCONNECT().
    DELETE OBJECT hAppSrv NO-ERROR.
```

When the RUN statement for order.p completes, a reference to the persistent procedure context is saved in handle hOrder. The SERVER attribute for hOrder contains a reference to the server handle hAppSrv. When the internal procedure GetExistingOrder is run for persistent procedure hOrder, ABL uses the value of hAppSrv to determine the AppServer instance where the persistent procedure is located.
Example 3: Remote persistent procedure example using the FUNCTION statement

This example shows how a client application invokes a user-defined function, `GetExistingOrder`, that is implemented remotely on an AppServer. It is presented as an alternative way to Example 2. The `FUNCTION GetFirstOrder` statement in the following example replaces the internal procedure `GetExistingOrder` referenced in Example 2:

```abl
DEFINE VARIABLE hAppSrv AS HANDLE NO-UNDO.
DEFINE VARIABLE hOrder AS HANDLE NO-UNDO.
DEFINE VARIABLE iCustNum LIKE Order.Order-Num NO-UNDO.
DEFINE VARIABLE lReturn AS LOGICAL NO-UNDO.
DEFINE TEMP-TABLE ttOrder LIKE Order.
DEFINE TEMP-TABLE ttOrderLine LIKE Order-Line.

FUNCTION GetFirstOrder RETURNS LOGICAL
  (INPUT piCustNum AS INTEGER,
   OUTPUT TABLE ttOrder,
   OUTPUT TABLE ttOrderLine) IN hOrder.

CREATE SERVER hAppSrv.

lReturn = hAppSrv:CONNECT("-AppService inventory -S 5162 -H zeus",
"SMITH", "STARSHIP").
IF NOT lReturn THEN DO:
  DELETE OBJECT hAppSrv NO-ERROR.
  RETURN ERROR "Failed to connect to AppServer".
END.
IF ERROR-STATUS:ERROR THEN DO:
  DELETE OBJECT hAppSrv NO-ERROR.
  RETURN ERROR RETURN-VALUE.
END.
... 
RUN order.p ON hAppSrv TRANSACTION DISTINCT PERSISTENT SET hOrder.
IF NOT GetFirstOrder (INPUT iCustNum, OUTPUT TABLE ttOrder,
OUTPUT TABLE ttOrderLine) THEN DO:
  lReturn = hAppSrv:DISCONNECT().
  DELETE OBJECT hAppSrv NO-ERROR.
  RETURN ERROR.
END.
...
lReturn = hAppSrv:DISCONNECT().
DELETE OBJECT hAppSrv NO-ERROR.
```

Like Example 2, when the `RUN` statement for `order.p` completes, a reference to the persistent procedure context is saved in handle `hOrder`, and the `SERVER` attribute for `hOrder` contains a reference to the server handle `hAppSrv`. However, instead of running an internal procedure, it runs the user-defined function, `GetFirstOrder`.

For more information about creating user-defined functions, see the `FUNCTION Statement` section in *OpenEdge Development: ABL Reference*.

**Note:** If a client application exits without executing the `DISCONNECT()` method on the appropriate handles, OpenEdge automatically disconnects the client from the AppServer. This automatic disconnection can also occur if the AppServer machine loses network connectivity with the client application. This can happen, for example, if there is a network failure.
Example 4: Remote non-persistent procedure with a class-based object as a parameter

This example shows how to run a remote, non-persistent procedure, `updatecredit.p`, on an AppServer identified as `hAppSrv`. The `TRANSACTION DISTINCT` option is used on the `RUN` statement to specify that the client application's transaction is not propagated to `hAppSrv`. The object `rCreditObj` is an instance of class `myObjs.CreditObj`, which has a public data member `CustCredit` of type `DECIMAL` and a public method `ShowCustCredit()` that displays a message with the value of `CustCredit`:

```
USING myObjs.*.
DEFINE VARIABLE hAppSrv AS HANDLE NO-UNDO.
DEFINE VARIABLE lReturn AS LOGICAL NO-UNDO.
DEFINE VARIABLE rCreditObj AS CLASS CreditObj.
CREATE SERVER hAppSrv.
  lReturn = hAppSrv:CONNECT("-AppService inventory -S 5162 -H zeus",
                          "SMITH", "STARSHIP").
  IF NOT lReturn THEN DO:
    DELETE OBJECT hAppSrv NO-ERROR.
    RETURN ERROR "Failed to connect to AppServer".
  END.
  IF ERROR-STATUS:ERROR THEN DO:
    DELETE OBJECT hAppSrv NO-ERROR.
    RETURN ERROR RETURN-VALUE.
  END.
  . . .
  rCreditObj:CustCredit = 0.
  RUN updatecredit.p ON hAppSrv (INPUT-OUTPUT rCustObj).
  rCreditObj:ShowCustCredit().
  CATCH eSysError AS Progress.Lang.SysError:
    MESSAGE eSysError:GetMessage(1).
  END CATCH.
  . . .
  FINALLY:
    lReturn = hAppSrv:DISCONNECT().
    DELETE OBJECT hAppSrv NO-ERROR.
  END.
```

The value of `rCreditObj:CustCredit` is initialized with 0. When the `RUN` statement is executed, a remote procedure request to run `updatecredit.p` is sent to the AppServer instance that is connected via the `hAppSrv` server handle. The object `rCustObj` is passed as an INPUT-OUTPUT parameter. If the `RUN` statement completes without error, the `ShowCustCredit()` method is called on `rCustObj`, which was updated and returned by `updatecredit.p`.

Disconnecting from an AppServer instance

You can disconnect an AppServer instance by executing the `DISCONNECT( )` method on the server object handle for the connection.
Using the DISCONNECT( ) method

Using the DISCONNECT( ) method terminates the connection with the specified AppServer or application service by sending a disconnect request to the connected AppServer or application service. When the AppServer or application service receives the disconnect request, the following occurs:

- **For a state-reset or state-aware AppServer** — Control returns immediately to the client application, and any configured Disconnect procedure executes in the connected AppServer agent.
- **For a stateless AppServer** — Any configured Disconnect procedure executes in an available AppServer agent, then control returns to the client application.
- **For a state-free application service** — The client connection pool is released with no affect on any previously connected AppServers (no Disconnect procedures execute) and client control continues without interruption.
- **For any AppServer with pending asynchronous requests** — All running or pending asynchronous requests are cancelled and the corresponding event procedure is called for each request. The CANCELLED attribute on the asynchronous request handle for all such cancelled requests is set to TRUE. For more information on asynchronous requests, see Managing asynchronous requests on page 122.

Disconnecting from a session-managed AppServer

The following example shows how a client application can disconnect from a session-managed AppServer instance:

```assembler
DEFINE VARIABLE hAppSrv AS HANDLE NO-UNDO.
DEFINE VARIABLE lReturn AS LOGICAL NO-UNDO.
CREATE SERVER hAppSrv.
lReturn = hAppSrv:CONNECT("-AppService inventory -S 5162 -H zeus",
                         "SMITH", "STARSHIP").
IF NOT lReturn THEN DO:
   DELETE OBJECT hAppSrv NO-ERROR.
   RETURN ERROR "Failed to connect to Inventory AppServer".
END.
IF ERROR-STATUS:ERROR THEN DO:
   lReturn = hAppSrv:DISCONNECT().
   DELETE OBJECT hAppSrv NO-ERROR.
   RETURN ERROR RETURN-VALUE.
END.
...lReturn = hAppSrv:DISCONNECT().
DELETE OBJECT hAppSrv NO-ERROR.
```

When the DISCONNECT( ) method executes, the client application disconnects from the AppServer referenced by the hAppSrv handle. When the disconnection request completes, you can send no more remote requests using the disconnected server handle.
**Disconnecting from a session-free application service**

The following example shows how a client application can disconnect from a session-free application service:

```plaintext
DEFINE VARIABLE hInventory AS HANDLE NO-UNDO.
DEFINE VARIABLE lReturn AS LOGICAL NO-UNDO.
CREATE SERVER hInventory.
lReturn = hInventory:CONNECT("-AppService inventory -S 5163 -H 4glServices
   -sessionModel Session-free -initialConnections 25").
IF NOT lReturn THEN DO:
   DELETE OBJECT hInventory NO-ERROR.
   RETURN ERROR "Failed to connect to Inventory application service".
END.
IF ERROR-STATUS:ERROR THEN DO:
   lReturn = hInventory:DISCONNECT().
   DELETE OBJECT hInventory NO-ERROR.
   RETURN ERROR RETURN-VALUE.
END.
```

When the `DISCONNECT()` method executes, the client application disconnects all physical connections to AppServer brokers in the connection pool for the application service referenced by the `hInventory` handle. When the disconnection request completes, you can send no more remote requests using the disconnected server handle.

**Deleting the server handle**

To delete a server object handle, you must first disconnect from the AppServer or application service. Once you have disconnected an AppServer, you can release the associated server object handle by executing the `DELETE OBJECT` statement, as shown in the previous code example.

**Note:** If you attempt to delete the server handle before disconnecting the AppServer, OpenEdge returns an error to the client.

**Handling conditions and return values**

Typically, you pass context across the process barrier between client and AppServer sessions using parameters on remote procedures. Mechanisms for handling conditions and return values provide another way to exchange context between the two process spaces.

**Raising the STOP condition**

An ABL client can raise a STOP condition within a remote procedure when a user presses the STOP key. Raising the STOP condition while a remote procedure is executing causes the STOP condition to be propagated to the appropriate AppServer session where a STOP condition is then raised.
Handling conditions and return values

A STOP condition sent to the remote procedure does not necessarily mean that the remote procedure terminates immediately. ON STOP handlers within the remote procedure might prevent the procedure from actually being terminated. For information on how a remote procedure handles a STOP condition received from the client, see the information on condition handling in Programming the AppServer on page 49.

Handling conditions on the RUN statement

When you execute a synchronous remote procedure, all of the following conditions are handled by OpenEdge exactly as if they were raised within a local procedure call:

• Any condition raised during execution of the synchronous RUN statement that invokes the remote procedure
• Any ERROR condition raised by executing the RETURN ERROR statement in the remote procedure itself

If you use the NO-ERROR option on the RUN statement, you can obtain any information about errors that result from executing the RUN statement using the ERROR-STATUS system handle.

A remote procedure executed persistently returns as a non-persistent procedure when the remote procedure does one of the following:

• Completes with an unhandled STOP or QUIT condition
• Executes a RETURN ERROR statement

For an asynchronous remote procedure, conditions that occur on the RUN statement itself are handled exactly as for a synchronous remote procedure. However, unhandled conditions that occur during execution of the asynchronous remote procedure are propagate back to the client in the context of the event procedure. The result can be referenced within the event procedure using the RETURN-VALUE function, the ERROR-STATUS system handle, and the associated asynchronous request handle (SELF). Any conditions that result from problems with the event procedure itself are handled in the context of the PROCESS EVENTS or blocking I/O statement that executes the event procedure. For more information on handling conditions for asynchronous remote procedures, see Handling the response from an asynchronous request on page 123.

Using the RETURN-VALUE function across sessions

The value of the RETURN-VALUE function in an ABL client session is affected by remote procedure execution. If a remote procedure returns a value by executing the RETURN statement, this value is returned to the client as the value of the RETURN-VALUE function.

Note: The LONGCHAR and MEMPTR data types cannot be returned from a user-defined function across an AppServer.
Managing asynchronous requests

Asynchronous requests allow an ABL client to execute a remote procedure and continue execution before the AppServer has completed (or even begun) executing the remote request. This mechanism enhances client performance, especially in an event-driven application where the user can continue interacting with the client application while the AppServer processes the remote request.

In the following sections, when executing an asynchronous request, ABL places the request on a send queue for the connection until the AppServer can execute the specified remote procedure. Similarly, when the procedure completes execution, ABL places the results on a response queue, where the results wait until the client handles the PROCEDURE-COMPLETE event associated with the request. As a result, all asynchronous requests on a connection are guaranteed to be executed, with results returned, in the order you submit them. For more information on the send and response queues, see AppServer and Client Interaction on page 23.

Executing an asynchronous request

You execute an asynchronous request using the RUN statement with the ASYNCHRONOUS option (the RUN...ASYNCHRONOUS statement). With this option, you can specify a variable to return an asynchronous request handle used to uniquely identify and maintain the status of the asynchronous request. You can also specify an internal procedure to use as the event procedure to handle the response from the asynchronous request. For more information on the RUN...ASYNCHRONOUS statement, see the RUN statement entry in OpenEdge Development: ABL Reference.

When specifying the RUN...ASYNCHRONOUS statement, you can also include INPUT, OUTPUT, and INPUT-OUTPUT parameters similar to the parameters on a synchronous RUN statement. As with the synchronous RUN statement, the input parameters represent values passed to the remote procedure. However, unlike the synchronous RUN statement, the output parameters do not return results from the remote procedure directly on the RUN...ASYNCHRONOUS statement. The values for these output parameters are not available until the asynchronous request completes and the client handles the associated PROCEDURE-COMPLETE event. On detecting this event, the client executes the specified event procedure, which returns the values as input parameters to the event procedure. Thus, the output parameters on the RUN...ASYNCHRONOUS statement serve only as placeholders to map the signature of the remote procedure on the AppServer. For more information, see Handling the response from an asynchronous request on page 123.

Asynchronous request handles

Associated with each asynchronous request is an asynchronous request handle. An instance of this handle is created each time you execute the RUN...ASYNCHRONOUS statement. Thus, you can use this handle to obtain information on a specific asynchronous request.

The ABL client allows you to access this handle in three ways:

- Using the SETasync-request-handle option on the RUN...ASYNCHRONOUS statement.
- Using the FIRST-ASYNC-REQUEST and LAST-ASYNC-REQUEST attributes of the server handle or the NEXT-SIBLING and PREV-SIBLING attributes of the asynchronous request handle to walk through the valid asynchronous request handles for the specified AppServer.
- Accessing the SELF system handle within an event procedure in response to the asynchronous request.
For more information on using the asynchronous request handle, see Handling the response from an asynchronous request on page 123.

Handling the response from an asynchronous request

When an asynchronous request completes execution, it sends a response to the AppServer client, which places it on the response queue for the appropriate server handle. To signify that the response is ready to be processed, the PROCEDURE-COMPLETE event is placed on the event queue where it can be processed in the context of blocking I/O statements, such as WAIT-FOR or the PROCESS EVENTS statement.

PROCEDURE-COMPLETE events

The PROCEDURE-COMPLETE event is an event on the asynchronous request handle. It indicates that the asynchronous request associated with the handle has completed execution, and that the corresponding event procedure can be executed. ABL executes the event procedure in the context of a blocking I/O statement much like it executes the trigger for a user-interface event.

Note that you do not have to specify the PROCEDURE-COMPLETE event explicitly in your client code. ABL processes the event like any other ABL event. You also do not have to define a trigger for the event (for example, using an ON statement) because you specify an event procedure in the RUN...ASYNCRONOUS statement that defines the “trigger code” for the event.

To process a PROCEDURE-COMPLETE event for a particular asynchronous request handle, ABL:

- Decrements the ASYNC-REQUEST-COUNT attribute on the following handles:
  - The server handle contained in the SERVER attribute of the asynchronous request handle
  - The procedure handle contained in the PERSISTENT-PROCEDURE attribute of the asynchronous request handle, if the PERSISTENT-PROCEDURE attribute refers to a valid persistent procedure
- Sets the COMPLETE attribute for the asynchronous request handle to TRUE
- Sets the STOP, QUIT, or ERROR attribute for the asynchronous request handle appropriately as indicated by the response message from the AppServer, and stores any error information returned from the AppServer for the request in the ERROR-STATUS system handle
- Sets the return value for the RETURN-VALUE function, if a return value was returned by the AppServer
- Attempts to execute the event procedure specified by the EVENT-PROCEDURE and the EVENT-PROCEDURE-CONTEXT attributes for the asynchronous request handle, if EVENT-PROCEDURE is not the empty string (""), and:
  - Sets each INPUT parameter for the event procedure to the Unknown value (?) or, if the parameter is a TEMP-TABLE parameter, the temp-table remains unchanged, if the response message indicates that the remote request finished with a STOP, ERROR, or QUIT condition
  - Sets the INPUT parameter values for the event procedure to the OUTPUT and INPUT-OUTPUT parameter values returned by the remote procedure, if the response message indicates that the remote request completed successfully
  - Displays an error message, if a specified event procedure fails to execute for any reason
Event procedures

To handle the response from an asynchronous request, you must define an event procedure. In this context, an event procedure is an internal procedure that executes in response to a PROCE- COMPLETE event. To define an event procedure you must:

- Specify the name of the event procedure using the EVENT-PROCEDURE option on the RUN...ASYNCHRONOUS statement that executes the request.
- Define an internal procedure with the same name as the event procedure specified in the RUN...ASYNCHRONOUS statement.

Typical places where you might define the specified internal procedure include:

- The same procedure context as the RUN...ASYNCHRONOUS statement that specifies the event procedure.
- An external procedure context that you specify using the EVENT-PROCEDURE option on the RUN...ASYNCHRONOUS statement using its procedure handle. In this case, the external procedure context must be active in the client session before you execute the RUN...ASYNCHRONOUS statement.

You must define the specified internal procedure with INPUT parameters that correspond in type and order to the OUTPUT and INPUT-OUTPUT parameters that you specify on the RUN...ASYNCHRONOUS statement.

To receive results from the asynchronous request in an event procedure, you can access:

- The asynchronous request handle for the request using the SELF system handle
- The attributes and methods of the ERROR-STATUS system handle to obtain error information from the request
- Values passed as output parameters from the remote procedure to corresponding input parameters in the event procedure
- The RETURN-VALUE function to get the return value from the request

Obtaining error information

With an asynchronous remote procedure request, the RUN...ASYNCHRONOUS statement returns once the client request has been validated. Any errors that occur while processing the RUN...ASYNCHRONOUS statement are returned as a STOP or ERROR condition in the context of the RUN...ASYNCHRONOUS statement. Information on errors that occur once the asynchronous request has been submitted can be obtained by using the ERROR-STATUS system handle within the event procedure for the asynchronous request.

If the asynchronous request completes with a condition, you can use attributes on the asynchronous request handle to determine the condition that the request completed with. For example:

- ERROR attribute indicates if the request completed with an ERROR condition
- STOP attribute indicates if the request completed with a STOP condition
- QUIT attribute indicates if the request completed with a QUIT condition

Note that any asynchronous request that raises the STOP condition (by executing the STOP statement), and does not handle the condition itself, also sets the STOP attribute of its corresponding asynchronous request handle to TRUE.
If any errors occur when the event procedure attempts to execute, The AVM (ABL Virtual Machine) displays an error message in the context of the blocking I/O statement, and no information is stored in the asynchronous request handle. Errors where this might occur include, for example, an event procedure whose \textit{INPUT} parameters do not match in order and type with the output parameters from the remote request.

\section*{Obtaining parameter values}

As explained earlier, the values returned for \textit{OUTPUT} or \textit{INPUT-OUTPUT} parameters on an asynchronous remote procedure are not returned to the \texttt{RUN} statement as for a synchronous remote procedure. Instead, they are returned in the corresponding \textit{INPUT} parameters of the specified event procedure.

Note that \textit{INPUT} parameters in an event procedure observe the same rules of definition and scoping as any internal procedure, as shown:


define variable hAsync as handle no-undo.
define variable iCount as integer no-undo.

define temp-table ttOrder ...
define button bCheckOrders.
create server hAppSrv.
hAppSrv:CONNECT().
run order.p on server hAppSrv asynchronous set hAsync event-procedure GetOrderRecords in this-procedure (input-output table ttOrder, output iCount).
display iCount. /* Displays 0 */
on choose of bCheckOrders
  if iCount GT 0 THEN
    message "You have" iCount "orders." view-as alert-box
  else
    message "Orders are pending ..." view-as alert-box.
  . . .
wait-for window-close of current-window.
procedure GetOrderRecords:
define input parameter table for ttOrder.
define input parameter piOrderCount as integer.

  if self:error or self:stop then /* Handle condition */
    else iCount = piOrderCount.
  return.
end.

Thus, in this example, the \texttt{ttOrder} temp-table (defined in the outer block) receives its content directly from the \texttt{ttOrder INPUT} parameter of the \texttt{GetOrderRecords} event procedure. However, you must explicitly assign the \texttt{iCount} variable (defined in the outer block) to the \texttt{piOrderCount} parameter in order to make the parameter value available to the outer block. (That is, if you define the \texttt{INPUT} parameter as \texttt{iCount} instead of \texttt{piOrderCount}, ABL does not automatically make the parameter value available as \texttt{iCount} in the outer block. Only the \texttt{TABLE} parameter is scoped to the outer block and thus receives its value from the parameter.)

Note also that the event procedure checks for and handles any unhandled conditions generated from the execution of \texttt{order.p} using AppServer error-handling mechanisms.
Obtaining the return value

You can obtain the return value for a remote procedure that you call asynchronously by invoking the `RETURN-VALUE` function within the event procedure that handles the asynchronous request. This function thus returns the value specified by a corresponding `RETURN` statement executed in the remote procedure. As for a synchronous request, if the remote procedure returns no value in the `RETURN` statement, the `RETURN-VALUE` function returns the empty string ("").

Cancelling asynchronous requests

You can cancel all asynchronous requests that are executing or pending on behalf of the client on a particular AppServer by executing the `CANCEL-REQUESTS()` method on the server object handle. This method raises the `STOP` condition in the context of the asynchronous procedure currently executing and causes results to be returned for any pending asynchronous requests, as described in the following paragraph.

The associated event procedures execute for all cancelled requests the next time the client blocks for I/O or executes the `PROCESS EVENTS` statement. Each event procedure receives the following results, depending on the final state of its corresponding asynchronous request:

- **Any request that is completed at the time of cancellation but whose event procedure has not yet run** — The event procedure receives all input parameters passed from the request, and the `COMPLETE` attribute on the asynchronous request handle (SELF) is set to `TRUE`. This is the same result as a normal asynchronous request completion.

- **The request that is executing and that is stopped in response to the `CANCEL-REQUESTS()` method** — The event procedure input parameters are set to Unknown value (?), and the `COMPLETE` attribute on the asynchronous request handle (SELF) is set to `TRUE`. If the stopped request does not handle the `STOP` condition raised by the `CANCEL-REQUESTS()` method, the `STOP` attribute on the asynchronous request handle (SELF) is also set to `TRUE`. This is the same result as if the `STOP` condition were raised on the AppServer agent running the request.

- **Any request that is removed from the send queue prior to its execution** — The event procedure input parameters are set to the Unknown value (?) (or unchanged for `TEMP-TABLE` parameters), and the `CANCELLED` attribute on the asynchronous request handle (SELF) is set to `TRUE`. This result can only occur for a request that is cancelled while waiting for execution.

**Note:** If you disconnect an AppServer using the `DISCONNECT()` method, this also cancels all asynchronous requests still running or pending on the AppServer. For more information, see **Disconnecting from an AppServer instance** on page 118.
Cancelling asynchronous requests after a specified time limit

The CANCEL-REQUESTS-AFTER( ) method lets you specify the number of seconds to wait before calling the CANCEL-REQUESTS( ) method on the server object. This feature allows you to prevent calls from taking too much time when the call must be completed in a timely manner. When called, the CANCEL-REQUESTS-AFTER( ) method all requests currently running or queued to run on the server object will be canceled after the specified number of seconds have elapsed, regardless of when they were started.

If the method is called on a server object that already has a timer running, the timer is restarted with the new time interval, measured from that moment.

If the method is called with a parameter whose value is less than or equal to zero, then the timer on that server object is stopped.

Calling this method on a server object has no immediate effect on any asynchronous requests that are currently running (or queued to run) on that server object. Moreover, the length of time these requests have already been running prior to the call does not effect the timer.

The method may be called before or after the asynchronous requests are run. Any asynchronous requests that are run after the method is called must complete before the timer expires. The time available to complete such a request depends on when it was executed, with respect to the CANCEL-REQUESTS-AFTER( ) method call.

The timer has no effect on requests run synchronously during the time interval. Synchronous requests are not canceled, even if they run longer than the specified timeout.

Manually calling the CANCEL-REQUESTS( ) method in the application program during the timeout period has no effect on the timer.

Examples

The following simple example uses the CANCEL-REQUESTS-AFTER( ) method:

```plaintext
DEFINE VARIABLE hSrv AS HANDLE NO-UNDO.
DEFINE VARIABLE bool AS LOGICAL NO-UNDO.
CREATE SERVER hSrv.
hSrv:CONNECT("-URL AppServerDC://hostName/svc").
bool = hSrv:CANCEL-REQUESTS-AFTER(30).
RUN foo.p ON SERVER hSrv ASYNCHRONOUS ("Hello World").
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```
The following is a server program used in the previous example to demonstrate the `STOP-AFTER` phrase:

```abl
/* srvrNested3.p */
DEFINE INPUT-OUTPUT PARAMETER p1 AS CHARACTER NO-UNDO.
DEFINE VARIABLE srvrTimeLimit AS INTEGER NO-UNDO INITIAL 10.
DEFINE VARIABLE spinLimit AS INT64 NO-UNDO INITIAL 15000.

p1 = FILL("Y", 30). /* 30 "Y" characters will be sent to the client */
DO STOP-AFTER srvrTimeLimit:
    RUN spinHere (spinLimit).
END.
RETURN "30 Ys".

PROCEDURE spinHere:
    DEFINE INPUT PARAMETER spinLimit AS INT64 NO-UNDO.
    DEFINE VARIABLE loopFlag AS LOGICAL NO-UNDO.
    DEFINE VARIABLE endTime AS INT64 NO-UNDO.
    ASSIGN
        loopFlag = TRUE
        endTime = ETIME(FALSE) + spinLimit.
    DO WHILE loopFlag:
        IF (ETIME(FALSE) > endTime) THEN
            loopFlag = FALSE.
        END.
    END.
END PROCEDURE.
```

The following is a new example of a client program that asynchronously invokes the server program (srvrNested3.p):

```abl
/* clntNested4.p */
DEFINE BUTTON send-button LABEL "SEND".
DEFINE BUTTON close-button LABEL "CLOSE".
DEFINE VARIABLE ah AS HANDLE NO-UNDO.
DEFINE VARIABLE cstr AS CHARACTER NO-UNDO.
DEFINE VARIABLE rcvcnt AS INTEGER NO-UNDO.
DEFINE VARIABLE sh AS HANDLE NO-UNDO.
DEFINE VARIABLE S1 AS CHARACTER NO-UNDO FORMAT "X(30)".
DEFINE VARIABLE S2 AS CHARACTER NO-UNDO FORMAT "X(30)".
DEFINE VARIABLE xmtcnt AS INTEGER NO-UNDO.

/* ============================================================== */
FORM send-button close-button
    WITH FRAME buttonFrame.

DEF FRAME foo
    S1 AT 1 WITH NO-LABELS.

DEF FRAME bar
    S2 AT 1 WITH NO-LABELS.

/* ============================================================== */
ON CHOOSE OF send-button DO:
    RUN runRemoteProc.
    S1 = "Ran proc(" + STRING(xmtcnt) + ")".
    DISPLAY S1 WITH FRAME foo 1 DOWN.
    HIDE FRAME bar.
END.
```

OpenEdge Application Server: Developing AppServer Applications

Chapter 3: Programming ABL Client Applications
ON CHOOSE OF close-button DO:
  sh:DISCONNECT().
  DELETE OBJECT sh.
  QUIT.
END.

ON WINDOW-CLOSE OF CURRENT-WINDOW DO:
  sh:DISCONNECT().
  DELETE OBJECT sh.
  QUIT.
END.

ENABLE send-button close-button WITH FRAME buttonFrame.

CREATE SERVER sh.
  cstr = "-URL AppServerDC://hostname:portnum/svc".
  sh:CONNECT(cstr).

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW OR CHOOSE OF close-button.

  sh:DISCONNECT().
  DELETE OBJECT sh.

/* =============================================================== */

PROCEDURE runRemoteProc:
  DEFINE VARIABLE ix AS CHARACTER NO-UNDO.
  ASSIGN
    xmtcnt = xmtcnt + 1
    ix = FILL("X", 30).
  sh:CANCEL-REQUESTS-AFTER(10).
  DO STOP-AFTER 5:
    RUN srvrNested3.p ON SERVER sh ASYNCHRONOUS SET ah
      EVENT-PROCEDURE "remoteProcHandler"
      IN THIS-PROCEDURE (INPUT-OUTPUT ix).
  END.
END PROCEDURE.

/* =============================================================== */

PROCEDURE remoteProcHandler:
  DEFINE INPUT PARAMETER ix AS CHARACTER NO-UNDO.
  ASSIGN
    rcvcnt = rcvcnt + 1
    S2 = "remoteProcHandler(" + STRING(rcvcnt)
    S2 = S2 + IF (SELF:STOP) THEN ") STOPPED"
      ELSE IF (SELF:CANCELLED) THEN ") CANCELLED"
      ELSE ") FINISHED".
  DISPLAY S2 WITH FRAME bar 1 DOWN.
  DELETE OBJECT SELF.
END PROCEDURE.

With this local procedure, the CANCEL-REQUESTS-AFTER( ) method is called on the server handle to establish a time limit of 10 seconds for remote asynchronous procedures; if the timer is already active, the time limit is reset to 10 seconds. The srvrNested3.p is run asynchronously on the AppServer from within a block that specifies a time limit of 5 seconds. The remote procedure does not implicitly adopt the 5 second time limit of the local procedure (as would be the case for running a synchronous remote procedure). In this example, the local time limit of 5 seconds is essentially irrelevant, since it only times the RUN statement. Since this is an asynchronous RUN, it returns immediately after sending the request.

If the response to the request returns before 10 seconds has elapsed, the event procedure remoteProcHandler is run.
If the time limit is exceeded before the response returns, the client AVM will call the `CANCEL-REQUESTS( )` method on the server handle. This will cause a `STOP` message to be sent to the AppServer running the asynchronous request. As with remote synchronous requests, this will cause the `STOP` condition to be raised in the server application. Depending on the `STOP` handling logic in the server application, the `STOP` condition may be returned to the client application in the response. This is indicated by the `STOP` attribute in the asynchronous request handle, which is set to `TRUE` prior to running the event procedure.

It is important to note that the time limit for the asynchronous remote procedure call restarts each time the `CANCEL-REQUESTS-AFTER( )` method is called. This effectively extends the time limit for any asynchronous requests that have already been run (or queued to run) on that server handle.

**Note:** Another way to time limit procedure calls is to set the `ubroker` property called `srvrExecutionTimeLimit` in the `[UBroker]` section of the `ubroker.properties` file. This is a non-negative integer property that specifies the maximum time in seconds that a remote procedure may execute on the given AppServer. This timeout value applies to all remote procedures that execute on the AppServer.

---

**Asynchronous requests and persistent procedures**

Client applications can asynchronously instantiate a persistent procedure and then execute an internal procedure on the persistent procedure before the remote persistent procedure has actually been created on the AppServer. When the `RUN...ASYNCHRONOUS` statement is executed for the purposes of creating the remote persistent procedure, a proxy procedure handle is returned to the client application.

The client application can then immediately use the proxy procedure handle to run an internal procedure in the remote persistent procedure. If the asynchronous request to instantiate the remote persistent procedure has completed, you can run the internal procedure synchronously or asynchronously. If it has not completed, you can only run the internal procedure asynchronously.

Because both the request to instantiate the persistent procedure and the asynchronous request to execute the internal procedure are executing on the same connection, ABL guarantees that the request to instantiate the persistent procedure executes before the request to run the internal procedure. If the request to instantiate the persistent procedure ultimately fails, the request to run the internal procedure will ultimately fail as well, because the proxy procedure handle used to run the internal procedure is invalid. When the internal procedure request fails, The AVM sets the `ERROR` attribute to `TRUE` on both the asynchronous request handle and the `ERROR-STATUS` system handle, indicating that an invalid proxy procedure handle was used to execute the internal procedure.

---

**Deleting asynchronous remote persistent procedures**

You can delete a remote persistent procedure that you have executed asynchronously just like one you have executed synchronously, using the `DELETE OBJECT` or `DELETE PROCEDURE` statement. However, for an asynchronous procedure, the delete request raises an `ERROR` condition if the `ASYNC-REQUEST-COUNT` attribute on the proxy procedure handle is greater than zero (0). If `ASYNC-REQUEST-COUNT` is zero (0), the local proxy procedure handle is deleted immediately.

If an asynchronous request for another procedure is executing in the AppServer agent at the time of the delete request, the delete request is queued on the AppServer until the AppServer agent becomes available to delete the remote persistent procedure.
Deleting server handles

You can delete a server handle using the `DELETE OBJECT` statement. However, the delete request raises an `ERROR` condition if the `ASYNC-REQUEST-COUNT` attribute on the server handle is not zero (0).

Deleting asynchronous request handles

You can delete an asynchronous request handle using the `DELETE OBJECT` statement. This frees all resources (memory) used by this handle. The recommended technique is to delete the handle at the end of the event procedure using the `SELF` handle (that is, `DELETE OBJECT SELF`).

Debugging asynchronous requests

The AppServer supports two models for debugging ABL code, the remote and distributed debugging models. An asynchronous request can only be debugged using the remote debugging model. The distributed debugging model can only be used for debugging synchronous requests. For more information on the AppServer debugging models, see Debugging on page 179.

Mixing synchronous and asynchronous requests

If you attempt to execute a synchronous request on an AppServer connection with outstanding asynchronous requests, The AVM raises the `ERROR` condition. You can determine if there are any outstanding asynchronous requests for an AppServer by checking the value of the `ASYNC-REQUEST-COUNT` attribute of the appropriate server handle.

You can also ensure that all asynchronous requests for a given server handle have completed execution by the way you set up your event management in the client. Specify the `PROCEDURE-COMPLETE` event in the `WAIT-FOR` statement for the asynchronous request handle that corresponds to the last asynchronous request submitted before the `WAIT-FOR` statement is executed. For example, assume that you submit three asynchronous requests for some server handle and that the asynchronous request handles are returned in `HANDLE` variables `hdl1`, `hdl2`, and `hdl3`. To ensure that there are no outstanding asynchronous requests before submitting a synchronous request, execute the `WAIT-FOR` statement specifying the `PROCEDURE-COMPLETE` event for `hdl3`.

Running asynchronous requests on the SESSION system handle

You can execute a `RUN...ASYNCHRONOUS` statement using a server object handle or a `SESSION` system handle. If you use a `SESSION` handle, the request is executed synchronously on the client as if it was called using a synchronous `RUN` statement. However, unlike a standard synchronous procedure call that returns the results directly to the `RUN` statement, the results are returned by the event procedure specified on the `RUN...ASYNCHRONOUS` statement as though it were responding to a `PROCEDURE-COMPLETE` event from an AppServer.
Thus, using the `SESSION` handle, the event procedure runs synchronously as part of the `RUN` statement and returns execution to the statement immediately following the same `RUN` statement. This is different from using a server handle, where the event procedure runs only in the context of a blocking I/O or `PROCESS EVENTS` statement that processes the `PROCEDURE-COMPLETE` event. However, this still allows you to use the `RUN...ASYNCHRONOUS` statement to program local and remote requests in a similar way, for example, when an AppServer is not available.

**Note:** In addition to order of execution, using the `SESSION` handle to run an asynchronous request as a local synchronous request results in error handling that is slightly different from using a server handle. For this reason, you might want to develop using a local AppServer when a network AppServer is not available. For more information on running a local AppServer, see *OpenEdge Application Server: Administration*. For more information on error handling when using the `SESSION` handle, see the `RUN` statement entry in *OpenEdge Development: ABL Reference*.

### Examples

The following examples show simple implementations of various client and remote procedures. The first two examples (Figure 1 and Figure 2) compare two similar implementations, one synchronous and the other asynchronous. These are non-executing examples, shortened for illustration. The third example (`client.p` and `server.p`) shows a fully executing, asynchronous implementation of client and remote procedures.

#### Synchronous request execution model

The following figure shows a remote procedure executed synchronously. The application sets a simple status flag (Done) to indicate that the remote request has completed.

As an event-driven GUI application, the client first executes a `WAIT-FOR` statement after having set up code for handing events (1). When the client executes `sync.p` as a `synchronous remote procedure` (2), the client blocks while the AppServer completes the request (3) and only continues execution after the remote procedure completes (4). Results from a synchronous remote procedure call are returned to the client immediately after the point of invocation (4), exactly as they are returned from a local procedure call.

Note that in this example all of the remote processing and handling of the result occurs in the trigger for `bStatus`. While straight forward to implement, the client can block for an indeterminate period of time at the synchronous `RUN` statement waiting for `sync.p` to complete, depending on the application. Asynchronous requests allow a client to avoid this bottleneck.
Asynchronous request execution model

The following figure shows a remote procedure executed asynchronously. As in Figure 1, the client is an event-driven GUI and the application sets a simple status flag (Done) to indicate that the remote request has completed.
In this implementation, the client first executes `async.p` as an asynchronous remote procedure (1, specified by the `ASYNCHRONOUS` keyword on the `RUN` statement). The client immediately continues execution, until it reaches the `WAIT-FOR` statement to get events (2) at the same time that the AppServer executes the remote request. Thus, at this point (1 and 2), the client and remote request are running in parallel. The client can continue to execute, calling additional asynchronous remote procedures on the same or different AppServer connections.

As each asynchronous request completes (like `async.p` at 3), the client handles the results in a specified event procedure (4). This event procedure, specified in the asynchronous `RUN` statement, is essentially a “trigger” that executes on the client in response to a `PROCEDURE-COMPLETE` event posted to the client when the associated asynchronous request completes.
As with user-interface events, the client can handle `PROCEDURE-COMPLETE` events in the context of a blocking I/O statement, such as the `WAIT-FOR` statement (4), or by executing the `PROCESS EVENTS` statement. The AVM maps the `PROCEDURE-COMPLETE` event for each asynchronous request to the appropriate event procedure using a unique asynchronous request handle that is generated for each request (not shown). This handle provides the mechanism that you can use to monitor the status of each request.

Note that the example in the previous figure provides the same functionality as Figure 1. In fact, `sync.p` and `async.p` are identical, except for their names, which are changed for readability. The AppServer has no idea whether its procedures are being called synchronously or asynchronously. The type of request is entirely a function of the client and its implementation.

The main difference, is that the `bStatus` trigger in the previous figure tests the status of the remote request and performs actions based on whether the request finished, rather than invoking the request itself, as in Figure 1. Thus, the synchronous bottleneck is removed. In all such cases, the synchronous bottleneck is avoided by handling the results asynchronously.

**A complete asynchronous request example**

In the following example, the ABL client procedure, `client.p`, calls `server.p` (following) asynchronously on an AppServer. The `server.p` procedure returns the customer number corresponding to a customer name provided by `client.p`.

The `client.p` procedure blocks and waits for the `PROCEDURE-COMPLETE` event that triggers execution of the `GetCustNum` event procedure. The `GetCustNum` procedure displays the result and deletes the corresponding asynchronous request handle (referenced by the `SELF` system handle).

In a more typical example, additional event handlers (triggers) would be active for user-interface events, allowing the user of `client.p` to perform other tasks with the procedure while completion of the request is pending. In this case, the client application blocks immediately after submitting the request to wait for the result and terminate on the `PROCEDURE-COMPLETE` event.

**client.p**

```abl
define variable cCustomer AS CHARACTER NO-UNDO INITIAL "WILNER".
define variable hRequest AS HANDLE NO-UNDO.
define variable hServer AS HANDLE NO-UNDO.
define variable lReturn AS LOGICAL NO-UNDO.
create server hServer.
lReturn = hServer:CONNECT("-S 5162 -H zeus -AppService Inventory").
run server.p on hServer ASYNCHRONOUS SET hRequest
   event-procedure "GetCustNum"
      (input cCustomer, output iCustomer AS INTEGER).
wait-for procedure-complete of hRequest.
delete object hRequest NO-ERROR.
hServer:DISCONNECT().
delete object hServer NO-ERROR.
procedure GetCustNum:
   define input parameter piCustomer AS INTEGER NO-UNDO.
       display piCustomer.
end.
```
server.p

DEFINE INPUT PARAMETER pcCustomer AS CHARACTER NO-UNDO.
DEFINE OUTPUT PARAMETER piCustomer AS INTEGER NO-UNDO.

FOR FIRST customer WHERE customer.name = pcCustomer NO-LOCK:
  piCustomer = customer.cust-num.
END.
Design and Implementation Considerations

This chapter highlights specific performance, transaction management, deployment, and security internationalization issues related to planning and implementing a distributed application with AppServer. It explains some programming techniques, especially those commonly used by both ABL client and AppServer procedures, to help you write distributed applications.

For details, see the following topics:

• Primary performance considerations
• Secondary performance considerations
• Transaction and record management considerations
• Deployment considerations
• Security considerations
• Multi-language support
• Distributed application design and implementation

Primary performance considerations

There are two fundamental performance reasons for implementing an application using AppServer:

• To minimize network traffic by putting data processing closer to the database
• To offload resource-intensive processing tasks to a machine more capable of executing the tasks
These two points can work together to provide maximum performance benefits. The following sections discuss each point separately.

**Minimizing network traffic**

When you deploy an AppServer on the database server machine and connect to a database using a self-service connection, you have the opportunity to minimize network traffic.

In a traditional client/server model, table joins and application-specific record filtering are done by the ABL client, implying that many records must be passed over the network. By using an AppServer, all this processing can be completed local to the database and network traffic is restricted to the final results being returned to the client application.

Using the AppServer, you can provide a remote procedure that returns a temp-table that contains a single record for each sales representative along with each representative’s associated statistical information. The procedure that you write and is executed within the AppServer uses the same query that was used when the application was executed using the traditional client/server model. However, using the AppServer, the query was executed using a self-service connection to the database. Also, your procedure performs all other record processing and filtering that is necessary to reduce the data to the actual results required by the client application. The end result of transforming your application from the traditional client/server model to a model that takes advantage of the AppServer is that one record is sent over the network for each sales representative rather than several.

For example, consider a sales representative application. One of the application's activities might be to accumulate a set of statistics for each sales representative. These statistics might include the number of orders, the number of customers, and the average order value. In the traditional client/server model, all the records for all sales representatives are passed back to the client application where the statistical information is generated.

**Off loading resource-intensive processing tasks**

You can additionally obtain performance gains using the AppServer by offloading resource-intensive processing to a machine that is separate from the client machine and more capable of executing these tasks. For example, performance gains might be obtained by offloading tasks, such as complex mathematical algorithms or activity associated with transaction processing to a database server machine. These types of performance gains can be achieved with the AppServer.

**Choosing an AppServer operating mode**

You can configure an AppServer to run in one of four operating modes:

- State-reset
- State-aware
- Stateless
- State-free

In addition, stateless operating mode supports two connection states, bound and unbound, that are determined by your AppServer application at run time. State-free operating mode also provides support for bound connections, but PSC recommends that you avoid them. For information on how each operating mode affects the operation of an AppServer installation, see *AppServer and Client Interaction* on page 23.
Application characteristics to consider

To understand the trade-offs between these operating modes, you must first determine the goals of your application. In general, each operating mode offers different trade-offs among three general characteristics:

- Design complexity
- Resource consumption
- Performance (request throughput and response time)

The best operating mode for your application depends on a number of factors, including:

- The complexity of the application
- The amount of context that needs to be maintained between requests
- The current and future performance goals for the application
- The computer resources available for the AppServer machine

In some cases, it might be significantly more complex to program your AppServer application with one operating mode rather than another. In many cases there might be more than one operating mode and design strategy that is appropriate. The purpose of this section is to explain some of the trade-offs associated with each operating mode so you can determine the most appropriate operating mode and design strategy to use for your situation.

The following table lists each operating mode, comparing it in terms of the three general characteristics with performance divided between throughput and response time. These comparisons apply, for each operating mode, in a network where available AppServer resources are the same.

<table>
<thead>
<tr>
<th>Table 8: Characteristic operating modetrade-offs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating mode</td>
</tr>
<tr>
<td>State-reset</td>
</tr>
<tr>
<td>State-aware</td>
</tr>
<tr>
<td>Operating mode</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Stateless</td>
</tr>
<tr>
<td>State-free</td>
</tr>
</tbody>
</table>

**Managing larger amounts of context**

If your application has a large amount of context to manage between each request, then state-reset or state-aware are appropriate choices. Stateless is not a good choice because of the complexity and performance impact of managing this connection context through the Activate and Deactivate procedures for each request.
If there is a large amount of context that you need to establish for each connection that is independent of the user who is running the client application (for example, loading a temp-table with static data), and only a small portion of the context is user specific, then state-aware might be a better choice than state-reset. This is true because with state-aware, you only need to establish the connection-independent context once, in the Startup procedure. You generally establish the user-specific context within the Connect procedure, using the user ID and password information provided by the client application.

The Startup procedure is not available for state-reset. Thus, both the connection-independent context and the user-specific context have to be established within the Connect procedure. Under these conditions, the time it takes to connect to a state-aware AppServer can be significantly shorter than the time it takes to connect to a state-reset AppServer because all of the connection-independent context is established before the client actually attempts to connect to the AppServer.

**Managing smaller amounts of context**

If there is a minimal amount of context that needs to be maintained between each request and you have limited computing resources that need to service a large client load, the stateless operating mode might be the appropriate choice for your application. This increases your ability to service a large number of clients, because no AppServer agent is dedicated to any particular client. The minimal context that needs to be maintained between requests can be stored either in the `SERVER-CONNECTION-CONTEXT` attribute or within an application-specific context database. No matter where you store the context, you probably want to use the Activate and Deactivate procedures to manage that context for each request.

**Limiting bound connections**

However, you need to use bound connections on a stateless AppServer with care. If client connections to a stateless AppServer are bound the majority of the time, the stateless operating mode is not a good choice, especially under heavy client load. In this case, the stateless AppServer is effectively running much like a state-aware or state-reset AppServer where each AppServer agent remains effectively dedicated to one client for the duration of the bound connection.

In some cases, if most of the context that you need to maintain between requests is minimal, but occasionally you need to maintain a large amount of context for a small portion of the application, then stateless might still be an appropriate choice. In that case, you can temporarily make the connection bound by having the client run a remote persistent procedure and delete shortly after, or by setting the `SERVER-CONNECTION-BOUND` request attribute to `TRUE` within the AppServer session and setting it to `FALSE` shortly after.

**Note:** You might want to run all of your requests on a bound connection, if you use a *briefly-bound* approach. In a briefly-bound approach, you reserve all context maintenance to a few requests on the bound connection during processing that is free of user interaction or other client-blocking activity. Then, you make the connection unbound before performing any blocking or other intensive activity (think time) on the client. With briefly-bound connections, the AppServer remains free to service other clients most of the time and with fewer AppServer agent required.
In addition, all requests for a stateless AppServer go through the AppServer broker. Thus, two messages are sent for each request. One message travels between the client and the AppServer broker and the other message travels between the AppServer broker and the AppServer agent. However, because a state-aware or state-reset AppServer maintains a direct connection between the client and the AppServer agent, only one message is sent per request. Under these conditions, your only choice, to both maintain AppServer context and increase the response of the AppServer to client connections, might be to use state-aware or state-reset and to add additional computing resources at the AppServer site to sustain the client connections.

**Considering ample computer resources**

Note that for the most part, using stateless, and especially state-free, makes the best sense when you must provide high throughput with minimal computing resources. Even if you only have minimal context that needs to be maintained between requests, if you have plenty of computing resources relative to your current and expected client load, then there is no inherent advantage to using stateless. Under these more favorable conditions, even if there is only minimal context to manage, using state-aware (or state-reset) might provide a somewhat simpler programming model.

**Considering the future**

When considering whether to use stateless, be sure to consider your current and future performance goals. Redesigning a state-aware or state-reset application to be stateless or state-free can be very complicated. Even if your current goals do not require stateless or state-free operating modes, one of these modes might still be the right choice based on your future performance requirements. Note especially that state-free is the most scalable operating mode of all.

**Setting the number of AppServer agents**

For best performance, you must determine the optimal settings for the server pool parameters (Pool Range subcategory of the Agent parameters) in the OpenEdge Explorer or OpenEdge Management:

- **Initial number of servers to start** *(initialSrvrInstance property in ubroker.properties)*
- **Minimum servers** *(minSrvrInstance property in ubroker.properties)*
- **Maximum servers** *(maxSrvrInstance property in ubroker.properties)*

In general, you must consider the number of clients, the design of your application (including the AppServer operating mode), and the hardware resources that run your AppServer. At a minimum, follow these guidelines in setting these values:

- **For a state-aware or state-reset AppServer** — You must have one AppServer agent for each user that connects to the AppServer. That is, you need as many AppServer agents as users who connect concurrently to the AppServer.

- **For a stateless or state-free AppServer** — You can have one AppServer agent work on requests for multiple users. At a minimum, Progress Software Corp. recommends one AppServer agent for each CPU in the machine. You can expect each AppServer agent to handle requests from up to 20 AppServer clients (or more). This is your per-agent client load. You can then scale up the number of agents to support the required multiple of per-agent client load to match your total AppServer client load. The per-agent client load also depends on the length of time it takes to complete an average client request. The longer it takes for an agent to complete client requests, the fewer clients are supported by each agent, and the more agents you need.
to handle the total AppServer client load. Thus, the total number of agents required is very application specific.

For more information on setting the server pool parameters, see *OpenEdge Application Server: Administration*.

**Choosing a transaction model**

The client application and AppServer run in two separate sessions. Transactions are not propagated between these sessions. However, the AppServer supports two types of transactions:

- **Normal ABL transactions** — Where a single transaction begins and ends within the scope of a single remote procedure request from the client. The AppServer starts and manages the transaction within the scope of the remote procedure in exactly the same way an ABL client starts and manages a transaction within a local client procedure.

- **Automatic transactions** — Where a single transaction can span multiple remote procedure requests from the client. The AppServer can terminate the transaction at any time from the moment it starts until the end of the connection.

The choice of which type of transaction to use depends entirely upon the nature of your application. By definition, automatic transactions that span more than one remote procedure request tend to be larger than normal ABL transactions (assuming similar data transfers per request). Also, between multiple client requests with an open automatic transaction, there is more opportunity to make the transaction even larger if UI interactions occur on the client that effectively become part of the same transaction. This, in turn, creates more opportunity for lock conflicts and deadlocks among database clients, including the AppServer session.

For more information on trade-offs and issues for both types of transactions, see *Transaction and record management considerations* on page 147.

**Using load balancing**

If you install the AppServer Load Balancer for a NameServer installation, you can optimize overall performance based on the machine capacity of each installation. Using this option, the NameServer distributes connection requests among the AppServer installations according to weight factors that you set on each AppServer. For more information on load balancing and session models, see the information on load balancing and session models in *AppServer and Client Interaction* on page 23.

**Secondary performance considerations**

This section highlights additional specific performance considerations that you should review and understand as you design and implement an application using the AppServer.

**Connect and disconnect performance issues**

This section discusses performance issues related to connecting to and disconnecting from AppServers.
As with any computer architecture that uses remote-procedure-call technology, the time that it takes to make a connection can be costly. This initial startup time is increased when you use an AppServer Connect procedure.

### Using the AppServer Connect procedure

The AppServer Connect procedure provides a convenient place to perform connection activities. Although these connection activities have value for a variety of reasons, be aware that using these could, to various extents, affect your overall connection performance.

**Note:** A state-free Appserver never executes a Connect procedure.

You might want to perform any of the following activities independently of each other, or in combination with any other items:

- Authenticate and authorize based on additional parameters you can pass to the Connect procedure (*user-id*, *password*, and/or *app-server-info*)
- For a state-reset or state-aware AppServer, provide security for procedure execution by using the `EXPORT( )` method to define the allowed export list for client applications

**Note:** For a stateless AppServer, you probably want to set the export list in the Startup procedure.

- Connect to one or more databases, or to other AppServers on a state-aware or state-reset AppServer
- Initiate application-specific audit trails and/or activity logs

It might be difficult to predict exact connection speed and the effects of using any, all, or some combination of these options. However, in general, each of these options has some overhead. Although none of these options are required, you might want to consider them as part of your security model. See [Security considerations](#) on page 156 for more information.

### Performance-related issue for the CONNECT( ) method

Connecting to session-managed AppServers can be expensive depending on agent startup and initialization tasks defined in your AppServer Connect procedure. Connecting to a session-free application service can be expensive depending on the number of initial connections and NameServer responsiveness on the network. For these reasons, you might want to connect to AppServers at client application startup. Also, in most designs, it is advisable that you do not disconnect from AppServers if you intend to use services offered by a particular AppServer more than once prior to ending the client application session.
Performance-related issues for the DISCONNECT( ) method

On a state-reset or state-aware AppServer, the DISCONNECT( ) method has no significant performance impact on your client application regardless of whether there is an AppServer Disconnect procedure defined. The AppServer Disconnect procedure is run by the AppServer after the client connection is terminated and DISCONNECT( ) method execution has completed. That is, unlike other procedures executed in an AppServer session, the AppServer Disconnect procedure is run by a state-reset or state-aware AppServer concurrently with processing occurring in the client application.

For a state-reset or state-aware AppServer, you might be able to take advantage of the way the AppServer Disconnect procedure is run to improve overall application performance. When designing your distributed application, consider tasks to be performed by an AppServer that could be performed independently of the connection between the client application and the AppServer. These tasks would typically be ones where the client application does not require any information about the success or failure of the operation. Time-consuming actions are very good candidates for consideration. By moving these tasks to the AppServer Disconnect procedure, these actions can be performed concurrently with other client application processing, improving overall application performance.

On a stateless AppServer, any defined Disconnect procedure runs before the client connection is terminated. Thus, especially for frequent and short-duration connections, you probably want to ensure that the Disconnect procedure does not perform any processing that significantly delays the execution of the DISCONNECT( ) method. This can both impact the client and delay the availability of an AppServer agent.

In a session-free application (state-free operating mode), the DISCONNECT( ) method has minimal performance impact because no Disconnect procedure is executed.

Compiled compared to precompiled code

The execution process for noncompiled code on an AppServer is identical to the process for regular runtime compiled code on an ABL client. Standard ABL compilation processing activities and behaviors occur. Therefore, when noncompiled code must be used, from a performance perspective, there is the possibility of some improvement gains when the AppServer is running on a more powerful machine than the client application.

For example, if the AppServer you are running is on a faster machine, then the compilation process might occur faster for code compiled on it. Also, if AppServer agents have a self-service connection to the database, then compile-time schema information will not have to be sent over the network.

ABL browse design considerations

In an ABL client application, the ABL browse widget is bound to a query at compile time and controls record retrieval and presentation at run time.

In a traditional client/server application where a browse query satisfies a large number of records, the browse widget does not require all records to be delivered immediately; it caches some records not initially in the browse view port and, transparent to the user, requests additional records when the user scrolls to the end of the browse records. Therefore, there are no significant time delays relating to the delivery of a complete set of results prior to viewing data in the browse widget.

If browse data is to be retrieved from an AppServer, the following actions must occur:
1. A query running on an AppServer must gather data from the database.

2. Results must be returned to the client application using temp-table parameters.

3. The ABL client application must open the browse query against the returned temp-table.

This implies that if the remote query satisfies a large number of records, there might be a significant delay prior to viewing data in the browse widget. You might choose to accept this delay since once this delay is incurred no further network access is required. However, if this time behavior is unacceptable, consider choosing alternatives such as:

- Limiting the number of records that are delivered to the client application. For example, transfer only the first 200 records to the client application.

- Creating your own caching and delivery scheme, using the **APPEND** option of temp-table parameters. For example, transfer to the client application the first 200 records and an indicator as to whether there are more records available. The client application can then notify the user that there are additional records available. If the user actually requires the additional records, the client application can request the next batch of records from the AppServer agent, using the **APPEND** option, to add to the records that already have been transferred. For information about the **APPEND** option of temp-table parameters, see *OpenEdge Getting Started: ABL Essentials*.

**Note:** This alternative assumes that the AppServer agent building the original query is dedicated in state-reset or state-aware operating mode or is bound in stateless operating mode to the client application.

---

### Performance issues related to schema transmission

When there is a remote connection to a database from an ABL client, database schema information has to be constructed at the client. This involves network traffic to communicate the schema information from the database server to the client (even local schema caches require updates due to schema changes). For information on local schema caches, see the section on reducing connection times when connecting ABL clients to databases in *OpenEdge Deployment: Managing ABL Applications*.

By using the AppServer agent as the only mechanism to access the database, an ABL client no longer needs to connect to the database. This eliminates network traffic, associated with the delivery of the schema information, to the remote client.

### Garbage collection for class-based objects

Garbage collection protects against memory leaks by automatically deleting class-based objects that are no longer referenced. For AppServer applications, garbage collection will run always before a switch in the connection context. Thus, AppServer applications running in all modes are protected by garbage collection. For more information on class-based objects, see *OpenEdge Development: Object-oriented Programming*. 
Transaction and record management considerations

The following sections describe trade-offs and techniques to help you design and implement the transaction model for your distributed application.

**Note:** These sections apply mostly to session-managed applications (state-reset, state-aware, and stateless AppServer operating mode). For a session-free application, all transactions must be atomic, started, and completed in a single request.

### Comparing normal ABL and automatic transactions

Normal ABL transactions are designed to keep transaction scope very small, within the context of a single remote procedure call. Small transactions minimize the chances of conflict during concurrent accesses to a single database. This type of transaction is also identical to the type of transaction supported for ABL clients.

Because automatic transactions can remain active for multiple remote procedure calls, the scope of these transactions expands accordingly. This expanded scope allows much greater opportunity for conflict during concurrent accesses to a single database. As a result, you must be very cautious about how you use automatic transactions.

For example, you might provide an interface that closely mirrors the actual transaction mechanics, called from the client, as shown:

```plaintext
RUN startTransaction.p ON SERVER ...
RUN commitTransaction.p ON SERVER ...
```

Another example is the interface provided by `a-txtest.p` (see Programming the AppServer on page 49).

However, you might better simplify this process for the client with a more abstract interface, called from the client, as shown:

```plaintext
RUN initiateCustomerUpdate.p ON SERVER ...
RUN finishCustomerUpdate.p ON SERVER ....
```

These examples appear to accomplish the same thing. However, the first example implies that remote procedure return values monitor the transaction mechanism, while the second example implies that the remote procedure return values monitor the status of customer information. A well-designed outcome of the second example might be your guaranteeing that `finishCustomerUpdate.p` always commits the transaction appropriately, without the client having to be concerned with it.
Ensuring a small transaction size

One of the main advantages of using normal ABL transactions on the AppServer is their small scope, requiring no communication with the client for the AppServer transaction to complete. In general, to implement a normal ABL transaction in an AppServer procedure, you follow the same programming practices as when writing a small-scoped transaction for an ABL client.

In particular, avoid including code that can block the AppServer agent or otherwise delay execution, such as:

- Writing to or reading from external files
- Invoking any other operating system calls (for example, the `OS-COMMAND` statement)

For more information on implementing transactions in an ABL procedure, see *OpenEdge Getting Started: ABL Essentials*.

Returning transaction status to the client

When a client runs an AppServer procedure that implements a normal ABL transaction, the client can know the status of the transaction at the point the transaction succeeds or fails. You can communicate this status on the AppServer to the client by passing back a return value (`RETURN string` or `RETURN ERROR string`) or output parameter value from the remote procedure that encapsulates the transaction.

Thus, your AppServer procedure completely hides the actual mechanics of a normal ABL transaction from the client. For example, your return value might indicate whether or not a customer record was successfully updated. Managing the several reasons why the customer record might not be updated is the responsibility of your AppServer procedure implementation. Most importantly, once the remote procedure that implements the transaction returns to the client, all record locks are released and all records are available for access by other database clients.

Maintaining transaction consistency between client and AppServer

One disadvantage of using normal ABL transactions on the AppServer is that, because a normal ABL transaction completes and releases all locks before returning to the client, you may have to do extra work to ensure that the transaction completes as the client expects.

A typical scenario where this behavior might pose a problem is where the AppServer returns data derived from a database to a client application. The client then modifies the data and returns it to the AppServer with the intent that the changes to the data are to be applied to the database. Because database locks are not held across interactions with the AppServer, the AppServer cannot simply change the data in the database because the data might have already been changed by another user. Thus, transaction consistency in this situation cannot be easily maintained.

One approach to work around this problem, especially on a state-aware or state-reset AppServer is to have the AppServer keep a copy of the original data that is returned to the client application. On an AppServer running in state-reset operating mode, state-aware operating mode, or stateless operating mode over a bound connection, you can maintain a separate buffer or temp-table to hold a copy of the data. On a stateless AppServer over an unbound connection, you must use the `SERVER-CONNECTION-CONTEXT` attribute of the `SESSION` system handle or a context database to maintain a copy of the data. For more information, see the sections on managing stateless connection context in *Programming the AppServer* on page 49.
When it is time to apply the updated data, the AppServer agent handling the request determines whether the database was changed by another user by comparing the original data to the data in the database. The following figure describes this approach.

**Figure 6: Transaction data flow**

<table>
<thead>
<tr>
<th>Client application</th>
<th>Application server process</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Diagram of transaction data flow" /></td>
<td></td>
</tr>
</tbody>
</table>

The numbers in the previous figure correspond to the following explanation of transaction data flow:

1. The client application runs a remote persistent procedure on the AppServer to request the data. The AppServer gets the data from the database and stores it in a temp-table (Orig).

2. As a result of the remote persistent procedure call, the AppServer returns a copy of the Orig temp-table to the client application (Client).

3. The client application makes application-specific changes to its copy (Client~) that include creating, deleting, and modifying records.

4. The client application sends its modified copy (Client~) to the AppServer by running an internal procedure of the persistent procedure instantiated in Step 1. The AppServer stores its copy of the table in Client-Out.
5. Within a single transaction, the AppServer compares the records that are in the Orig temp-table to the data in the database. If there are any differences, the AppServer agent knows that the data was changed by another user, aborts the transaction, and notifies the client application that the updates failed.

6. If the data has not been changed (within the same transaction as Step 5), the AppServer then compares the Client-Out table to the Orig table. For each record within the Client-Out table that has been created, modified, or deleted, the AppServer makes the same changes to the data within the database.

There are variations on the approach suggested in the previous figure that provide similar results, and exactly how you vary this scheme will depend on your particular application requirements. For example, in Step 5, even if the AppServer finds a difference between the records in the Orig temp-table and the records in the database, it might not be necessary to abort the transaction. An alternative approach is to simply note which record has changed, find the corresponding record in the Client-Out temp-table returned to the AppServer, and skip that when applying the changes in Step 6. Only you can decide what approach is appropriate for your application.

Managing conflicts between client and AppServer

All AppServer transactions (normal or automatic) occur in a context that is separate and distinct from the client. One way of understanding this is that if both a client and an AppServer access the same database, each one accesses the database as a different user, whether or not the client is connected to the AppServer. As a result, any combination of AppServer clients and AppServer sessions can cause lock contention with each other as multiple database clients of the same database. (For more information on locks and lock contention, also known as deadlock, see OpenEdge Getting Started: ABL Essentials.)

Thus, it is important when performing direct database access from both client and AppServer sessions that you avoid or otherwise manage the possibility that all of these sessions might concurrently access the same record. If they do, and you attempt to run a transaction that changes the record on either the client or the AppServer, the SHARE-LOCK held by each one will prevent the record from being changed, and ultimately the transaction from being committed. If the continued execution of both the client and any Application Sever session depends on the transaction being completed, then a deadlock occurs that prevents the entire application from continuing.

To avoid a deadlock, you can use the Lock Wait Timeout (-lkwtmo) startup parameter for both an ABL client and the AppServer session (specified as part of the srvrStartupParam property value in your AppServer properties file). This parameter sets a limited time that any ABL procedure must wait to continue execution because of a lock. For more information on this parameter, see OpenEdge Deployment: Startup Command and Parameter Reference. However, this is not a solution to any fundamental design problems that might avoid the conflict altogether, such as an application that does not conform to the OERA.

The OpenEdge Reference Architecture (OERA), the Progress Software Corp. guide for designing OpenEdge enterprise applications, avoids the problem of client-server lock contention completely. In an application whose design conforms to the OERA, a client never accesses the enterprise data source and thus can never have a deadlock with the AppServer. The data source is managed solely by the AppServer, which also manages all exchange of enterprise data with the client.

However, if you cannot redesign your application to avoid this type of lock contention, you can code as described in the following sections:

- On the client on page 151
- On the AppServer on page 151
On the client

When a lock wait time out occurs on the AppServer because of a lock conflict that exceeds the time specified by the Lock Wait Timeout (\texttt{-lkwtmo}) startup parameter, a \texttt{STOP} condition is raised on the client side.

You cannot trap this stop condition simply by checking \texttt{ERROR-STATUS:ERROR} for a \texttt{RUN} statement that executes a remote procedure. The only way to handle this situation is to use the \texttt{ON STOP} phrase, as shown in this code fragment, as shown:

```
RUN testlkwtmo.
MESSAGE RETURN-VALUE VIEW-AS ALERT-BOX.

PROCEDURE testlkwtmo:
  DO ON STOP UNDO, RETURN "STOP occurred":
    DEFINE VARIABLE hServer AS HANDLE NO-UNDO.
    DEFINE VARIABLE lFlag AS LOGICAL NO-UNDO.
    /* Invoke procedure on the server that locks records. */
    CREATE SERVER hServer.
    hServer:CONNECT("-AppService asbroker1 -H localhost").
    RUN lockrecords.p ON SERVER hServer (OUTPUT lFlag) NO-ERROR.
    MESSAGE "flag :" lFlag SKIP "ERROR-STATUS:ERROR :" ERROR-STATUS:ERROR VIEW-AS ALERT-BOX.
  END.
END PROCEDURE.
```

If \texttt{lockrecords.p} terminates on the AppServer because of a lock wait time out, the AppServer raises a \texttt{STOP} condition on the client. The client program flow never reaches the message following the \texttt{RUN} statement that executes \texttt{lockrecords.p}. However, the client can catch the \texttt{STOP} condition and handle character string returned in \texttt{RETURN-VALUE}.

On the AppServer

On the AppServer, another technique for managing lock conflicts is to use the \texttt{NO-WAIT} and \texttt{NO-ERROR} options on any \texttt{FIND} statement that also applies an \texttt{EXCLUSIVE-LOCK} to the record. You can then test the resulting record buffer using the \texttt{LOCKED} function to detect a conflict:

```
FIND FIRST customer EXCLUSIVE-LOCK NO-WAIT NO-ERROR.
IF LOCKED customer THEN RETURN "customer,FIRST,LOCKED".
```

Buffer currency and \texttt{NO-LOCK}

If you try to find a record with \texttt{NO-LOCK} and that record is already located in a buffer, then depending on the exact nature of the request, The AVM (ABL Virtual Machine) may not reread the record from the database.
The fact that the AVM does not always reread NO-LOCK records may cause unexpected behavior in your distributed application. More specifically, although a record was updated within one session (client or AppServer), that update may not be seen within another session if that record was previously read with NO-LOCK. There are two scenarios where this may occur. These scenarios are called currency conflicts to denote the fact that they involve a buffer whose contents is not current:

- **Client-server currency conflict**
  In this scenario, a client finds a record NO-LOCK, and then sends a request to an AppServer to update the same record. When the request finishes, the client attempts to find the same record. Because the AVM may not reread the record, the client buffer might contain the copy of the record before the update occurred rather than the updated copy of the record.

- **Server-server currency conflict**
  In this scenario, a client sends a request to an AppServer whose operating mode is stateless. The AppServer agent who processes the request updates some record. The client then sends another request to the AppServer. This request happens to go to an AppServer agent that is different then the one that processed the first request. The AppServer agent attempts to find the same record that was updated using NO-LOCK. If the record was already stored in a buffer due to some previous clients request executed at this AppServer agent, the AVM may not reread the record. When this occurs, the AppServer agent buffer will contain the copy of the record before the update occurred rather than the updated copy of the record.

If resolving client-server or server-server currency conflicts is important to you, there are three general approaches that you can use:

- **Reading the current record** on page 152
- **Releasing the record** on page 152
- **Setting the -rereadnolock parameter** on page 152

### Reading the current record

If you know exactly which buffer is out of date, use FIND CURRENT or GET CURRENT with NO-LOCK. The CURRENT keyword indicates to the AVM that the record must be reread.

### Releasing the record

Use the RELEASE statement on all buffers that may be out of date before the record is reread. The RELEASE statement will clear the records from all buffers to which it is applied. At that point, all records that the AVM reads will need to be read from the database because a buffer copy no longer exists.

### Setting the -rereadnolock parameter

The -rereadnolock parameter indicates to the AVM that when an attempt is made to find a record NO-LOCK, even if the record is already in a buffer, then the record should be reread from the database. Use it as an ABL client startup parameter to resolve client-server currency conflicts. Use it as an AppServer startup parameter via the OpenEdge Explorer or OpenEdge Management or by setting the srvrStartupParam property in the ubroker.properties file for the appropriate AppServer to resolve server-server currency conflicts.

There are several things to keep in mind when using the -rereadnolock startup parameter.
• The -rereadnolock parameter has no affect on records that are being retrieved via RECID or ROWID. In that case, the AVM will not reread the record. It will use the copy of the record already stored in the buffer. If the most current version of the record is needed, then use the RELEASE statement on all buffers that contain a copy of the record before reading the record, or use FIND CURRENT or GET CURRENT statement to reread the record.

• The -rereadnolock parameter has no affect on the behavior of the query cache used for a NO-LOCK query as specified via the CACHE n phrase of the DEFINE QUERY statement. If the record is in the cache, it will not be reread regardless of whether -rereadnolock is set. To force the record to always be reread set CACHE 0. Note that setting the cache size to zero (0) may significantly degrade performance if the database is being accessed across a network. Only set the cache size to zero (0) when it is critical to retrieve the most current version of a record.

• The -rereadnolock parameter has no affect on the behavior of the prefetch cache that is used by default when retrieving records NO-LOCK across the network. By default, when executing a CAN-FIND function, or the FIND, FOR, or OPEN QUERY statements on a database which is being accessed across a network, the AVM fetches several records at a time, and stores those records within a prefetch cache. The AVM will only sends a request to the database server to fetch more records if the requested record is not contained within the current prefetch cache. If the record is in this cache, a new copy of that record will not be read even if -rereadnolock is set. To eliminate this cache so that the most current version of the record is always read use the NO-PREFETCH keyword in the appropriate statements. Note that using the NO-PREFETCH keyword may significantly degrade performance. Only set NO-PREFETCH when it is critical to retrieve the most current version of a record.

Deployment considerations

The following section highlights some deployment considerations that you might want to consider as part of your strategy for deploying an application with the AppServer.

Minimizing the effects of schema changes

In the traditional client/server model where all ABL clients are connected to a database, database schema changes can invalidate r-code run by the client. In contrast, by designing your ABL client application to be independent of database schema, you can use the AppServer to minimize the effects of schema changes by masking the changes from the client application.

The following list identifies specific techniques you should use to shield these schema changes from affecting client applications:

• Set up database access so that it is accomplished entirely through AppServer agents (the client is not connected to the database and all records are transferred from the AppServer agent through temp-table parameters)

• Avoid using LIKE syntax for temp-tables and variables in client application code and in procedure parameters for remote procedures

• AppServer agent code must map data based on the new schema to the existing procedure parameters for the remote procedure
Advantages of using portable r-code

ABL r-code is the intermediate binary code that OpenEdge generates when it compiles ABL source files. This is the code that is actually run when a procedure is executed. ABL r-code is portable between two dissimilar platforms if all of the following apply:

- The source code is not dependent on the user interface
- The source code is not dependent on DataServer connections
- The platforms are compatible between 32-bit and 64-bit architectures

For more information, see the sections on portable r-code in *OpenEdge Deployment: Managing ABL Applications*.

Using portable r-code, you can perform application-server development on a platform that is different from your application-server deployment platform. For example, you can compile r-code that is destined to run on a UNIX 32-bit AppServer agent on a Windows ABL client. Further, this means that by compiling on the Windows ABL client, a compiler license for the UNIX platform is not required.

Note the following cautions if you intend to use portable r-code:

- Examine your code for platform-specific code blocks. The presence of these code blocks can limit your ability to make your r-code portable.
- Ensure that your ABL code addresses all target r-code platforms on which you want your code to run.

The following examples compare platform-specific preprocessor directives that will be resolved at compile time with a comparable ABL code that is resolved at run time.

**Preprocessor directive resolved at compile time:**

```
&IF &OPSYS = "WIN32" &THEN
...
&ELSE
 &IF &OPSYS = "UNIX" &THEN
...
 &ELSE
 ...
```

**Non-preprocessor ABL code resolved at run time:**

```
IF OPSYS = "WIN32" THEN
...
ELSE
 IF OPSYS = "UNIX" THEN
...
 ELSE
 ...
```

If the code from the first example is compiled on a Windows client, the "WIN32" logic would be executed even if the resulting r-code is run by an AppServer agent on a UNIX host. Therefore, if there can be multiple deployment platforms for AppServer agent code, you should use run-time resolution for platform-specific blocks as the second code example in this section shows.
Server configurations and resources

The AppServer architecture provides potential for improvements in performance, security, and deployment at the expense of server resources. By locating AppServer processes on the same machine as your database server, you might require:

- Additional memory (physical and virtual)
- Disk space (to locate application code to be run by AppServer agents)
- Additional network resources (for more information, see Network deployment considerations on page 155)

Actual memory requirements that extend beyond database needs vary, depending on the number and configuration of AppServer brokers, AppServer agents, and NameServers, the number of clients to be serviced, the operating mode, as well as the actual ABL procedures to be executed by the AppServer agents, based on a set of hardware guidelines.

You can thus use the following guidelines to help identify physical memory requirements for an AppServer configuration:

- **AppServer agents** — Each AppServer agent adds a minimum of 4MB. For a state-aware or state-reset AppServer, the client connection adds another 100KB.

- **AppServer brokers** — AppServer broker memory depends on the operating system and its message traffic, but might run from about 50MB to 80MB (round numbers). For a stateless or state-free AppServer, each concurrent connected client adds about 100K to broker memory. This reflects concurrent client connections to the broker for stateless operating mode and concurrent client messages handled by the broker for state-free operating mode.

Network deployment considerations

As you plan your overall deployment strategy, keep in mind that the AppServer supports only:

- Transport Control Protocol/Internet Protocol (TCP/IP) for client access to the AppServer
- Universal Datagram Protocol (UDP) for client (and AppServer) access to the NameServer

However, an AppServer agent, like any ABL client session, can communicate with databases and DataServers using TCP/IP or SNA.

If you have to connect the AppServer to a DataServer to access a non-OpenEdge database (and possibly through another network protocol), do the following:

- Use the appropriate OpenEdge DataServer
- Use the OpenEdge OEBuild utility to create a customized AppServer agent executable if the appropriate DataServer is not already built into the client

For more information on building an AppServer agent executable, see the sections on building OpenEdge executables in *OpenEdge Deployment: Managing ABL Applications*.
Security considerations

The general security framework for your networking system is governed by security elements associated with your operating system, networking software, file system, application, and database. In addition to these features, the AppServer offers you specific security options.

As you read through this section, consider your security goals. For example, what type of database access do you want to make available for your user community? How much control do you want to exert over individual users' ability to run specific procedures? Do you have a business need to produce audit trail details? This type of appraisal will help you determine whether to implement a tight security model, with strict control over application data, or a loose security model, with fewer access controls. That is, although each security option is unique and independent of other options, the total effect of the options you choose to implement will tend toward a tighter or looser security model.

Overview of the AppServer security options

The following table summarizes the various application-specific security options available.

Table 9: AppServer security considerations

<table>
<thead>
<tr>
<th>Security consideration</th>
<th>Design strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>User authentication and authorization</strong> on page 157</td>
<td>Use the AppServer Connect procedure to validate a user</td>
</tr>
<tr>
<td><strong>Database access</strong> on page 158</td>
<td>Limit or prevent client application access to a database</td>
</tr>
<tr>
<td><strong>AppServer session access</strong> on page 158</td>
<td>Limit users’ access to specific procedures through the <code>EXPORT( )</code> method</td>
</tr>
<tr>
<td><strong>Audit trails</strong> on page 161</td>
<td>Use the AppServer Connect and AppServer Disconnect procedures to assist in management of audit trails</td>
</tr>
<tr>
<td><strong>Run-time compilation</strong> on page 163</td>
<td>Protect any executable code that you deploy in source form</td>
</tr>
<tr>
<td><strong>Operating system</strong> on page 163</td>
<td>Protect the files that you deploy for an AppServer</td>
</tr>
<tr>
<td><strong>Secure Sockets Layer (SSL)</strong> on page 163</td>
<td>Provide data privacy over connections to the AppServer and authentication between clients and servers on those connections</td>
</tr>
</tbody>
</table>

The remaining sections on security provide additional information about each of these considerations.

For information about developing these features, see Programming the AppServer on page 49. Also, for more information about standard ABL security features, see OpenEdge Development: Programming Interfaces.
User authentication and authorization

You typically handle authentication and authorization tasks at the point of connection, using the Connect procedure, which can accept parameters for user authentication and refuse a client connection according to the result. For a state-reset or state-aware AppServer, you can take full advantage of this to authorize, on a user basis:

- What procedures (entry points) can be run by the connected AppServer agent by setting an export list using the EXPORT( ) method.
- Connections to databases and other AppServers from the connected AppServer agent. For more information on limiting database access through a connected AppServer agent, see Database access on page 158.

However, for stateless and state-free AppServers, you cannot directly authorize and implement these options at connect time.

Authorization on a state-reset or state-aware AppServer

The following code shows a typical authentication and authorization example:

```prolog
DEFINE INPUT PARAMETER pcPassword AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER pcUserId AS CHARACTER NO-UNDO.

/* Authenticate user - if the user/password does not exist an error condition will automatically be raised. */
FIND FIRST app_user WHERE app_user.user_id = pcUserId
  AND app_user.password = ENCODE(pcPassword) NO-LOCK.

/* Authorize access to particular procedures */
IF NOT SESSION:EXPORT(app_user.can_run) THEN DO:
  /* Log failure message to AppServer agent logfile and refuse connection */
  MESSAGE "Failed to create export list for" app_user.user_id.
  RETURN ERROR.
END.
```

First, the user_id and password established by the client application's CONNECT( ) method are authenticated against an external source (in this case, valid users are identified in a database table named app_user). Secondly, the particular user is authorized to request execution of an established, user-specific, list of procedures (entry points) through the EXPORT( ) method.

To create a tight security model, establish an export list in conjunction with operating-system security to restrict access from the client applications host to the remote procedure sources. For more information on operating-system security, see Operating system on page 163.

Authorization on a stateless or state-free AppServer

For a stateless AppServer, you cannot easily set an export list and database connections at connect time, because the AppServer agent that runs the Connect procedure is not necessarily the one that executes subsequent remote procedure requests for the connected client. For a state-free AppServer, you cannot perform these functions at connect time at all, because the client maintains no physical connection to any one AppServer, and the AppServer never runs a Connect procedure.

For stateless and state-free AppServers, setting an export list and database connections works much more effectively as a global activity that you implement in the Startup procedure, which executes for every AppServer agent when it starts up.
You can also pass connection-based authentication and authorization information to each AppServer agent that handles a remote procedure request for a client connection. The AppServer agent can retrieve and resave the authorization information for the next AppServer agent that handles the connection using the Activate and Deactivate procedures. For more information on using these procedures, see Programming the AppServer on page 49. For more information on creating an export list using the \texttt{EXPORT( )} method, see the AppServer session access on page 158.

### Database access

The traditional client/server model requires a connection from the ABL client to the database server. The AppServer allows alternative strategies to be considered. One strategy is for the ABL client to have no connection to the database and for all database access to be achieved through an AppServer. Such an implementation provides tight security since database access is limited to the parameters, and functionality contained in the procedures that are remotely executed by the AppServer.

A less severe strategy is to allow the ABL client read-only access to the database, but provide full update access to the AppServer agent. This preserves full control of updates through the restricted set of procedures that can be run in an AppServer agent, yet allows the client a broader range of database query capabilities. This alternative might be required to achieve certain performance objectives, for example, those peculiar to browse widget behavior. See ABL browse design considerations on page 145 for detailed information about browse behavior.

### AppServer session access

You can publish entry points (remote procedure pathnames) that control access to an AppServer session at run time using the \texttt{EXPORT( )} method on the \texttt{SESSION} handle. Publishing an export list limits the remote procedures that a client application can execute to those that you specify with this method. Because this is a run-time operation, you can change the list at any time that your application requires, and you can reset the list to empty, allowing access to the entire set of procedures available from the current AppServer PROPATH setting.

However, setting this list applies only to the AppServer session in which you execute the \texttt{EXPORT( )} method. This means that you might have to manage the list differently in your application depending on the operating mode that you configure for the AppServer.

**Note:** As this is a context management problem, the only way to effectively manage this kind of information for a session-free AppServer is to maintain your own context database for it. Therefore this section applies only to session-managed AppServers.

For more information on setting and resetting entry points in an AppServer session, see Programming the AppServer on page 49.

### Operating mode interactions

Because of how the \texttt{EXPORT( )} method and AppServer operating modes work together, you must use it differently, depending on your AppServer operating mode. When you invoke the method with a list of procedure names, the method adds this list to any existing list of entry points that you have previously specified for an AppServer session.
If the AppServer is running in state-reset operating mode, each AppServer agent automatically resets any export list to empty with each new connection. However, if the AppServer is running in state-aware or stateless operating mode, you must manage all setting and resetting of entry points manually. Otherwise, each time you execute the method with a list of procedure names, they are added to the existing list of entry points. Thus, unless you explicitly reset the list, on a state-aware AppServer, the number of entry points set for an AppServer agent can grow with each client connection, and on a stateless AppServer, the number can grow with each client request.

How to set and reset an export list for each operating mode

Each AppServer operating mode requires a different arrangement for setting and resetting the published entry points during an AppServer session:

- **State-reset operating mode on page 159** — You can set an export list any time before a client application uses them; you never need to reset your export list from one connection to the next.

- **State-aware operating mode on page 159** — You can set an export list any time before a client application uses them; if you set them in remote procedures, you probably want to reset them with each connection.

- **Stateless operating mode on page 160** — You can set an export list any time before a client application uses them; you generally set them in the Startup procedure for the entire AppServer session. You can also set them with a finer degree of session granularity, but you must exercise caution when doing so.

- **State-free operating mode on page 161** — You can most practically set an export list only in the Startup procedure for the entire AppServer session. You can also set them with a finer degree of session granularity, but you must exercise caution and consider the connectionless nature of state-free remote requests when doing so.

State-reset operating mode

When the connection terminates on a state-reset AppServer, the export list for the disconnected AppServer agent is automatically reset to empty. Thus, you must set any export list on a state-reset AppServer for each new connection and in some procedure that is guaranteed to be executed when you need to have the list set.

You might want to set the export list in the Connect procedure if you know the export list criteria at the start of each connection. The criteria can be based on the user or any other information that you determine at the time of connection. Otherwise, you must set or reset the list in some remote procedure executed by the client.

State-aware operating mode

For a state-aware AppServer, you can conveniently set a single export list for all AppServer agents, regardless of the connection, using the Startup procedure. Otherwise, you can set it, like on a state-reset AppServer, in the Connect procedure or some remote procedure executed by the client.
When the connection terminates on a state-aware AppServer, the disconnected AppServer agent maintains the last setting of the export list. If you want the export list to be empty at the start of every connection, you must explicitly reset the list at some point between connections. You can guarantee this for every AppServer session by resetting the list in the AppServer Connect or Disconnect procedure, as shown:

```c
lReturn = SESSION:EXPORT(). /* Reset to empty */
```

**Stateless operating mode**

For a stateless AppServer, you can conveniently set a single export list for all AppServer agents (global to the AppServer session), using the Startup procedure. Because export list settings apply only to the AppServer agent in which they occur, setting and resetting an export list at any other point on a stateless AppServer requires careful attention.

**Caution:** Setting an export list during remote procedure requests to a stateless AppServer is an error-prone activity. If your application requires this fine degree of control, the rest of this section explains how you can do it. However, you might consider implementing your AppServer using state-aware operating mode, instead.

Like a state-aware AppServer, once you set an export list, the list remains set in the AppServer session until you reset it. For an unbound connection, because you cannot guarantee when an AppServer agent will execute what procedure, you must manage any export list settings appropriately during each request. If you do not, you can leave an unpredictable set of export list settings across AppServer agents in the AppServer pool.

For an unbound connection, you can maintain consistent export list settings using the Connect, Activate, and Deactivate configuration procedures. This is especially useful to provide user-based access control.

To maintain consistent export list settings using the Connect, Activate, and Deactivate configuration procedures:

1. In the Connect procedure, establish an export list based on the authenticated user.
2. Save the export list using stateless context storage, such as the `SERVER-CONNECTION-CONTEXT` attribute or a context database.
3. In the Activate procedure, retrieve the export list from your stateless context storage and set it using the `EXPORT()` method. The list is now set to filter the current remote procedure call and ensure its validity for the user.
4. In the Deactivate procedure, reset the export list to empty or to any other global session value using the `EXPORT()` method. This leaves the AppServer agent to handle remote procedures calls from all connections, as appropriate.

**Note:** The Disconnect procedure is not effective to reset a user-based export list, especially for an unbound connection, because the Disconnect procedure can run in an AppServer agent other than the one where the last export list setting occurred.
You can also use the Activate procedure to make application-based settings, where you set the list depending on some application state other than the authenticated user. In this case, you might determine the setting indirectly from stateless context storage or from some other source available to ABL. For more information on using stateless context storage, see Programming the AppServer on page 49.

State-free operating mode

For a state-free AppServer, you cannot perform these functions at connect time at all, because the client maintains no physical connection to any one AppServer, and the AppServer never runs a Connect procedure. For state-free AppServers, setting an export list and database connections works much more effectively as a global activity that you implement in the Startup procedure, which executes for every AppServer agent when it starts up. However, note that the criteria for initializing the export list cannot easily be user-based, as there is no connected user.

You can pass request-based export list information to each AppServer agent that handles a remote procedure request. The AppServer agent can retrieve and resave the export list information for the next AppServer agent that handles the client using the Activate and Deactivate procedures.

Caution: Setting an export list during remote procedure requests to a state-free AppServer is an error-prone activity. If your application requires this fine degree of control, the rest of this section explains how you can do it. However, you might consider implementing your AppServer using state-aware operating mode, instead.

To maintain consistent export list settings using the Activate and Deactivate configuration procedures:

1. In the Startup procedure, establish an initial export list based on whatever criteria your application requires (for example, the time of day).

2. Save the initial export list using state-free context storage, such as a context database or operating system file. This context storage must be accessible to all AppServers registered to handle the same state-free application service.

3. In the Activate procedure, retrieve the export list from your state-free context storage and set it using the EXPORT( ) method. The list is now set to filter the current remote procedure call and ensure its validity according to whatever criteria you determine for the request.

4. In the Deactivate procedure, reset the export list to empty or to any other global session value using the EXPORT( ) method and resave the reset export list to your state-free context storage. This leaves the current or any other AppServer agent in any AppServer that supports the state-free application service to handle future remote procedure requests, as appropriate.

Audit trails

Certain classes of applications require audit trails to be generated to trace events such as:

- Who connected to a particular service and when they connected
- What activity they performed during the life of the connection
- What breaches in security they tried to perform during the life of the connection
Note: Audit trails based on a client connection do not apply for a session-free application, because there is no meaningful SERVER-CONNECTION-ID or CLIENT-CONNECTION-ID value for this purpose. For both session-free and session-managed applications, you can use the ClientContextId property on the Progress.Lang.OERequestInfo class. For more information on this property, see Managing state-free and stateless client context on page 69.

AppServer application audit trails

On a session-managed AppServer, you can access the current connection ID using the SERVER-CONNECTION-ID attribute on the SESSION handle. On a session-free or session-managed AppServer, you can access the ClientContextId property on the Progress.Lang.OERequestInfo class instance referenced by the SESSION handle's CURRENT-REQUEST-INFO attribute.

Ideal locations to generate an audit trail on an AppServer include the AppServer Connect and Disconnect procedures. For example, your Connect procedure can use the user-id parameter and the SERVER-CONNECTION-ID attribute or ClientContextId property value to write an audit trail record when a client application connects to the AppServer. Also, any code to create an audit trail record in the Disconnect procedure is always executed, since this procedure is guaranteed to run whether the connection is explicitly disconnected or forcibly disconnected in response to the QUIT condition.

On an AppServer operating in stateless operating mode, you can also use the AppServer Activate and Deactivate procedures to generate audit trails for each request. However, this use can reduce performance, depending on the load on your AppServer and the amount of information in the audit trail.

For more information on programming these procedures and on accessing the SERVER-CONNECTION-ID attribute, see Programming the AppServer on page 49. For more information on accessing the ClientContextId property, see Managing state-free and stateless client context on page 69.

Note: Audit information created in an AppServer agent using the MESSAGE statement or other external output syntax such as the DISPLAY statement, is sent to the AppServer log file unless otherwise directed.

Client application audit trails

On a session-managed ABL client session, you can access the current connection ID using the CLIENT-CONNECTION-ID attribute on the server object handle used to connect the AppServer. For an Open Client (.NET, Java, or Web service Open Client) application, OpenEdge provides appropriate methods for you to access the connection ID. On a session-free or session-managed ABL client, you can access the ClientContextId property on the Progress.Lang.OERequestInfo class instance referenced by the server object handle's REQUEST-INFO attribute.

Typically, you might generate audit trail information on the client in code associated with AppServer connection and disconnection requests, or even with each remote procedure request, depending on how much information you need to capture.

For more information on accessing the CLIENT-CONNECTION-ID attribute from an ABL client application, see Accessing the connection ID on a session-managed client on page 104. For information on the common methods for accessing the connection ID from an open client, see OpenEdge Development: Open Client Introduction and Programming.
Run-time compilation

Compiling code at run-time (compile-time execution) gives you the capability to execute dynamic procedures that execute queries and other immediate compile-and-run-related activities. (Although, with Version 9, you are more likely to use dynamic queries than dynamic procedures to execute immediate queries.) Compile-time execution can be an asset while developing an application, but a security issue when deploying, if other safeguards are not provided to protect the source code.

If an application is deployed using runtime compilation, ensure that your security framework is tightly defined so that your other security efforts cannot be violated. For example, you should ensure that all source files are appropriately secured at the operating-system level.

Operating system

In any application environment, access to directories and files can pose a security risk. To provide a tightly secured environment in which to run the AppServer, do not allow client access (using remote log ins, shells, copying, or file sharing) to directories that contain:

- AppServer startup/shutdown scripts
- AppServer log files
- Application code to be executed by AppServer agents

Secure Sockets Layer (SSL)

You can use the Secure Sockets Layer (SSL) to provide a security infrastructure that protects AppServer communications both on the Internet and the intranet. SSL provides data privacy over network connections and authentication between clients and servers on those connections using elements of Public Key Infrastructure (PKI). These elements include private and public keys that the clients and servers use to authenticate each other and to set up data encryption and decryption services between the initiator of the communications (SSL client) and the responder (SSL server). The server is identified by the private key that it stores and the client is identified as a valid SSL client for that server by the public key that it stores and provides to the server. SSL clients gain access to public keys using digital (public key) certificates provided by a trusted certificate authority (CA) that also provides the private key confidentially to the SSL server.

SSL is both application and transport independent. For the AppServer, OpenEdge supports SSL in two basic transport contexts:

- **HTTP** — For Internet applications
- **AppServer protocol** — For intranet applications

For more information on SSL and how it uses private and public keys and public key certificates to handle security tasks in these contexts, see *OpenEdge Getting Started: Core Business Services - Security and Auditing*.

**Note:** SSL incurs heavy performance penalties, depending on the client, server, and network resources and load.
Secure AppServer configurations

To enable SSL security between SSL clients and servers, you must manage access to the public and private keys for each SSL client and server, respectively. How you do this depends on the SSL configuration you are managing. To secure an AppServer using SSL, you can configure the AppServer connections in two different ways or in combination, as follows:

1. Internet-secure AppServer — A secure connection over the Internet between the AppServer client and a Web server that hosts an appropriate OpenEdge adapter to access the AppServer. The connection between the client and Web server is secured using HTTPS (HTTP tunneled through SSL).

2. SSL-enabled AppServer — A secure connection directly between the client and the AppServer. This connection is secured by tunneling the direct AppServer protocol through SSL.

3. Internet-secure and SSL-enabled AppServer — A combination of steps 1 and 2, where the connection between the AppServer client and the Web server is secured using HTTPS (as in step 1) and the connection between the appropriate OpenEdge adapter and AppServer is secured using SSL tunneling, similar to a direct connection between the AppServer client and AppServer (as in step 2).

Note: This configuration is likely to incur the heaviest performance penalty of the three because it uses two SSL connections, each of which requires encryption and decryption of data for each client request and for each AppServer response.

The remaining sections on SSL and the AppServer describe:

- Internet-secure AppServer on page 164
- SSL-enabled AppServer on page 167
- SSL connection handling for brokers and agents of an SSL-enabled AppServer on page 170
- Identifying the SSL server connected to an AppServer client on page 171

Internet-secure AppServer

You can configure HTTPS security for an AppServer application on the Internet, depending on the type of client and AppServer adapter. For an AppServer client that directly calls AppServer application services (ABL client or .NET and Java Open Client), you use the AppServer Internet Adapter (AIA) to access the AppServer on behalf of the client. For a Web service client that accesses AppServer application services as OpenEdge Web services, you use the Web Services Adapter (WSA) to access the AppServer on behalf of the Web service accessed by the client. The OpenEdge Adapter for Sonic ESB also supports Web service client or general ESB client and service access to the AppServer as an OpenEdge service with Internet and intranet security on the ESB side provided by the Sonic Enterprise Service Bus (ESB).

In all cases, to secure an Internet connection between the client and AppServer using HTTPS, you must configure:

- The client as an HTTPS client
- The server hosting the AppServer adapter as an HTTPS server

You can configure and connect the client (AppServer or Web service client) as an HTTPS client, depending on the client type, as described in the following table:
### Table 10: HTTPS management for clients of an Internet-secure AppServer

<table>
<thead>
<tr>
<th>Client type</th>
<th>HTTPS configuration is supported by . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABL client</td>
<td>OpenEdge using the <code>certutil</code> command-line tool to manage public key certificates. The client connects using an HTTPS URL formatted for accessing an Internet-secure AppServer. For more information on managing the public key certificates, see the sections on managing OpenEdge key and certificate stores in <em>OpenEdge Getting Started: Core Business Services - Security and Auditing</em>. For information on connecting to an Internet-secure AppServer from an ABL client, see the sections on using the <code>-URL</code> connection parameter in <em>Programming ABL Client Applications</em> on page 87 and the sections on formatting an Internet URL to the AppServer using HTTPS in <em>Connecting to AppServers Using a URL</em> on page 187.</td>
</tr>
<tr>
<td>Open Client</td>
<td>Microsoft tools to manage public key certificates for .NET Open Clients and by OpenEdge using the <code>procertm</code> command-line tool to manage public key certificates for Java Open Clients. The client connects using an HTTPS URL formatted for accessing an Internet-secure AppServer. For more information on managing public key certificates for .NET clients, see the information on managing certificate stores in the Microsoft .NET documentation. For more information on managing public key certificates for Java clients, see the information on managing certificate stores in <em>OpenEdge Development: Java Open Clients</em>. For information on connecting to an Internet-secure AppServer from an Open Client, see the sections on connecting to the AppServer in <em>OpenEdge Development: Open Client Introduction and Programming</em>, and the sections on formatting an Internet URL to the AppServer using HTTPS in <em>Connecting to AppServers Using a URL</em> on page 187.</td>
</tr>
</tbody>
</table>
HTTPS configuration is supported by . . .

<table>
<thead>
<tr>
<th>Client type</th>
<th>HTTP configuration is supported by . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web service client</td>
<td>Web service client platforms for Web service clients. Web service clients connect to or access the Web service through the Web server using the WSA URL configured for it and provided to the Web service client as part of other Web service access information included in the Web Services Description Language (WSDL) file for the specified Web service. The WSA, itself, maintains the connection information to the Web service specified in the WSDL file. For more information on managing public key certificates for Web service clients, see the documentation on your Web service client platform. For more information on how a Web service client connects to and accesses the WSA-managed Web service, see the information on programming Web service clients in <em>OpenEdge Development: Web Services</em> and the information on managing a WSA and deploying an OpenEdge Web service in <em>OpenEdge Application Server: Administration</em>.</td>
</tr>
<tr>
<td>ESB client or service</td>
<td>The Sonic ESB. An ESB client or service accesses other ESB services using the interfaces provided by the Sonic ESB. For more information on managing public key certificates and any other ESB security configuration options for ESB clients and services, and on accessing ESB services in general (including OpenEdge services), see the Sonic ESB documentation. For more information on using the OpenEdge Adapter for Sonic ESB to support OpenEdge services on the Sonic ESB, see <em>OpenEdge Development: Messaging and ESB</em>.</td>
</tr>
</tbody>
</table>

On the server end of the Internet connection, you can configure the HTTPS server and OpenEdge adapters that access the AppServer as described in the following table.

**Table 11: HTTPS management for Internet-secure AppServer adapters**

<table>
<thead>
<tr>
<th>Adapter type</th>
<th>HTTP configuration is supported by . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIA</td>
<td>The Web server utilities provided to manage private key stores for the Web server accessed as an HTTPS server. For more information on managing key stores on a Web server, see the Web server documentation. Also, in order to ensure that AppServer clients use HTTPS to access the AIA, you must set the <code>httpsEnabled</code> property appropriately in the <code>ubroker.properties</code> file for the AIA. For more information, see the sections on configuring the AIA in <em>OpenEdge Application Server: Administration</em>.</td>
</tr>
</tbody>
</table>
SSL-enabled AppServer

To secure a direct connection between the client and AppServer using SSL, you must configure:

- The AppServer client as an SSL client
- The AppServer, itself, as an SSL server

**Note**: A given AppServer can support only SSL or non-SSL connections. It cannot support both simultaneously. So, to allow applications to run in both an SSL-enabled and non-SSL-enabled AppServer environment, you must dedicate at least one AppServer to SSL connections and another AppServer to non-SSL connections. SSL-enabled and non-SSL-enabled AppServers can both run as Internet-secure AppServers using the AIA (see Internet-secure AppServer on page 164).

Configure and connect the SSL client to the AppServer by client type, as detailed in the following table.

**Table 12: SSL management for clients of SSL-enabled AppServers**

<table>
<thead>
<tr>
<th>Client type</th>
<th>SSL configuration is supported by . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABL client</td>
<td>OpenEdge using the <code>certutil</code> command-line tool to manage public key certificates. The client connects using an AppServer URL formatted for accessing an SSL-enabled AppServer. You can specify this URL using the <code>AppServerS</code> protocol for an SSL connection mediated by a</td>
</tr>
</tbody>
</table>
### SSL configuration is supported by . . .

<table>
<thead>
<tr>
<th>Client type</th>
<th>Support Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>NameServer or using the AppServerDCS protocol for an SSL connection directly to a specified AppServer. For more information on managing the public key certificates, see the sections on managing OpenEdge key and certificate stores in <em>OpenEdge Getting Started: Core Business Services - Security and Auditing</em>. For information on connecting to an SSL-enabled AppServer from an ABL client, see the sections on using the -URL connection parameter in <em>Programming ABL Client Applications</em> on page 87 and the sections on formatting an AppServer URL using the AppServerS or AppServerDCS protocol in <em>Connecting to AppServers Using a URL</em> on page 187.</td>
<td>Open Client: Microsoft tools to manage public key certificates for .NET Open Clients and by OpenEdge using the procert command-line tool to manage public key certificates for Java Open Clients. The client connects using an AppServer URL formatted for accessing an SSL-enabled AppServer. You can specify this URL using the AppServerS protocol for an SSL connection mediated by a NameServer or using the AppServerDCS protocol for an SSL connection directly to a specified AppServer. For more information on managing public key certificates for .NET clients, see the information on managing certificate stores in the Microsoft .NET documentation. For more information on managing public key certificates for Java clients, see the information on managing certificate stores in <em>OpenEdge Development: Java Open Clients</em>. For information on connecting to an SSL-enabled AppServer from an Open Client, see the sections on connecting to the AppServer in <em>OpenEdge Development: Open Client Introduction and Programming</em>, and the sections on formatting an AppServer URL using the AppServerS or AppServerDCS protocol in <em>Connecting to AppServers Using a URL</em> on page 187.</td>
</tr>
</tbody>
</table>
### SSL configuration is supported by . . .

<table>
<thead>
<tr>
<th>Client type</th>
<th>SSL configuration is supported by . . .</th>
</tr>
</thead>
</table>
| WSA         | OpenEdge:  
|             | • Using the `certutil` command-line tool to manage public key certificates for the WSA as an SSL client of the AppServer  
|             | • Using the OpenEdge Explorer or OpenEdge Management to configure the required Web service properties for each Web service managed by the WSA that you want to establish an SSL connection to the AppServer  
|             | Each Web service managed by a WSA can be configured individually as an SSL-enabled Web service. However, you manage the public key certificates for all SSL-enabled Web services that are managed by a single WSA using the same certificate store.  
|             | With a given WSA-managed Web service in an SSL connection to the AppServer, the Web service client would typically connect to the Web service itself using HTTPS to ensure a secure connection all along the way to the AppServer. See [Internet-secure AppServer](#) on page 164.  
|             | For information on managing the public key certificates for a WSA as an SSL client of the AppServer, see the sections on managing OpenEdge certificate stores in *OpenEdge Getting Started: Core Business Services - Security and Auditing*. Or, for configuring a Web service as an SSL-enabled Web service, see the sections on WSA and Web service administration in *OpenEdge Application Server: Administration*.  
|             | Both:  
|             | • OpenEdge using the `certutil` command-line tool to manage public key certificates for the OpenEdge Adapter for Sonic ESB as an SSL client of the AppServer  
| OpenEdge Adapter for Sonic ESB | • Sonic ESB using the Sonic ESB Explorer to configure the required OpenEdge service properties for each OpenEdge service managed by the OpenEdge Adapter for Sonic ESB that you want to establish an SSL connection to the AppServer  
|             | Each OpenEdge service managed by a OpenEdge Adapter for Sonic ESB can be configured individually as an SSL-enabled OpenEdge service. However, you manage the public key certificates for all SSL-enabled OpenEdge services that are managed by a single OpenEdge Adapter for Sonic ESB using the same OpenEdge certificate store.  
|             | In this configuration, with a given OpenEdge service in an SSL connection to the AppServer, a Web service client would typically connect to the OpenEdge service itself using HTTPS to a SonicMQ broker, which then communicates with the OpenEdge Adapter for Sonic ESB in order to ensure a secure connection all along the way to the AppServer. See [Internet-secure AppServer](#) on page 164.  
|             | For more information on managing the public key certificates for a OpenEdge Adapter for Sonic ESB as an SSL client of the AppServer, see the sections on managing OpenEdge certificate stores in *OpenEdge Getting Started: Core Business Services - Security and Auditing*. For more information on configuring an OpenEdge service as an SSL-enabled OpenEdge service, see the sections on OpenEdge Adapter for Sonic ESB.
SSL configuration is supported by setting the appropriate SSL properties for the AppServer using the OpenEdge Explorer or OpenEdge Management. You must also manage a key store that contains the private key(s) for the AppServer using the `pkiutil` command-line tool.

For more information on configuring an SSL-enabled AppServer, see the sections on AppServer administration in *OpenEdge Application Server: Administration*. For more information on managing key stores for an SSL-enabled AppServer, see the sections on managing OpenEdge key stores in *OpenEdge Getting Started: Core Business Services - Security and Auditing*.

**Note:** An SSL-enabled AppServer can accept only SSL AppServer connections.

### SSL connection handling for brokers and agents of an SSL-enabled AppServer

When you configure an SSL-enabled AppServer, the AppServer broker and its agents use SSL-enabled server sockets depending on the AppServer operating mode, as shown in the following table.

<table>
<thead>
<tr>
<th>For this AppServer operating mode . . .</th>
<th>SSL server sockets are used for the . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>State-reset</td>
<td>AppServer broker and agents</td>
</tr>
<tr>
<td>State-aware</td>
<td>AppServer broker and agents</td>
</tr>
<tr>
<td>Stateless</td>
<td>AppServer broker only</td>
</tr>
<tr>
<td>State-free</td>
<td>AppServer broker only</td>
</tr>
</tbody>
</table>

Thus, for state-reset and state-aware modes, the AppServer broker accepts an initial SSL connection from the SSL client, then disconnects and assigns that client an SSL connection to an available AppServer agent. Because these connections are sequential, encryption overhead is essentially reduced to a single connection per SSL client-server exchange, even though the client connects to both the broker and agent. The only additional overhead is the brief exchange between the broker and agent to locate and hand off the agent connection to the client.

For stateless and state-free operating modes, the client is always connected to and exchanges all network communications with the AppServer broker, limiting encryption overhead to this single connection. All communications between the broker and agents occurs on the same system with no data broadcast on the network where it can be intercepted. Therefore, the communications between broker and agents is in clear text, eliminating any needless encryption overhead between them.
Identifying the SSL server connected to an AppServer client

When you establish an SSL session as part of the client connection to an AppServer, you can determine the identity of that server using an interface mechanism appropriate for the type of client, as shown in the following table.

Table 14: Determining SSL server identity for an AppServer connection

<table>
<thead>
<tr>
<th>Client type</th>
<th>Identifies its SSL server using the . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABL client</td>
<td>SSL-SERVER-NAME attribute on the server object handle</td>
</tr>
<tr>
<td>.NET or Java Open Client</td>
<td>Common method on the AppObject</td>
</tr>
<tr>
<td>Any client of an OpenEdge Web service</td>
<td>SSL interface supported by the Web service client platform</td>
</tr>
</tbody>
</table>

For ABL and OpenClients, the SSL server identity is returned as the SSL server's X.500 Subject name field (from the server certificate) after the SSL session as part of the AppServer connection has been established. For Web service client platforms, the method of identifying the SSL server is platform-determined, but is typically also the value of the SSL server's X.500 Subject name field.

For an Internet-secure AppServer or OpenEdge Web service, the SSL server is the Web server that hosts the AIA or WSA. For an SSL-enabled AppServer, the AppServer itself is the SSL server. For more information on SSL and OpenEdge SSL clients and servers, see OpenEdge Getting Started: Core Business Services - Security and Auditing.

Multi-language support

OpenEdge supports multi-language (for example, French and German) applications through use of the CURRENT-LANGUAGE statement and function. You can use this ABL support in AppServer procedures to coordinate different languages between the client and AppServer agent. In stateless or state-free operating mode, you must use the activate procedure to set the correct language for each client request.

If you do write a multi-language application, you must ensure that the client and your server-side code handle the following features in a compatible fashion:

- **Collation** — If the AppServer agent and client use different collations, the client might need to re-sort output TEMP-TABLE or result set parameters. An ABL client can do this by defining its own index on the appropriate columns.

- **Date and number formats ( -d and -E startup parameters)** — To minimize any problems using different date and number formats, always exchange dates as an ABL DATE values and exchange numeric fields as ABL INTEGER or DECIMAL values, rather than as character strings.

- **Case rules (such as in BASIC and French)** — Based on the anticipated language environment, you can specify tables at AppServer startup (using the -cpcase startup parameter) that the AppServer agent uses to handle case conversions.
• **Year offset \((-yy\) startup parameter)** — Using the Century \((-yy)\) startup parameter, you can ensure that the AppServer agent calculates dates based on a specified hundred-year period that is compatible with the client.

For more information on using the `CURRENT-LANGUAGE` statement and function in an ABL session, see *OpenEdge Development: Internationalizing Applications*.

## Distributed application design and implementation

This section provides some suggestions and techniques to write distributed applications using the AppServer, and to migrate your application code from your present configuration to a distributed AppServer agent environment.

### Understanding the scope of persistent, single-run, and singleton procedures

A remote procedure on the AppServer that you run with the `PERSISTENT`, `SINGLE-RUN`, or `SINGLETON` option, and a local procedure that you run with the `PERSISTENT` option, both perform in identical fashion with respect to scope. That is, the remote procedure's context is created when the instantiating procedure starts to execute, and that context persists after it returns until the end of the ABL session, or until it is explicitly deleted.

When a remote persistent, single-run, or singleton procedure is run, its context is managed strictly within the AppServer session where the procedure runs. This is consistent with the existence of the processing boundary that separates an AppServer session distinctly from a client application session. If a remote procedure creates other local ABL objects (persistent procedures), these objects are available only locally within the AppServer agent in which they were created. These objects cannot be shared between an ABL client application and the AppServer session.

#### Cleanup of remote procedure instances

Persistent, single-run, and singleton procedures differ significantly with respect to the removal of instances from the AppServer. You should code your application with the following behavior in mind:

• **Persistent procedures** — Simply delete the proxy handle on the client. This causes the remote instance to be deleted.

• **Single-run procedures** — The remote instance is automatically deleted after execution of each call to an internal procedure or user-defined function. The client should delete the local proxy handle.

• **Singleton procedures** — The remote instance must be explicitly deleted, typically by code within the remote procedure itself. Otherwise, the instance remains in memory, even after the client disconnects, until the AppServer agent shuts down. The client should delete the local proxy handle, but doing so does not affect the remote instance.
The behavior described above varies slightly if any client calls a remote procedure with run
single-run while the singleton instance of the same procedure is already running. In such a
scenario, the AVM deletes the singleton instance and runs the procedure in single-run mode. The
AVM does not re-instantiate the singleton until it is explicitly called by a client. Note that combining
single-run and singleton calls to the same procedure within the same application is not
recommended.

Understanding procedure handles in distributed ABL sessions

Every active ABL procedure, whether running on an ABL client or in an AppServer agent, has a
procedure handle associated with it. Among the various attributes of the procedure handle, the
proxy, remote, persistent, single-run, and singleton attributes combine to provide
important information about the procedure and its handle in a distributed application environment.

REMOTE attribute

The this-procedure system handle returns the current procedure handle value from within any
executing block of an external procedure. A current procedure handle created within an AppServer
session for a remote procedure request is a remote procedure handle. Its remote attribute is,
therefore, set to true. If the remote attribute is false, then the current procedure was created
as the result of a local procedure call (that is, a procedure call initiated and executed within the
current ABL session context).

PERSISTENT, SINGLE-RUN, and SINGLETON attributes

When you execute an external procedure from a local procedure call, as with any procedure, the
external procedure can reference its own context, using the this-procedure system handle.
This system handle returns a procedure handle to the external procedure’s own context.
Also, the procedure that executes the run statement that creates the external procedure can obtain
a reference to the same external procedure context, using the set option of the run statement.
This option returns the same procedure handle as the this-procedure system handle accessed
from within the external procedure.

Thus, for a local persistent, single-run, or singleton procedure, there is only one handle for a specific
external procedure context, but there might be many references to it. For any such reference to
an external procedure handle, the persistent, single-run, or singleton attribute (whichever
matches the option used with the run statement that instantiated or initialized the external
procedure) returns the value true.

PROXY attribute

When an ABL client application executes a remote persistent, single-run, or singleton procedure,
two procedure handles are created: one within the client application session, and another separate
handle within the AppServer session where the external procedure is created. OpenEdge internally
maintains a mapping between the two handles. The handle within the client application is a proxy
procedure handle; its proxy attribute and its persistent, single-run, or singleton attribute are set to true, but its remote attribute is false. The handle within the AppServer agent is a
remote procedure handle; its remote attribute and the persistent, single-run, or singleton
attribute are set to true, but its proxy attribute is false.
Unlike persistent procedure handles used for local procedures, the proxy procedure handle and the remote procedure handle are truly separate handles. For example, setting the PRIVATE-DATA attribute on a remote procedure handle has no effect on the PRIVATE-DATA attribute of the corresponding proxy procedure handle.

How the attributes work together

The following table summarizes the settings of the PROXY, REMOTE, and PERSISTENT procedure handle attributes for various types of local and remote procedure call scenarios.
<table>
<thead>
<tr>
<th>Procedure call type</th>
<th>Caller handle</th>
<th>Callee handle (THIS-PROCEDURE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local procedure</td>
<td>Not applicable</td>
<td>PROXY = FALSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>REMOTE = FALSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PERSISTENT = FALSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SINGLE-RUN = FALSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SINGLETON = FALSE</td>
</tr>
<tr>
<td>Remote procedure</td>
<td>Not applicable</td>
<td>PROXY = FALSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>REMOTE = TRUE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PERSISTENT = FALSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SINGLE-RUN = FALSE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SINGLETON = FALSE</td>
</tr>
<tr>
<td>Local persistent procedure</td>
<td>PROXY = FALSE</td>
<td>PROXY = FALSE</td>
</tr>
<tr>
<td></td>
<td>REMOTE = FALSE</td>
<td>REMOTE = FALSE</td>
</tr>
<tr>
<td></td>
<td>PERSISTENT = TRUE</td>
<td>PERSISTENT = TRUE</td>
</tr>
<tr>
<td></td>
<td>SINGLE-RUN = FALSE</td>
<td>SINGLE-RUN = TRUE</td>
</tr>
<tr>
<td></td>
<td>SINGLETON = FALSE</td>
<td>SINGLETON = FALSE</td>
</tr>
<tr>
<td>Remote persistent procedure</td>
<td>PROXY = TRUE</td>
<td>PROXY = FALSE</td>
</tr>
<tr>
<td></td>
<td>REMOTE = FALSE</td>
<td>REMOTE = TRUE</td>
</tr>
<tr>
<td></td>
<td>PERSISTENT = TRUE</td>
<td>PERSISTENT = TRUE</td>
</tr>
<tr>
<td></td>
<td>SINGLE-RUN = FALSE</td>
<td>SINGLE-RUN = FALSE</td>
</tr>
<tr>
<td></td>
<td>SINGLETON = FALSE</td>
<td>SINGLETON = FALSE</td>
</tr>
<tr>
<td>Local single-run procedure</td>
<td>PROXY = FALSE</td>
<td>PROXY = FALSE</td>
</tr>
<tr>
<td></td>
<td>REMOTE = FALSE</td>
<td>REMOTE = FALSE</td>
</tr>
<tr>
<td></td>
<td>PERSISTENT = FALSE</td>
<td>PERSISTENT = FALSE</td>
</tr>
<tr>
<td></td>
<td>SINGLE-RUN = TRUE</td>
<td>SINGLE-RUN = TRUE</td>
</tr>
<tr>
<td></td>
<td>SINGLETON = FALSE</td>
<td>SINGLETON = FALSE</td>
</tr>
<tr>
<td>Remote single-run procedure</td>
<td>PROXY = TRUE</td>
<td>PROXY = FALSE</td>
</tr>
<tr>
<td></td>
<td>REMOTE = FALSE</td>
<td>REMOTE = TRUE</td>
</tr>
<tr>
<td></td>
<td>PERSISTENT = FALSE</td>
<td>PERSISTENT = FALSE</td>
</tr>
<tr>
<td></td>
<td>SINGLE-RUN = TRUE</td>
<td>SINGLE-RUN = TRUE</td>
</tr>
<tr>
<td></td>
<td>SINGLETON = FALSE</td>
<td>SINGLETON = FALSE</td>
</tr>
<tr>
<td>Local singleton procedure</td>
<td>PROXY = FALSE</td>
<td>PROXY = FALSE</td>
</tr>
<tr>
<td></td>
<td>REMOTE = FALSE</td>
<td>REMOTE = FALSE</td>
</tr>
<tr>
<td></td>
<td>PERSISTENT = FALSE</td>
<td>PERSISTENT = FALSE</td>
</tr>
<tr>
<td></td>
<td>SINGLE-RUN = FALSE</td>
<td>SINGLE-RUN = FALSE</td>
</tr>
<tr>
<td></td>
<td>SINGLETON = TRUE</td>
<td>SINGLETON = TRUE</td>
</tr>
<tr>
<td>Remote singleton procedure</td>
<td>PROXY = TRUE</td>
<td>PROXY = FALSE</td>
</tr>
<tr>
<td></td>
<td>REMOTE = FALSE</td>
<td>REMOTE = TRUE</td>
</tr>
<tr>
<td></td>
<td>PERSISTENT = FALSE</td>
<td>PERSISTENT = FALSE</td>
</tr>
<tr>
<td></td>
<td>SINGLE-RUN = FALSE</td>
<td>SINGLE-RUN = FALSE</td>
</tr>
<tr>
<td></td>
<td>SINGLETON = FALSE</td>
<td>SINGLETON = FALSE</td>
</tr>
</tbody>
</table>
In the previous table, *caller handle* refers to the handle returned by the `RUN` statement. This column is relevant only for persistent, single-run, or singleton procedures, because you can only reference the procedure handle of a procedure that you call with a `RUN` statement that includes the `SET` option. *Callee handle* refers to the procedure handle within the procedure that is being executed. You can use the `THIS-PROCEDURE` system handle to reference this handle, regardless of how the current procedure was called.

### Understanding condition handling in distributed ABL sessions

The processing boundary that exists between an ABL client application session and an AppServer session influences how error conditions are viewed and handled in each session. The following table defines the four basic ABL conditions and describes how raising these conditions in one session affects the processing in the other, associated session.

**Table 16: ABL conditions**

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ERROR</strong></td>
<td>An unhandled <code>ERROR</code> condition raised in a client application has no effect on any of the AppServer agents to which it is connected. Conversely, an unhandled <code>ERROR</code> condition raised in an AppServer agent has no affect on the client application to which it is connected. Handling this condition conforms to standard ABL rules. For more information about these rules, see <em>OpenEdge Getting Started: ABL Essentials</em>. The only way an AppServer session can raise <code>ERROR</code> in the client application is for a remote procedure or an <code>Activate</code> procedure to execute the <code>RETURN ERROR</code> statement. This raises <code>ERROR</code> on the <code>RUN</code> statement in the client application.</td>
</tr>
<tr>
<td><strong>ENDKEY</strong></td>
<td>An unhandled <code>ENDKEY</code> condition raised in a client application has no effect on any of the AppServer agents to which it is connected. Conversely, an unhandled <code>ENDKEY</code> condition raised in an AppServer agent has no affect on the client application to which it is connected. Handling this condition conforms to standard ABL rules. For more information about these rules, see <em>OpenEdge Getting Started: ABL Essentials</em>. An AppServer session raises the <code>ENDKEY</code> condition if an attempt to find a database record fails or an <code>INPUT FROM</code> statement</td>
</tr>
<tr>
<td>Condition</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>reaches the end of an input stream. Like a batch client, there are no user-interface events that can raise the <strong>ENDKEY</strong> condition.</td>
<td></td>
</tr>
<tr>
<td><strong>STOP</strong></td>
<td>A <strong>STOP</strong> condition raised in a client application while it is executing a remote procedure causes a <strong>STOP</strong> condition to be raised in the context of the executing procedure in the AppServer session. An unhandled <strong>STOP</strong> condition in an AppServer session results in the request being returned with the <strong>STOP</strong> condition. <strong>On a root client application</strong> — An unhandled <strong>STOP</strong> condition causes OpenEdge to restart the client Startup Procedure ((-p)). This restarted session retains connections with databases, but deletes outstanding persistent procedures. This restart also disconnects any AppServer agents it is connected to and deletes active remote persistent procedures. <strong>In an AppServer agent</strong> — An unhandled <strong>STOP</strong> condition causes the <strong>STOP</strong> condition to be propagated back to the client application where it is raised again in the context of the outstanding remote procedure request. It does not cause a restart of the AppServer session. All active remote persistent procedures and AppServer agent connections remain intact.</td>
</tr>
<tr>
<td><strong>QUIT</strong></td>
<td>An unhandled <strong>QUIT</strong> condition raised in a client application disconnects the client application from all AppServer agents it is connected to and deletes the proxy procedure handles for that client application. An unhandled <strong>QUIT</strong> condition raised in an AppServer session results in the immediate, successful completion of the current remote procedure request. The client application is then automatically disconnected from the AppServer agent.</td>
</tr>
</tbody>
</table>

### Schema triggers

Schema trigger code defined in the database is executed by database clients. Any database access by an AppServer agent that invokes database trigger code causes the AppServer agent to run such code. This is because the AppServer application acts as a client to the database.

There is no functional difference between how schema triggers run in an AppServer session and how they run in an ABL client session. However, trigger code executed by an AppServer agent can:

- Fail if it requires user input or output to a user interface other than a character user interface
- Send any output to the AppServer log file, unless specifically directed otherwise

Invoking schema triggers on a client application raises a basic security issue. Code in the AppServer agent is protected because of *encapsulation* (the session barrier that isolates AppServer code from the client); code on your client application is not protected.
Conditions that make migrating your application model easier

Certain conditions related to your current application framework allow you to more easily take advantage of AppServer functionality. These conditions include any of the following:

- Your present server has adequate resources such as memory, speed, and disk space to be easily scaled up for use with the AppServer.

- Your current model has a predominance of small, well-structured, identifiable transactions. The AppServer favors a small transaction model. For more information, see Transaction and record management considerations on page 147.

- You are already using temp-tables to manipulate data before committing the data to the database. Or, if you are not using temp-tables, it is easy to get your present code to work with temp-tables rather than deal directly with your database.

- Your current model is already set up such that neither procedures nor persistent procedures rely on shared/global data or buffers as parameters. These types of information cannot be shared between a client application and an AppServer agent.

- Your current model already has a distinct division between the logic associated with the user interface and the logic associated with your application. This enables your code to be modular and portable, in keeping with how you will want your code to be deployed in a distributed environment using the AppServer.
Debugging

This chapter presents information about how to examine code associated with a distributed application that uses the AppServer.

For details, see the following topics:

• AppServer debugging
• Using distributed debugging
• Using remote debugging
• Summary of debugging requirements
• Examining the AppServer agent log file

AppServer debugging

The Debugger supports debugging application code in a distributed environment in two modes. These modes provide two different models for debugging your distributed application. No matter what mode you are using, standard debugging commands, such as executing one or more lines of a procedure and its subprocedures, breaking and continuing at specified lines, displaying data from a procedure that you are running, and modifying any of this data, can now be used to debug remote procedures.

The two debugging modes include:

• Distributed debugging on page 180
• Remote debugging on page 180
To initiate any debugging on an AppServer, you must run it with the `debuggerEnabled` property (in the `ubroker.properties` file) set to 1.

The following sections describe each of these modes. For a complete description of the Debugger, see *OpenEdge Development: Debugging and Troubleshooting*.

**Note:** If you have Progress Developer Studio for OpenEdge installed, note that, in addition to the standalone OpenEdge Debugger that is described in this book, OpenEdge provides remote debugging support in the Debug perspective of Progress Developer Studio for OpenEdge. The Debug perspective is an integral part of Progress Developer Studio for OpenEdge and works directly with the ABL Editor, making it easy to identify and fix problems as you work. To debug an application on an AppServer version prior to OpenEdge Release 11.0, you must use the standalone OpenEdge Debugger.

---

### Distributed debugging

The **distributed debugging** allows you to debug several associated ABL sessions within a distributed application environment. That is, one debugger session, running in one session within the distributed environment, controls the debugging process for all sessions associated with it. The procedure call stack reflects all procedures in all sessions involved, enabling the root client application to be aware of the AppServer sessions that it is debugging. The *root client application* is the client application that initiates the Debugger using distributed debugging.

You can use distributed debugging on AppServers running in all operating modes except the state-free operating mode.

You can run the Debugger using distributed debugging to debug local procedures, remote procedures, and remote procedures executed by other remote procedures. An AppServer agent that is once, twice, or further removed from the initial debugger session can be debugged in this manner. More information on this debugging mode follows in later sections.

### Remote debugging

In **remote debugging**, the Debugger runs in such a way that it has direct access to the ABL code in an AppServer session, by-passing any client code completely. Remote debugging is the only debugging mode that you can use to debug remote procedures that are:

- Executed by an Open Client (non-ABL) or Web service client application
- Executed asynchronously in an ABL client application

This mode restricts your debugging activities to debugging only AppServer configurations and remote application procedures; you cannot debug any procedure in the client application session. It allows you to debug application code running on an AppServer agent independently of the calling client application.

For example, an engineer who has developed a set of AppServer procedures does not have to examine the client code that his AppServer code will eventually support. When he is ready to debug his work, he only needs to examine his own code. In this situation, remote debugging is the ideal choice. When he debugs his code using this mode, he sees only his code on the AppServer agent.

In this debugging mode, the client application driving the AppServer agent is unaware that the Debugger process is running on the AppServer agent. That is, the procedure call stack only reveals entries up to the top level remote procedure call. It does not reveal anything about the client application.
If the AppServer agent where the engineer has developed his code calls remote procedures on other AppServer agents, he can debug these other remote procedures using distributed debugging, as a client of the other AppServer (see Distributed debugging on page 180).

You can also use the Debug perspective in Progress Developer Studio for OpenEdge to debug the code running on an AppServer invoked by the client application. You do so by attaching the Debug perspective to the AppServer through a host name and port number. For more information, see the Progress Developer Studio for OpenEdge online help.

Using distributed debugging

Distributed debugging is the mode you are most likely to use for ABL client applications that drive AppServers because it more naturally follows the structure of your distributed application. It allows you to fully examine your procedures just as they are invoked in this environment.

Note: You cannot use this mode for .NET and Java Open Client applications or for OpenEdge Web services.

From your root client application, you can initiate distributed debugging from the Procedure Editor or the AppBuilder Tools. Then you can debug remote procedures in any of the connected AppServer sessions. Next, you can set breakpoints, step into remote procedures, and examine program variables and buffers in an integrated, rather than a disjointed manner.

About the Debugger window

When you are using distributed debugging, the Debugger window and the procedure call stack help orient you to the procedures available to debug. For example, the title bar of the Debugger Listing Panel changes as you change the procedures that you are debugging. The following table indicates the dynamic information changes that occur in the title bar.

Table 17: Content of the Debugger Listing Panel title bar

<table>
<thead>
<tr>
<th>When you are stepping through a ...</th>
<th>Then the title bar reads ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local procedure, such as example.p.</td>
<td>example.p (LOCAL)</td>
</tr>
<tr>
<td>Remote procedure, such as remote.p, on an AppServer agent whose server handle's NAME attribute is RemoteServer1.</td>
<td>remote.p (ON SERVER RemoteServer1)</td>
</tr>
<tr>
<td>Remote procedure, such as distant.p, on an AppServer agent whose server handle’s NAME attribute is RemoteServer2, which was run from a procedure on RemoteServer1 (RemoteServer2 is an AppServer agent once removed from the root client application.)</td>
<td>distant.p (ON SERVER RemoteServer2 via RemoteServer1)</td>
</tr>
</tbody>
</table>

The procedure call stack reflects the entire call stack for the distributed application.
For example, if `example.p`, line 4, runs `remote.p`, and `remote.p`, line 3 runs `distant.p` and you are stepping through line 17, the procedure stack displays the following information:

```plaintext
example.p():4
remote.p(...):3 ON SERVER RemoteServer1
distant.p(...):17 ON SERVER RemoteServer2 Via RemoteServer1
```

**Note:** Line numbers do not appear in local client procedures.

Note that when using distributed debugging, the root client application must run in an environment that supports the Debugger. However, you can step into a procedure running on an AppServer installed through OpenEdge Client Networking that does not itself contain the Debugger.

### Restrictions on Debugger operation

Note the following restrictions when running the Debugger for a distributed application:

- You cannot execute a `RUN` command from the Debugger window command line while the current procedure is running in an AppServer session.
- The Debug menu Interrupt option is not supported when executing remote ABL code. For information on this option, see the section on the Debug menu in *OpenEdge Development: Debugging and Troubleshooting*.

### Using remote debugging

You use remote debugging to debug an AppServer application independently of the client application. With remote debugging, the Debugger does not have access to the client application that calls remote procedures in the AppServer session. When viewing the procedure call stack associated with a debugging session initiated in this mode, you never see procedure call stack entries for any ABL client application that might be driving the AppServer.

Because the Debugger is an ABL debugger only, you must use remote debugging to debug AppServer procedures that are driven by .NET and Java Open Client applications or that are invoked as OpenEdge Web services.

You must also use remote debugging if you want to step through code in any of the AppServer configuration procedures that you specify in the AppServer properties file (`ubroker.properties`). These include the:

- Startup procedure (`srvrStartupProc` property)
- Shutdown procedure (`srvrShutdownProc` property)
- Connect procedure (`srvrConnectProc` property)
- Disconnect procedure (`srvrDisconnProc` property)
- Activate procedure (`srvrActivateProc` property)
- Deactivate procedure (`srvrDeactivateProc` property)

The Debugger does not have access to these procedures with distributed debugging.
You can initiate a remote debugging session in one of two ways:

- **Attaching the Debugger to an AppServer session** on page 183 — Attach a session of the Debugger that you have already started to a running AppServer agent.

- **Initiating the Debugger from within an AppServer session** on page 183 — Initiate a session of the Debugger from code running on an AppServer agent.

This section describes the basic tasks required to initiate remote debugging. For more information on detailed debugging tasks, such as using the Debugger window to step through code and exiting the Debugger process, see *OpenEdge Development: Debugging and Troubleshooting*.

For information about using the Progress Developer Studio for OpenEdge Debug perspective for remote debugging, see the Progress Developer Studio for OpenEdge online help.

### Attaching the Debugger to an AppServer session

As when attaching to any ABL session, to attach the Debugger to an AppServer session, you must first make the AppServer agent debug-ready in one of the following ways:

- Use the `-debugReady` startup parameter for starting the agent

- From the command line, run the `proDebugConfig` tool

- From a Debugger session, use the Make-ready and attach option in the Attach to Process dialog box

For any method other than using the `-debugReady` startup parameter, you must identify the AppServer agent you want to debug by its process ID. For many debugging situations, you can configure the AppServer to run only one agent at a time, and thereby eliminate any need to choose among them.

Attaching to an AppServer agent presents the same Debugger user interface as when you connect to a client process. The one difference is that there are times when an AppServer is not executing any code. In this case, the Debugger indicates that it is waiting for an AppServer request before it takes control on the first executable line of ABL code.

For more information on remote debugging by attaching the Debugger to an AppServer session, see *OpenEdge Development: Debugging and Troubleshooting,*

For information on remote debugging by attaching the Progress Developer Studio for OpenEdge Debugger to an AppServer session, see the Progress Developer Studio for OpenEdge online help.

### Initiating the Debugger from within an AppServer session

To initiate the Debugger from an AppServer session, you must both invoke the Debugger process and set an initial breakpoint from within your remote ABL code.

### Invoking the Debugger from ABL

To invoke the Debugger from ABL, you must explicitly invoke the debugger using the methods of the `DEBUGGER` system handle executed within the AppServer code. For more information on the `DEBUGGER` system handle, see *OpenEdge Development: Debugging and Troubleshooting* and the reference entries for the handle, its methods, and its attributes in *OpenEdge Development: ABL Reference.*
Establishing an initial breakpoint

With remote debugging, you must programmatically set an initial breakpoint in the ABL code by using the `SET-BREAK()` method of the `DEBUGGER` handle. For information on the `DEBUGGER` handle, see *OpenEdge Development: Debugging and Troubleshooting* and *OpenEdge Development: ABL Reference*.

The following example shows how to use the `INITIATE()` and `SET-BREAK()` methods for the `DEBUGGER` handle:

```plaintext
DEFINE VARIABLE lReturn AS LOGICAL NO-UNDO.
lReturn = DEBUGGER:INITIATE().
/* SET-BREAK( ) with no parameters breaks at next line */
lReturn = DEBUGGER:SET-BREAK().
RETURN. /* Debugger gets control at this line */
```

Summary of debugging requirements

The following requirements must be satisfied before you can debug an AppServer using either distributed debugging or remote debugging:

- The Debugger is supported on all platforms that support ABL.
- You must run the `ProDebugEnable` command and configure the AppServer for debugging by setting the `debuggerEnabled` property to 1 in the AppServer properties file (`ubroker.properties`). If you do not start an AppServer with this property setting, you cannot use the Debugger to debug code on the AppServer.
- When you set up an AppServer for debugging, it does not activate the Debugger when the AppServer starts, but when a procedure on the AppServer is executed in the context of a debugging session.
- The Debugger (`-debug`) startup parameter and the `DEBUGGER:DEBUG( )` method do not function on the AppServer.
- To use the Debugger with:
  - Distributed debugging, you must start it on the platform where the ABL client is running
  - Remote debugging, you must start it on the platform where the AppServer is running
Examining the AppServer agent log file

When you investigate errors and troubleshoot code, examine the contents of the AppServer agent log (specified using the `srvrLogFile` property in `ubroker.properties`). It can contain OpenEdge system messages (informational, error, and warning) and output from ABL `MESSAGE` statements that are not assigned an output destination by the `OUTPUT TO KEEP-MESSAGES` statement. For more information, see the `OUTPUT TO` statement entry in *OpenEdge Development: ABL Reference*.

You can choose to track and record procedure information using the following properties in the `ubroker.properties` file:

- `srvrLoggingLevel`
- `srvrLogEntryTypes`

These properties tell an AppServer agent to put procedure call trace messages into the AppServer agent log file with different levels of detail and for different types of information. Specifically, they issue a start message when a remote procedure begins and an end message when a remote procedure returns. Then, when you see a system message in the AppServer agent log file, you can determine what procedure generated a specific message.

You can also troubleshoot functions of the AppServer broker in your application using the corresponding properties in the `ubroker.properties` file:

- `brkrLoggingLevel`
- `brkrLogEntryTypes`

For more information on using log files generated with these and other properties, see *OpenEdge Development: Debugging and Troubleshooting*. 

---

**Examining the AppServer agent log file**

When you investigate errors and troubleshoot code, examine the contents of the AppServer agent log (specified using the `srvrLogFile` property in `ubroker.properties`). It can contain OpenEdge system messages (informational, error, and warning) and output from ABL `MESSAGE` statements that are not assigned an output destination by the `OUTPUT TO KEEP-MESSAGES` statement. For more information, see the `OUTPUT TO` statement entry in *OpenEdge Development: ABL Reference*.

You can choose to track and record procedure information using the following properties in the `ubroker.properties` file:

- `srvrLoggingLevel`
- `srvrLogEntryTypes`

These properties tell an AppServer agent to put procedure call trace messages into the AppServer agent log file with different levels of detail and for different types of information. Specifically, they issue a start message when a remote procedure begins and an end message when a remote procedure returns. Then, when you see a system message in the AppServer agent log file, you can determine what procedure generated a specific message.

You can also troubleshoot functions of the AppServer broker in your application using the corresponding properties in the `ubroker.properties` file:

- `brkrLoggingLevel`
- `brkrLogEntryTypes`

For more information on using log files generated with these and other properties, see *OpenEdge Development: Debugging and Troubleshooting*. 

---

**OpenEdge Application Server: Developing AppServer Applications** 185
Connecting to AppServers Using a URL

This appendix describes how to code a URL for an intranet or the Internet. For details, see the following topics:

- Internet and intranet connections
- Connecting over an intranet
- Connecting over the Internet

Internet and intranet connections

An ABL client or Open Client can connect to an AppServer or application service using a URL that specifies a private network (intranet) connection or a Web (Internet) connection. In an ABL client, you specify this URL using the `-URL` connection parameter of the server handle `CONNECT( )` method. For more information, see Programming ABL Client Applications on page 87. In an Open Client, you specify this URL using the appropriate AppObject connect method for your Open Client type. For more information, see OpenEdge Development: Open Client Introduction and Programming.

You can also use the same URL syntax described in this appendix to connect an ABL client to an OpenEdge Adapter for SonicMQ BrokerConnect using one of the ABL-JMS API message service connection procedures. In the following sections, wherever “AppServer” is mentioned, you can substitute “OpenEdge Adapter for SonicMQ BrokerConnect (BrokerConnect),” except in the actual protocol specification of the URL syntax or where otherwise indicated. For more information on connecting to message services using the OpenEdge Adapter for SonicMQ, see OpenEdge Development: Messaging and ESB.
Connecting over an intranet

The AppServer (and BrokerConnect) provides several options for a client connecting over a company intranet. When you connect to the AppServer (or BrokerConnect), you have the option of connecting through a NameServer or connecting directly to the AppServer.

Connecting through a NameServer

When you connect over a company intranet using a NameServer, the NameServer process coordinates access to multiple AppServers. Each AppServer identifies the application procedures it supports using an application service name, which it registers with a particular NameServer. The AppServer connection parameter provides the network address of a NameServer and the name of an application service. The specified NameServer locates an AppServer that supports the specified application service and provides the network address of that AppServer to the client. The client then connects your application to this AppServer. Using the NameServer also provides location transparency and load balancing capabilities for an AppServer.

This is the URL syntax for connecting an AppServer (or BrokerConnect) through a NameServer over an intranet using the AppServer protocol (ignore line breaks):

```
[AppServer: | AppServerS:] [ //name-server-host ] [ :name-server-port ] / AppService-name
```

AppServer

The client uses a NameServer to connect to the AppServer.

AppServerS

The client uses a NameServer to connect to an Secure Sockets Layer (SSL)-enabled AppServer as an SSL client. When the NameServer provides an SSL connection to the client, the communications between the client and NameServer is in clear text (no SSL encryption) because the NameServer only passes a connection to the client and no application data is broadcast that can be intercepted. For more information on using an SSL-enabled AppServer, see the sections on using SSL in Design and Implementation Considerations on page 137.

name-server-host

TCP/IP host address or host name of the NameServer that holds the service registration.

name-server-port

TCP/IP port address for the NameServer specified with name-server-port.

AppService-name

Name of the application service (AppServer or BrokerConnect).
For example, the following three URLs connect to a warehouse inventory service or a financial portfolio management service through an OpenEdge NameServer. In the first example, the NameServer is located at the default UDP port, 5162; in the second, at a non-standard UDP port, 4099; and in the third, an SSL connection is established, as shown:

```
AppServer://dbserver/inventoryService
AppServer://dbserver:4099/inventoryService
AppServerS://dbserver/financialService
```

### Connecting directly to an AppServer

Connecting directly to an AppServer without a NameServer is desirable if the configuration for your application is simple and static and, as a result, you do not need location transparency or load balancing. For these configurations, not taking advantage of the optional NameServer means having less to configure and manage.

Connecting directly to an AppServer also eliminates conflicts that might arise when you do not want to use the User Datagram Protocol (UDP) that the NameServer uses. If you have restrictions that prohibit using UDP, you can connect directly to an AppServer to avoid these restrictions.

The following is the URL syntax for connecting directly to an AppServer (or BrokerConnect) over an intranet using the AppServer protocol (ignore line breaks):

```
```

**AppServerDC**

The client connects directly to the AppServer.

**AppServerDCS**

The client connects directly to an SSL-enabled AppServer as an SSL client. For more information about using an SSL-enabled AppServer, see the sections on using SSL in Design and Implementation Considerations on page 137.

**AppServer-host**

AppServer TCP/IP host address or host name that holds the service.

**AppServer-port**

AppServer TCP/IP port address for the AppServer specified with `AppServer-host`.

**AppService-name**

Name of the application service (AppServer or BrokerConnect).
For example, the following three URLs perform a direct TCP/IP connection to a shipping dock AppServer service or a salary management service. In the first example, the AppServer service is located at the default TCP/IP port, 3090; in the second, at a non-standard TCP/IP port, 3651; and in the third, an SSL connection is established, as shown:

AppServerDC://dbserver/shippingService
AppServerDC://dbserver:3651/shippingService
AppServerDCS://dbserver/salaryService

Connecting over the Internet

When you distribute your application over the Internet, you can connect to an AppServer (or BrokerConnect) using the HTTP or HTTPS protocol. Using HTTPS allows clients to communicate through firewalls and optionally send secure encrypted data transmissions across the Internet. To provide access to an AppServer (or BrokerConnect) over the Internet, you must install and configure the AppServer Internet Adapter (AIA). You can also configure the AIA to use a NameServer or access an AppServer directly. For information on configuring and managing the AIA, see OpenEdge Application Server: Administration.

Note: In situations where the client must access the Internet using a proxy server (client-side firewall), be sure to specify the proxy server and port for the AppServer (or BrokerConnect) client using the connection parameters or properties available for that client (ABL or Open Client). For more information, see the sections on configuring the AIA for proxy servers in OpenEdge Application Server: Administration.

Connecting using HTTP

The HTTP protocol specifies that OpenEdge use HTTP to connect the client to the AppServer indirectly, through a Web server running the AppServer Internet Adapter.

The following is the URL syntax to connect an AppServer (or BrokerConnect) using HTTP (ignore line breaks):

http://[user-id[:password]@web-server-host[:web-server-port]/aia-path[?AppService=service-name]]

user-id

User account name to use to connect to a Web server that hosts the AIA.

password

Password to use with user-id.

web-server-host

TCP/IP host address or host name of the Web server that hosts the AIA.
web-server-port

TCP/IP port address of the Web server that hosts the AIA. The default is 80 for HTTP and 443 for HTTPS.

aia-path

URL path on the Web server where the AIA is installed. The aia-path identifies the virtual path and servlet name used to invoke the JSE and AIA. It must include the following information:

- A virtual path the JSE has configured within the Web server to recognize and pass directly to the JSE, instead of looking for that directory on the Web server
- Name of the Java servlet to be invoked by the JSE

The specification of the path depends on your Web server, JSE, and AIA configuration. For more information, see the sections on AIA administration in OpenEdge Application Server: Administration and the documentation for your Web server and servlet engine.

service-name

Name of the application service (AppServer or BrokerConnect). The service-name identifies the application service to which the AIA connects. If you do not specify an application service for a NameServer, the AIA uses the default application service registered with NameServer to complete the connection. If the AIA is configured to access the AppServer directly, you have no need to specify an application service at all.

The following are examples of the AppServer URL format, where external/aia1 is the path to the AIA:

http://jones@aia.acme.com/external/aia1?AppService=Partners
http://jones:brownie@aia.acme.com/external/aia1?AppService=Partners

Connecting using HTTPS

The HTTPS protocol specifies that OpenEdge use HTTP tunneled through the Secure Sockets Layer (SSL) to connect the client to the AppServer indirectly through a Web server running the AppServer Internet Adapter. HTTPS works like HTTP; in addition, it uses SSL to encrypt all communications between the client and AppServer.

This is the URL syntax to connect an AppServer (or BrokerConnect) using HTTPS (ignore line breaks):

https://[user-id[:password]@]web-server-host[:web-server-port]/aia-path[?AppService=service-name]

The option descriptions are the same as for HTTP (see Connecting using HTTP on page 190.)
Note: For a .NET Open Client accessing a secure Web server that is running on the client machine, you must specify the Internet host name for the client machine as the value of web-server-host (not "localhost") exactly as it appears in the CN field of the Web server certificate, because .NET automatically validates the exact value of web-server-host against this server certificate field.

For HTTPS, the AIA instance provides a check on Internet connection security, according to the setting of its httpsEnabled property. During the connection process, if OpenEdge determines that the client application is attempting to connect to a secure Web server using HTTP and this httpsEnabled property is turned on (set to 1), OpenEdge silently returns the required HTTPS URL to the client for it to retry the connection.

Note: The default value for this AIA property is off (set to 0). For more information on AIA configuration for HTTPS, see the sections on AIA administration in OpenEdge Application Server: Administration.

To use HTTPS, the client must have a public key certificate available that validates its access to the Web server. For information on how OpenEdge supports public key certificates for ABL applications, see OpenEdge Getting Started: Core Business Services - Security and Auditing. For information on how to manage public key certificates for .NET, Java, and Web service Open Clients, see OpenEdge Development: Open Client Introduction and Programming.
Index

A

ABL browse widget
145
performance 145
ABL elements
52, 88
ABL client programming 88
AppServer programming 52
Activate procedure
62–63
overview 62
usage requirements 63
AIA connections 190
See also AppServer Internet Adapter (AIA)
APPEND option 145
Application services
33, 50–51, 94, 96
accessing 94
connecting 96
session-free
33, 50
AppServer programming 50
session-managed
33, 51
AppServer programming 51
AppServer agents
142
setting the quantity 142
AppServer Internet Adapter (AIA)
190
AppServer connections 190
AppServer log files
185
debugging 185
AppServer session
24–33
client interaction
24, 26, 28, 30
state-aware operating mode 26
state-free operating mode 30
state-reset operating mode 24
stateless operating mode 28
connection management
25–26, 28, 31
state-aware operating mode 26
state-free operating mode 31
state-reset operating mode 25
stateless operating mode 28
context management
25, 27, 29, 33
AppServer session (continued)
context management (continued)
state-free operating mode 27
state-reset operating mode 33
state-aware operating mode 25
stateless operating mode 29
performance
25, 27, 29, 32
state-aware operating mode 27
state-free operating mode 32
state-reset operating mode 25
stateless operating mode 29
AppServers
50–51, 57–58, 63, 66, 69, 74, 78, 84, 94, 96,
99, 102, 137–138, 142–143, 151, 155, 163–
164, 166–167, 171, 179, 183, 188–191
accessing 94
accessing a remote procedure name 57
AIA connections 190
collection procedures 58
configuring number of agents 142
connecting 96
collection parameters
99, 102
basic 99
session model 102
collections
188, 190
URL-based Internet 190
URL-based intranet 188
controlling entry points 63
debugging
179, 183
attachable debugger 183
Internet-secure
164, 166–167
adapters for 167
clients of 166
Intranet-secure
167
clients of 167
memory requirements 155
multi-language support 171
operating modes
138
choosing for configuration 138
performance considerations
137, 143
primary 137
secondary 143
programming for Open Clients 84
Asynchronous requests (continued) examples (continued)
  complete client and server 135
  synchronous comparison 132
executing 122
handling response 123
managing 122
mixing with synchronous 131
persistent procedures 130
queue size 47
queuing overview 43
results overview 43
running on a local SESSION 131
session models 43
session-free queuing model 45
session-managed queuing model 44
Attachable debugger 183
Attributes
ASYNC-REQUEST-COUNT
  88, 123, 130–131
coordinating request types 131
deleting persistent procedures 130
deleting server handles 131
PROCEDURE-COMPLETE event 123
Asynchronous and synchronous requests
  41, 132
examples 132
process control flow 41
ASYNCHRONOUS option 107
Asynchronous remote procedures
  107, 121, 124–126, 180
debugging 180
error handling
  121, 124
  compared to synchronous 121
obtaining return value 126
overview 107
returned parameters 125
Asynchronous request handles 89, 122
Asynchronous request object handle 89
Asynchronous request return values 126
Asynchronous requests
  43–45, 47, 118, 122–123, 126, 130–133, 135
ABL execution model 133
cancelling 126
debugging 131
deleting
  130–131
  asynchronous request handles 131
persistent procedures 130
server handles 131
disconnecting AppServers 118
examples
  132–133, 135
  asynchronous comparison 133

AppServer agent 52
AppServer client 91
NAME 91
NEXT-SIBLING 91
PERSISTENT 53, 91, 173
PERSISTENT-PROCEDURE 91
PROCEDURE-NAME 92
PROXY 92, 173
QUIT 92
REMOTE 53, 173
REQUEST-INFO 92
RESPONSE-INFO 93
SERVER 93
SERVER-CONNECTION-BOUND 53, 68

Index

-AppService connection parameter 97
ASYNC-REQUEST-COUNT attribute
  88, 123, 130–131
coordinating request types 131
deleting persistent procedures 130
deleting server handles 131
PROCEDURE-COMPLETE event 123
Asynchronous and synchronous requests
  41, 132
examples 132
process control flow 41
ASYNCHRONOUS option 107
Asynchronous remote procedures
  107, 121, 124–126, 180
debugging 180
error handling
  121, 124
  compared to synchronous 121
obtaining return value 126
overview 107
returned parameters 125
Asynchronous request handles 89, 122
Asynchronous request object handle 89
Asynchronous request return values 126
Asynchronous requests
  43–45, 47, 118, 122–123, 126, 130–133, 135
ABL execution model 133
cancelling 126
debugging 131
deleting
  130–131
  asynchronous request handles 131
persistent procedures 130
server handles 131
disconnecting AppServers 118
examples
  132–133, 135
  asynchronous comparison 133

AppServers (continued)
record management 151
returning values 74
secure configurations 164
session model programming 50
session-free programming 50
session-managed programming 51
SSL support 163
SSL-enabled 167
state-free context management 69
stateless context management 66
transaction management 78
URL-based
  188–191
HTTP connections 191
HTTPS connections 190
intranet direct AppServer connections 189
intranet NameServer connections 188

OpenEdge Application Server: Developing AppServer Applications
Attributes (continued)

SERVER-CONNECTION-BOUND-REQUEST 28, 53, 67
SERVER-CONNECTION-CONTEXT 29, 54, 59, 62, 67
SERVER-OPERATING-MODE 23, 54
TRANS-INIT-PROCEDURE 55, 81
TRANSACTION 55, 80

Audit trails 161

Automatic transactions 78–82
  controlling 80
  example 82
  implementing 79
  restarting 82
  terminating 81

B

Bindings 96
  See also Session-free connections
Bound connections 28, 31, 67
  setting for stateless 67
  state-free 31
  stateless 28
Browse widget 145

C

CA 163
CANCEL-REQUESTS( ) method 89, 126
CANCELLED attribute 89
Cancelling asynchronous requests 126
Certificate authority (CA) 163
CHAINED option 82
Client application 88, 94, 96, 107, 118, 120
  accessing AppServer resources 94
  connecting an AppServer 96
  creating a server handle 96
  deleting the server handle 120
  disconnecting AppServers and application services 118
  executing remote procedures 107
  session model programming 88
Client context identifier 69–70, 72, 105
  generating value 70
  multi-tier security models 72
  Progress.Lang.OERequestInfo class 69, 105
  ABL client session 105
  AppServer session 69
CLIENT-CONNECTION-ID attribute 89, 104
Client-server record lock contention 150
ClientContextId property 62, 69, 105
  Activate procedure 62
  Deactivate procedure 62
  on ABL client 105
  on AppServer agent 69
Compiled and precompiled code 145
Configuration procedures 58–59, 61
  Activate and Deactivate 61
  Connect and Disconnect 59
  overview 58
  Startup and Shutdown 58
Connect procedure 59, 61, 144
  overview 59
  performance 144
  usage requirements 61
CONNECT( ) method 59, 89, 96–97, 102–103, 144
  application arguments 102
  client connections 96
  Connect procedure 59
  connection parameters argument 97
  examples 103
  performance 144
CONNECTED( ) method 89
Connecting 96
  an AppServer or application service 96
Connection ID 55, 104
  accessing 55, 104
    on ABL client 55
    on AppServer client 104
Connection parameters 97, 102
  basic 97
  discussion 97
  session model 102
Connection pool 30, 40
  load balancing 40
  state-free operating mode Physical connections 30
    state-free operating mode 30
  -connectionLifetime connection parameter 100
Connections 190–191
  HTTP 190
  HTTPS 191
Context management 66, 69
Context management (continued)
state-free operating mode 69
stateless operating mode 66
CREATE SERVER statement 90, 96
CURRENT-REQUEST-INFO attribute 52, 69
CURRENT-RESPONSE-INFO attribute 52, 69

D
Database access 158
security 158
Deactivate procedure 62–63
overview 62
usage requirements 63
DEBUGGER system handle 183
Debugger window 181
debuggerEnabled property 184
Debugging 180–182, 184
See also Distributed debugging, Remote debugging
asynchronous requests 180
Debugger window 181
 Distributed debugging 180
general requirements 184
Remote debugging 180
restrictions 182
distributed debugging 182
See also Distributed debugging, Remote debugging
DEFAULT-COMMIT attribute 52, 81
DEFINE INPUT PARAMETER statement 90
DELETE OBJECT statement 52, 74, 90, 114, 120
DELETE PROCEDURE statement 52, 74, 90, 114
Deleting 114, 120, 130–131
asynchronous request handles 131
remote persistent procedures
114, 130
asynchronous requests 130
server handles 120, 131
asynchronous requests 131
Deployment 153–155
minimizing the effects of schema changes 153
on a network 155
server configurations and resources 155
using portable r-code 154
Design and implementation 153, 155–156, 163, 172
deployment 153, 155
designing your distributed application model 172
security 156, 163
designing your distributed application model 176
Designing your distributed application model 172, 178
migrating your application model 178
scope of persistent procedures 172
Digital certificates 163
-DirectConnect connection parameter 98
Disconnect procedure 60–61
overview 60
usage requirements 61
DISCONNECT( ) method 90, 118, 145
definition 118
performance 145
Disconnecting 119–120
session-free application service 120
session-managed AppServer 119
Distributed debugging 180–182, 184
additional considerations 181
definition 180
general requirements 184
restrictions 182
using 181
Distributed sessions 172–174, 177
persistent procedure scope 172
procedure handle effects 173–174
schema triggers 177
ENDKEY condition 75, 77, 176
on AppServer 75, 77
unhandled 77
Entry points to AppServerEntry points to AppServer, See Export list
ERROR attribute 90
ERROR condition 74–75, 77, 176
on AppServer 74–75, 77
unhandled 77
Error handling 74–75, 77, 120–121, 124, 176
ABL client, STOP condition 120
AppServer session 74–75, 77
ENDKEY condition 75
ERROR condition 74–75
QUIT condition 77
STOP condition 75
Error handling (continued)
- asynchronous remote procedures 121, 124
  compared to synchronous 121
  client application 120
distributed session interactions 176
- remote persistent procedures 121
  synchronous remote procedures 121
- throwing errors from AppServer to client 75
unhandled conditions 77
Event procedures 124
- asynchronous request results 124
defining 124
Event queue 43
EVENT-PROCEDURE attribute 90
EVENT-PROCEDURE-CONTEXT attribute 90
Events 92
- PROCEDURE-COMPLETE 92
Executing 107, 122
- asynchronous requests 122
  synchronous requests 107
Export list 63–66
- definition 63
  management 64
  resetting 65
  resolving remote procedure names 66
  run-time operation 64
  setting 65
EXPORT( ) method 52, 63–64, 157
- calling 64
  security for executing procedures 157
F
FIRST-ASYNC-REQUEST( ) method 90
FIRST-PROCEDURE attribute 90, 113
FIRST-SERVER attribute 90
FUNCTION statement 52, 85, 90
- specifying super procedure prototypes 85
  Statements 90
  FUNCTION 90
G
Garbage collection 146
General debugging requirements 184
H
- H connection parameter 97
Handles 52, 88–90, 92–94, 107, 122
- asynchronous request object 89, 122
  Asynchronous request object 89
  Proxy procedure 92
  SELF system 93
  server object 88
  Server object 94
  SESSION 52, 90, 107
HTTP protocol 190
HTTPS clients of an Internet-secure AppServer 166
HTTPS protocol 191
HTTPS-based security 164
httpsEnabled property 191
I
Implementation and design 153, 155–156, 163, 172, 176
- deployment 153, 155
- designing your distributed application model 172
  security 156, 163
IN option, running remote internal procedures 107
IN SUPER option 85
- initialConnections connection parameter 100
INITIATE( ) method 184
Internationalization 171
- multi-language support 171
Internet connections 190
Internet security 164, 166–167
  AppServer 164
  AppServer adapters 167
  clients 166
Intranet connections 188
- URL-based 188
Intranet security 167
  AppServer 167
  clients 167
IS-OPEN attribute 52
L
LAST-ASYNC-REQUEST( ) method 91
LAST-PROCEDURE attribute 91, 113
LAST-SERVER attribute 91
- -lkwtmo startup parameter 150
Load balancing 39–40
- configuring 40
Load balancing (continued)  local procedure call
session models 39
definition 73
LOCAL-VERSION-INFO attribute
52, 71, 91, 106
ABL client session 106
AppServer agent 52
AppServer agent session 71
AppServer client 91
Lock Wait Timeout (-lkwtmo) startup parameter 150
loggingLevel property 185
Logical connections 96
See also Session-free connections

M
Managing asynchronous requests 122
-maxConnections connection parameter 101
Memory requirements 155
Methods
52, 54–55, 59, 63, 81, 89–91, 96, 118, 126, 184
CANCEL-REQUESTS( ) 89, 126
CONNECT( ) 59, 89, 96
CONNECTED( ) 89
DISCONNECT( ) 90, 118
EXPORT 52
EXPORT( ) 63
FIRST-ASYNC-REQUEST( ) 90
INITIATE( ) 184
LAST-ASYNC-REQUEST( ) 91
SET-BREAK( ) 184
SET-COMMIT( ) 54, 81
SET-ROLLBACK( ) 55, 81
Migrating your current application model 178
Mixing synchronous and asynchronous requests 131
Multi-language support 171
Multi-tier security models 72

N
NAME attribute 91
Network deployment 155
NEXT-SIBLING attribute 91
NO-LOCK option 151
-nohostverify connection parameter 98
Normal ABL transactions
78–79
implementing 79
-nosessionreuse connection parameter 98
-nosClientDelay connection parameter 102
-nosClientMaxPort connection parameter 101
-nosClientMinPort connection parameter 101
-nosClientPicklistExpiration connection parameter 101
-nsClientPicklistSize connection parameter 101
-nsClientPortRetry connection parameter 102

O
Objects 112
passing as parameters between AppServer and
client 112
ON SERVER option 107
Open Clients 34, 84
programming the AppServer for 84
session models 34
Operating modes
23–24, 26–27, 30, 138–142
checking from AppServer ABL 23
choosing for configuration 138
state-aware 26
state-free 30
state-reset 24
stateless 27
trade-offs between
139–142
application characteristics 139
considering future requirements 142
handling large context 140
handling small context 141
limiting bound connections 141
with ample computer resources 142
understanding 23
Operating system security 163
Options
82, 85, 107, 145, 151
APPEND 145
ASYNCHRONOUS 107
CHAINED 82
IN (remote internal procedures) 107
IN SUPER 85
NO-LOCK 151
ON SERVER 107
PERSISTENT SET 107
TRANSACTION DISTINCT 107

P
Performance 138, 143–146
ABL browse widget 145
choosing a transaction model 143
choosing an operating mode 138
compiled and precompiled code 145
connect and disconnect 143
CONNECT( ) method 144
DISCONNECT( ) method 145
load balancing 143
Performance (continued)

- minimizing network traffic 138
- off loading resource-intensive processing tasks 138
- schema transmission 146

PERSISTENT attribute

53, 91, 173
- distributed sessions 173

PERSISTENT SET option 107

PERSISTENT-PROCEDURE attribute 91

-pf connection parameter 99

Physical connections

40, 96
- session-free connection pools 40
- session-managed connections 96

Physical memory requirements 155

PKI 163

Portable r-code, advantages 154

PREV-SIBLING attribute

92
- Attributes 92
- PREV-SIBLING 92

Private keys 163

Procedure handles 113
- See also Proxy procedure handles, Remote procedure handles

PROCEDURE statement

53, 85
- specifying super procedure prototypes 85

PROCEDURE-COMPLETE event 92, 123

PROCEDURE-NAME attribute 92

ProcedureName property 56

PROCESS EVENTS statement 92

ProDebugEnable command 184

Progress.Lang.OERequestInfo class

69, 105
- ABL client session 105
- AppServer session 69

Progress.Lang.OEVersionInfo class

71, 106
- ABL client session 106
- AppServer agent session 71

PROPATH environment variable

63, 66
- export list effects 63
- remote procedure name resolution 66

ProtoGen utility 84

PROXY attribute

92, 173
- distributed sessions 173

Proxy persistent procedure handles

73, 113–114
- accessing 113
- definition 73, 113
- local persistent procedure handles compared 114

Proxy procedure handle 92

Public key certificates 163

Public Key Infrastructure (PKI) 163

Q

QUIT attribute 92

QUIT condition

77, 176
- on AppServer 77
- unhandled 77

R

Record management

151
- buffer currency conflicts 151

REMOTE attribute

53, 173
- distributed sessions 173

Remote debugging

179–180, 182–184
- attaching to an AppServer session 183
- definition 180
- general requirements 184
- initial breakpoint 184
- initiating from an AppServer session 183
- invoking from ABL 183
- using 182

Remote persistent procedure handles

73, 113
- definition 73, 113

Remote persistent procedures

74, 114, 121
- deleting 74, 114
- from a ABL client 114
- from the AppServer session 74
- error handling 121

Remote procedure handles 73

Remote procedure name 56
- accessing 56

Remote procedures

107, 113, 115–118, 173
- asynchronous 107
- examples 115–118
- non-persistent 115, 118
- object as parameter 118
- persistent, calling a user-defined function 117
- persistent, calling an internal procedure 116
- general behavior 113
- PROXY and REMOTE attributes 173
- running 107
Remote procedures (continued)
synchronous 107
Request types
  39, 41–42
  comparing 41
  process flow 41
  session models 39
  specifying 42
REQUEST-INF0 attribute 69, 92
RequestId property
  71, 106
  on ABL client 106
  on AppServer agent 71
-rereadnolock startup parameter 152
Resolving remote procedure names 66
Response queue 43
RESPONSE-INF0 attribute 69, 93
RETURN ERROR statement
  59, 62
  Activate procedure 62
  Connect procedure 59
RETURN-VALUE function
  121
  across sessions 121
RUN statement
  93, 107, 174
  PROXY, REMOTE, and PERSISTENT attribute effects 174
RUN Statement
  121
  handling conditions 121
RUN statement (remote)
  107, 121
  asynchronous execution 107
  handling conditions 121
  run-time parameters 107
  synchronous execution 107
Run-time compilation
  163
  security 163
Running remote internal procedures 107
Running remote procedures
  107, 173
  persistently 107
  PROXY and REMOTE attributes 173
  single-run 107

S
-S connection parameter 98
Schema changes 153
Schema transmission 146
Schema triggers
  177
  distributed session effects 177
Scope of persistent procedures 172
Secure Sockets Layer (SSL) security
  163–164, 167, 170–171, 188–189, 191
  agent and broker connections 170
  AppServer configurations 164
  AppServer support 163
  Internet
    164, 191
  URL connections 191
  intranet
    167, 188–189
  URL connections direct to AppServer 189
  URL connections using NameServer 188
  SSL server identification to client 171
Security
  156–158, 161–163
  audit trails
    161–162
  AppServer application 162
  client application 162
  authentication and authorization 157
  state-reset and state-aware AppServers 157
  stateless AppServers 157
  database access 158
  export lists for AppServer access 158
  operating mode interactions 158
  operating system 163
  options 156
  run-time compiled code 163
  SSL 163
  user authentication and authorization 157
Security models
  72
  multi-tier 72
SELF system handle 93
Send queue 43
SERVER attribute 93
Server handles 89, 94
Server object handle 94
Server Pool Parameters
  142
  properties affected 142
SERVER-CONNECTION-BOUND attribute
  53, 68
  managing stateless context 68
SERVER-CONNECTION-BOUND-REQUEST attribute
  28, 53, 67
  managing stateless context 67
  stateless overview 28
SERVER-CONNECTION-CONTEXT attribute
  29, 54, 59, 62, 67
  Activate procedure 62
  Connect procedure 59
  Deactivate procedure 62
  managing stateless context 67
SERVER-CONNECTION-CONTEXT attribute (continued)

stateless overview 29
SERVER-CONNECTION-ID attribute
54–55, 59, 61–62, 67
Activate procedure 61–62
Connect procedure 59
Deactivate procedure 62
managing stateless context 67
SERVER-OPERATING-MODE attribute 23, 54
Session models
33–35, 39, 50, 88
ABL client programming 88
application services 33
AppServer programming 50
choosing 34
load balancing 39
Open Clients 34
requirements for using 35
SESSION system handle
51–52, 55, 59, 61–62, 67–68, 93, 107, 131
AppServer session 52
client session 93
EXPORT() method 52
local execution of asynchronous requests 131
local execution of remote procedures 107
SERVER-CONNECTION-BOUND attribute,
managing stateless connection context 68
SERVER-CONNECTION-BOUND-REQUEST
attribute, managing stateless connection context 68
SERVER-CONNECTION-CONTEXT attribute
59, 61–62, 67
managing stateless connection context 67
SERVER-CONNECTION-ID attribute
55, 67
managing stateless connection context 67
state-free operating mode 51
Session-free
33–35, 38–40, 50, 96, 104, 120
advantages 35
AppServers
33, 50
programming 50
choosing 34
connection example 104
connections 96
context management 38
disconnecting from application service 120
load balancing 40
operations 39
Session-managed
33–35, 39–40, 51, 96, 103, 119
AppServers
33, 51
programming 51
choosing 34
Session-managed (continued)
connection example 103
connections 96
converting to session-free 35
disconnecting from AppServer 119
load balancing 40
operations 39
-sessionModel connection parameter 100
SET-BREAK( ) method 184
SET-COMMIT( ) method 54, 81
SET-ROLLBACK( ) method 55, 81
Shutdown procedure
59
overview 59
usage requirements 59
SSL clients of an SSL-enabled AppServer 167
-ssl connection parameter 98
SSL server identification to clients 171
SSL-based security 167
SSL-SERVER-NAME attribute
94
Attributes
94
SSL-SERVER-NAME 94
Startup procedure
58–59
overview 58
usage requirements 59
State-aware operating mode
26, 139, 157, 159
authentication and authorization 157
definition and operation 26
design trade-offs 139
export list interactions 159
State-free operating mode
30–31, 51, 69, 140, 161
bound and unbound connections 31
definition and operation 30
design trade-offs 140
export list interactions 161
managing client context 69
SESSION handle attributes 51
State-reset operating mode
24, 139, 157, 159
authentication and authorization 157
definition and operation 24
design trade-offs 139
export list interactions 159
Stateless operating mode
27–28, 66–69, 140, 157, 160
authentication and authorization 157
bound and unbound connections 28
connection management
67–68
making bound 67
making unbound 68
Stateless operating mode (continued)
definition and operation 27
design trade-offs 140
export list interactions 160
managing client context 69
managing connection context 66

Statements
52–53, 59, 62, 74, 79, 85, 90, 92–94, 96, 107,
114, 120–121
CREATE SERVER 90, 96
DEFINE INPUT PARAMETER 90
DELETE OBJECT 52, 74, 90, 114, 120
DELETE PROCEDURE 52, 74, 90, 114
FUNCTION 52, 85
PROCEDURE 53, 85
PROCESS EVENTS 92
RETURN ERROR 59, 62
RUN 93, 121
RUN (remote) 107
TRANSACTION-MODE 79
WAIT-FOR 94
STOP attribute 94
Attributes 94
STOP 94
STOP condition
75, 77, 120, 151, 176
client-server lock contention 151
on ABL client 120
on AppServer 75, 77
unhandled 77
Super procedure prototypes 84
Synchronous and asynchronous requests
41, 132
examples 132
process control flow 41
Synchronous remote procedures 121
error handling 121
Synchronous requests 132
ABL execution model 132

T

THIS-PROCEDURE system handle 173
distributed sessions 173
Top level execution 113
TRANS-INIT-PROCEDURE attribute 55, 81
TRANSACTION attribute 55, 80
TRANSACTION DISTINCT option 107
Transaction initiating procedure 79
Transaction initiating procedure (continued)
definition 79
Transaction object 79
definition 79
initializing 79
Transaction object handle 80
TRANSACTION-MODE statement 79
Transactions 78–79, 143, 147–148, 150–151
See also Automatic transactions
buffer currency conflicts 151
choosing a model 143, 147
comparing normal ABL and automatic 147
distributed session conflicts 150
distributed session consistency 148
implementing 79
automatic 79
normal ABL 79
maintaining small size 148
managing 78
returning status to client 148
types 78
See also Automatic transactions
TYPE attribute 94
Attributes 94

U

Unbound connections 28, 31, 68
setting for stateless 68
state-free 31
state-less 28
-URL connection parameter 99, 187–191
Internet 190–191
HTTP protocol 190
HTTPS protocol 191
intranet 188–189
AppServer direct connection 189
NameServer connection 188
syntax 187

V

VersionInfo property 71, 106
OpenEdge Release for ABL client 71
VersionInfo property  
OpenEdge Release for ABL client (continued) 
on AppServer agent 71 
OpenEdge Release for AppServer 
71, 106 
on ABL client 106 
returned to ABL client 71

W
WAIT-FOR statement 94
Web services 34
session models 34