Progress®



Application Development with PAS for OpenEdge

OpenEdge®

Copyright

© 2021 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.

DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Defrag This, Deliver More Than Expected, DevReach (and design), Icenium, Inspec, Ipswitch, iMacros, Kendo UI, Kinvey, MessageWay, MOVEit, NativeChat, NativeScript, OpenEdge, Powered by Chef, Powered by Progress, Progress, Progress Software Developers Network, SequeLink, Sitefinity (and Design), Sitefinity, Sitefinity (and design), SpeedScript, Stylus Studio, Stylized Design (Arrow/3D Box logo), Styleized Design (C Chef logo), Stylized Design of Samurai, TeamPulse, Telerik, Telerik (and design), Test Studio, WebSpeed, WhatsConfigured, WhatsConnected, WhatsUp, and WS_FTP are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries.

Analytics360, AppServer, BusinessEdge, Chef Automate, Chef Compliance, Chef Desktop, Chef Habitat, Chef WorkStation, Corticon.js, Corticon Rules, Data Access, DataDirect Autonomous REST Connector, DataDirect Spy, DevCraft, Fiddler, Fiddler Everywhere, FiddlerCap, FiddlerCore, FiddlerScript, Hybrid Data Pipeline, iMail, JustAssembly, JustDecompile, JustMock, KendoReact, NativeScript Sidekick, OpenAccess, PASOE, Pro2, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, Push Jobs, SafeSpaceVR, Sitefinity Cloud, Sitefinity CMS, Sitefinity Digital Experience Cloud, Sitefinity Feather, Sitefinity Insight, Sitefinity Thunder, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, Supermarket, SupportLink, Unite UX, and WebClient 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.

July 2021

Product version: Progress OpenEdge 12.4

Updated: 2021/06/28

Table of Contents

Preface....9

PAS for OpenEdge and Client Interaction....11

Understand application models....11
Application model context management....13
Application model performance....13
Application models and Open Client Objects....13
Choose the application model....14
Context management and the session-free model....17
Decide the scope of individual requests....17
Understand synchronous and asynchronous requests....18
Process control flow....19
Specifying the type of remote request....19
Handling asynchronous request results....20
Asynchronous request queuing....20
Understanding the session-managed queuing model....21
Understanding the session-free queuing model....22
Asynchronous requests and internal procedures....24
Tuning the size of the send and response queues....24

Programming the Progress Application Server for OpenEdge....25

Programming for a PAS for OpenEdge application model....26
Session-free programming....26
Session-managed programming....27
ABL for programming PAS for OpenEdge procedures....27
Accessing the connection ID....32
Accessing the name of the current remote procedure....33
Accessing a procedure name on PAS for OpenEdge....34
Using PAS for OpenEdge event procedures....35
Startup and Shutdown procedures....35
Connect and Disconnect procedures....37
Activate and Deactivate procedures....39
Controlling PAS for OpenEdge entry points....41
Export list operation....41
Setting and resetting the export list....41
Calling the EXPORT( ) method....42
Resolving remote procedure names....43
Manage context for bound and unbound session-managed connections....43
Use the SERVER-CONNECTION-CONTEXT attribute....44

Use the SERVER-CONNECTION-ID attribute....44
Use the SERVER-CONNECTION-BOUND-REQUEST....44
Use the SERVER-CONNECTION-BOUND attribute....45
Manage client context for session-free and unbound session-managed connections....46
Pass the client context identifier between client and server....46
Generate a client context identifier....47
Manage context from the server with OERequestInfo objects....48
Implementing multi-tier security models....49
Understanding remote procedure handles....49
Remote procedure handles....50
Deleting remote procedures....50
Handling conditions and returning values....50
Raising the ERROR condition and returning values....50
Throwing error objects to the client....51
Raising the ENDKEY condition....51
Handling the STOP condition....51
Handling the QUIT condition....53
Effects of unhandled conditions....53
Managing transactions....54
Types of a PAS for OpenEdge instance transactions....54
Implementing normal ABL transactions....55
Implementing automatic transactions....55
Programming for Open Client applications....60
Using the ProtoGen utility....60
Using code editors....61

Programming ABL Client Applications....63

Programming for the application model....64
ABL for programming PAS for OpenEdge client procedures....64
Accessing PAS for OpenEdge resources from a client....70
Steps for accessing PAS for OpenEdge resources....70
Features of client-PAS for OpenEdge interactions....71
Connecting to a PAS for OpenEdge instance....71
Creating a server object handle....72
Establishing a connection with the CONNECT( ) method....72
Connection examples....76
Accessing the connection ID on a session-managed client....77
Accessing client context regardless of application model....78
Managing context from the ABL client with OERequestInfo objects....78
Running and managing remote procedures....79
Running remote procedures....80
Understanding proxy procedure handles....85
Deleting remote persistent procedures....86
Remote procedure code examples....87

Disconnecting from a PAS for OpenEdge instance....90
Using the DISCONNECT( ) method....91
Disconnecting a session-managed client....91
Disconnecting a session-free client....92
Deleting the server handle....92
Handling conditions and return values....92
Raising the STOP condition....92
Handling conditions on the RUN statement....93
Using the RETURN-VALUE function across sessions....93
Managing asynchronous requests....93
Executing an asynchronous request....94
Handling the response from an asynchronous request....94
Canceling asynchronous requests....98
Canceling asynchronous requests after a specified time....98
Asynchronous requests and persistent procedures....101
Deleting asynchronous remote persistent procedures....102
Deleting server handles....102
Deleting asynchronous request handles....102
Mixing synchronous and asynchronous requests....102
Running asynchronous requests on the SESSION system handle....102
Examples....103

Design and Implementation Considerations....107

Primary performance considerations....107
Minimizing network traffic....108
Off loading resource-intensive processing tasks....108
Choosing an application model....108
Setting the number of multi-session agents, sessions, and connections....112
Choosing a transaction model....113
Using load balancing....113
Secondary performance considerations....113
Connect and disconnect performance issues....114
Compiled compared to precompiled code....115
ABL browse design considerations....115
Performance issues related to schema transmission....116
Garbage collection for class-based objects....116
Transaction and record management considerations....116
Comparing normal ABL and automatic transactions....117
Ensuring a small transaction size....117
Returning transaction status to the client....118
Maintaining transaction consistency between client and server....118
Managing conflicts between client and server....120
Buffer currency and NO-LOCK....121
Deployment considerations....123

Minimizing the effects of schema changes....123
Advantages of using portable r-code....123
Security considerations....125
Overview of the server security options....125
User authentication and authorization....125
Database access....127
PAS for OpenEdge session access....127
Audit trails....130
Digital certificate management....131
Multi-language support....132
Distributed application design and implementation....132
Understanding the scope of persistent, single-run, and....132
Cleanup of remote procedure instances....133
Understanding procedure handles in distributed ABL....133
Understanding condition handling in distributed ABL sessions....136
Schema triggers....137
Conditions that make migrating your application model....138

Preface

Purpose

The purpose of this content is to describe the basic ABL features available for building PAS for OpenEdge business applications and any ABL client applications that access them, as well as supporting elements for Open Clients.

ABL application development for PAS for OpenEdge is not unlike that for the classic AppServer. However, PAS for OpenEdge supports the classic AppServer's session models (referred to as application models on PAS for OpenEdge) in a different way, because every PAS for OE instance supports both the session-managed and session-free application models simultaneously. Ultimately, you continue to code ABL in much the same way as you did previously, but manage the configuration and administration of the server and its applications very differently.

Audience

This content is for developers who need to build new OpenEdge applications for PAS for OpenEdge. As a developer, you already have experience building multi-tier ABL applications.

Organization

PAS for OpenEdge and Client Interaction on page 11

Describes the application models and options that you can use to build an ABL application to run on PAS for OpenEdge, and the interactions that a client can have with it.

Programming the Progress Application Server for OpenEdge on page 25

Describes how to program the PAS for OpenEdge, including both application procedures and the types of procedures that you can define to set up and manage the PAS for OpenEdge session context for your application procedures. It also describes how to manage context, including transactions, for the different application models and client bindings, and how to handle error information that you pass to client applications. Code examples illustrate specific programming points.

Programming ABL Client Applications on page 63

Describes how to program a client application, with emphasis on ABL clients. It explains how to connect to and disconnect from a PAS for OpenEdge instance, run and manage remote procedures, including maintaining context between server requests, and how to handle error information returned from the server. Code examples illustrate specific programming points.

Design and Implementation Considerations on page 107

Identifies and describes various issues related to the design and implementation of PAS for OpenEdge applications. This chapter highlights specific performance, deployment, and security issues, and introduces some programming techniques to help you write PAS for OpenEdge procedures.

Note:

For additional PAS for OpenEdge documentation, including white papers on advanced topics, see:

https://community.progress.com/community_groups/openedge_development/m/documents

---

Documentation conventions

See Documentation Conventions for an explanation of the terminology, format, and typographical conventions used throughout the OpenEdge content library.

1

PAS for OpenEdge and Client Interaction

This content describes how the Progress Application Server for OpenEdge (PAS for OpenEdge) interacts with a client to support the client's access to business applications. The topics describe how application models work in PAS for OpenEdge, and how to develop applications to handle context management.

For details, see the following topics:

- Understand application models
- Application models and Open Client Objects
- Choose the application model
- Context management and the session-free model
- Decide the scope of individual requests
- Understand synchronous and asynchronous requests

Understand application models

The basis of PAS for OpenEdge support for client interaction is the *application model*, which determines how a PAS for OpenEdge instance processes client requests. The PAS for OpenEdge supports two possible application models, which a client specifies when it connects to a PAS for OpenEdge instance:

- **Session-managed** — In this application model, by default, the PAS for OpenEdge instance provides an ABL session from a pool of available sessions to execute each request from the connected client. After each client request completes, the instance returns the session to the instance's session pool where it is available to handle requests by other clients. Thus, by default, a session-managed client is not bound to

any single PAS for OpenEdge session, and each request over this same client connection can be executed by a different PAS for OpenEdge session. Any context shared between the client and server must be identified and passed between them with each request. Optionally in the session-managed model, either the client or the business application running on the PAS for OpenEdge instance can also bind the client to a single ABL session when the client initially connects, or at any point during the same client connection. This single bound ABL session then executes requests only from the bound client, and can thereby maintain an exclusive client context, either until the client disconnects from the PAS for OpenEdge instance, or until the client or the business application terminates the binding while the client remains connected to the instance. Once an ABL session is unbound from a session-managed client, the PAS for OpenEdge instance returns the session to the instance's session pool where it is again available to handle requests by other clients. In the session-managed model, whether the client is unbound from or bound to a single server session, PAS for OpenEdge guarantees that all requests on the same client connection are executed sequentially with responses returned in the same order as the requests are sent. Session-managed clients can achieve some performance benefits by handling the responses from server requests asynchronously, but responses over the same client connection continue to be returned from the server in the same order as the requests are sent.

- **Session-free** — In this application model, the PAS for OpenEdge instance provides an ABL session from a pool of available sessions to execute each request from the connected client. After each client request completes, the PAS for OE instance returns the session to the instance's session pool where it is available to handle requests by other clients. Thus, the client is not bound to a single PAS for OE session for the duration of the client connection, and each request over this same client connection can be executed by a different PAS for OpenEdge session. Any context shared between the client and server must be identified and passed between them with each request. In the session-free model, PAS for OpenEdge can execute all client requests in parallel, even for the same client connection, with responses returned to the client as they become available, and not necessarily in the same order as the requests were sent. As a result, session-free clients can achieve additional performance benefits over similar session-managed clients by handling the responses from their server requests asynchronously. Note that while a session-free client can temporarily bind to a single server session, Progress Software does not recommend this practice.

Note that the default application model for a client connection is session-managed, and regardless of the application model, the client can make either synchronous or asynchronous requests to the PAS for OpenEdge instance, which the client manages depending on the application model.

Application model context management

PAS for OpenEdge can store ABL session context from the time a session Startup event procedure runs and initializes the session context, to the time a session Shutdown event procedure runs and cleans it up. ABL session context is defined by whatever state and data an ABL business application creates and applies to all PAS for OpenEdge clients. The business application can then manage this session context on a request by request basis. PAS for OE also supports session access to client context. Client context is client specific state and session data that is initialized at the time the client connects and a Connect event procedure runs, to the time the client disconnects and a Disconnect event procedure runs and cleans up this client state and data. Because PAS for OpenEdge's session-managed and session-free client connections optimize memory resources in the multi-session agent, each client's ABL event procedures and ABL requests can execute in any ABL session in any multi-session agent. You must therefore ensure that the ABL business application makes the client context available to any ABL session, running in any multi-session agent, and in any application model that your application supports. Both ABL session and client context persist in an ABL session across request boundaries. The aspect they share is that the ABL application is responsible for initializing them, maintaining them, ensuring the correct context is available for any given client's request, and for cleaning it up. PAS for OpenEdge supports both automatic and manual binding of a client to a single ABL session handling a client request (including client connection) in a multi-session agent. Automatic binding to the current ABL session handling a request happens when a PAS for OpenEdge client runs a persistent procedure, and the binding is removed when the client deletes that persistent procedure. Manual binding to a current session-managed ABL session is managed by the PAS for OpenEdge business application by setting and clearing the `SESSION:SERVER-CONNECTION-BOUND-REQUEST` attribute. When a client is bound to a single ABL session using either mechanism, the session can then maintain context in memory between client requests, and the PAS for OpenEdge business application can then support transaction management that allows a single database transaction to span multiple client requests."

Application model performance

For a session-managed application, a PAS for OpenEdge instance handles requests over the same client connection sequentially (single-threaded), guaranteeing that each request completes before the subsequent request is executed. For bound connections, the server responds as quickly as the general server load permits, while limiting the number of clients that the server can service at a time to the maximum number of ABL sessions available. For unbound connections, the server can handle a higher number of client requests, but with decreasing response as the number of clients increases for the same number of available ABL sessions to handle them. Session-managed clients might also be able to increase overall response by implementing an asynchronous request model to manage requests.

For a session-free application, a PAS for OpenEdge instance handles all client requests in parallel (multi-threaded). This can increase the overall response for the same number of clients over an unbound session-managed application, depending on the underlying hardware and software resources of the server and the resource requirements for each session-free request. Because PAS for OpenEdge runs all session-free requests in parallel, session-free clients must generally implement an asynchronous request model that verifies the order of responses, if response order matters.

Application 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 in a PAS for OpenEdge ABL application 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 SOAP 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 SOAP 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 SOAP Web service operation

These Open Client objects support each application model differently, and the choice of application model affects the client programming required to access the application server procedures.

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

Choose the application model

The application model you choose determines how you want PAS for OpenEdge to service clients, and that choice depends on the type of application you want to support.

Session-managed model

The session-managed model, especially with clients bound to ABL sessions, is specifically designed for business applications or SOAP 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 PAS for OpenEdge instance maintains context between requests for each client that it services, and each such client participates in a persistent connection to the application server. Any bound ABL session is dedicated to serving that one client until the binding between them is unbound.

In this application model, all client requests are single-threaded, meaning that the application server 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 business applications or SOAP Web services that return a complete result, starting and ending any transactions, in a single request. Thus, the server maintains no context for any client that it services. Requests from a session-free client are handled by any available ABL session in any available multi-session agent that supports the required business application. The session-free client (or the session-free SOAP Web service on behalf of a SOAP Web service client) has each of its server requests executed by an available ABL session chosen by the PAS for OpenEdge session manager from its pool of ABL sessions.. Thus, PAS for OE executes requests from all session-free clients as it receives them, and completes each request independently of any prior request.

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

Mixed application model

A mixed application model supports a business application that manages data transactions that span multiple requests (session-managed), while at the same time supporting an atomic request model for client operations that stand alone, such as independent queries on the data (session-free).

If your application requires a mixed application model, you have to identify the application model and client binding of each client request before servicing it, because any PAS for OpenEdge session can handle requests for clients running in any application model when the session is not bound to a single session-managed client. This makes the business application code more complex, especially for event procedures you have configured to manage client and session context across client connections and requests.

Making the final choice

As an aid to choosing an application 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

Point comparison	Session-free	Session-managed
Business application programming model	Every business application or SOAP Web service request is independent of every other request. No client or session context is maintained between the client and the servers involved in the requests. The use of `RUN PERSISTENT` to instantiate remote persistent procedures is not recommended. Instead, use the `SINGLE-RUN` or `SINGLETON` option (supported for ABL clients only). REST Web services implicitly instantiate procedure objects using the SINGLE-RUN or SINGLETON option, as part of defining them. The use of connection-based interfaces (SubAppObjects and ProcObjects) is not recommended. Any Connect or Disconnect event procedures in the PAS for OpenEdge configuration do not execute when the client connects or disconnects (respectively) from the PAS for OpenEdge instance.	Every business application or SOAP Web service request is associated with a single connected client and PAS for OpenEdge server. Client and session context can be maintained across client requests. Note that for a SOAP Web service, both the Web service client and the Web service must maintain awareness of the connection using SOAP headers. The client developer must program the handling of these headers in the client application. Any Connect or Disconnect event procedures in the PAS for OE configuration execute as the client connects and disconnects (respectively) from the PAS for OpenEdge instance.

Point comparison	Session-free	Session-managed
Client Compatibility	Compatible with all ABL, REST, and SOAP Web service clients.	Compatible with all ABL and SOAP Web service clients.
Communications between client and server	Clients of the PAS for OpenEdge instance exchange messages transparently with multiple ABL sessions that the session manager provides from its session pool. These multiple ABL sessions handle all client requests in parallel, depending on the server resources available.	Each client creates a separate and exclusive connection over which all messages are exchanged with one or more ABL sessions that the session manager provides from its session pool until the connection is terminated. This connection handles each client request in sequence, executing each request only after the previous request from the client has completed.

Thus, if your application requires the application server to manage context for its clients, choose the session-managed application model. Also, if you are exposing an existing business application that relies on a specific application model as a SOAP Web service,, you must continue to use that application model for the Web service unless you make the necessary changes in the application to use the other application model.

Note: If you expose the application as a REST Web service, you must ensure that the application uses the session-free model.

If your application requires no context management on the application server, you can choose the session-free application model. For an existing session-managed business application that meets these requirements (maintains no context), you can immediately change the application model to session-free with no code changes, thus converting the entire application with potentially greater performance benefits.

If you can use the session-free application model, it has the following advantages over the session-managed model:

- Both the business application and SOAP Web services scale much more readily compared to session-managed.
- The programming model for session-free applications—for both the business application and for the client application can be simpler than for session-managed applications, especially if the relative order of responses to requests does not matter and context management is minimal or non-existent.

For more information on programming:

- Session-free and session-managed business applications, see Programming the Progress Application Server for OpenEdge on page 25.
- Session-free and session-managed ABL clients, see Programming ABL Client Applications on page 63.
- 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 SOAP Web services, see *OpenEdge Development: Web Services*.

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

- Specify the same application model for any SOAP service definition based on the same business application. For more information on specifying the application model for a SOAP Web service definition using ProxyGen, see *OpenEdge Development: Open Client Introduction and Programming*.

- Manage the SOAP Web service through the OpenEdge Explorer or OpenEdge Management, or using the command-line tools, according to the same model. For more information on managing SOAP Web services for a given application model, see the sections on Web service management and deployment.

Context management and the session-free model

A session-free application server can provide limited context management using a client-instantiated persistent procedure (Open Client or SOAP Web service ProcObject instantiation) on the application server. 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 application model, except that you might require more parameters on a remote request to identify the context (such as a customer order) to be maintained.

OpenEdge supports a client context ID that can be passed to a business application on each request to identify the client that sent the request. In ABL, you can access this client context ID as the `ClientContextId` property of the `Progress.Lang.OERequestInfo` class. An instance of this class is available using properties of the server object handle in the client session and complementary properties of the `SESSION` system handle in the server session that handles the request. For more information, see Manage client context for session-free and unbound session-managed connections on page 46

Decide the scope of individual requests

When designing a business application, 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 business application 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 over a bound connection

For session-managed applications that use bound connections, 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 or unbound session-managed operations

For a session-free application or a session-manage application that uses unbound connections, 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 or for an unbound session-managed application between requests using input-output parameters provided by the business application procedures and SOAP Web service operations. You can also use the `SESSION:SERVER-CONNECTION-CONTEXT` attribute to pass character data from one ABL session to another that executes requests from the same connected client. An example might be an order entry SOAP 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.

When you use the `SESSION:SERVER-CONNECTION-CONTEXT` attribute to pass character data as session context, the session manager ensures that whatever value you assign to the attribute for one request is available in the ABL session that executes the next request from the same client." For more information, see Use the SERVER-CONNECTION-CONTEXT attribute on page 44.

Understand synchronous and asynchronous requests

An ABL client and PAS for OpenEdge instance 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 application server makes it available

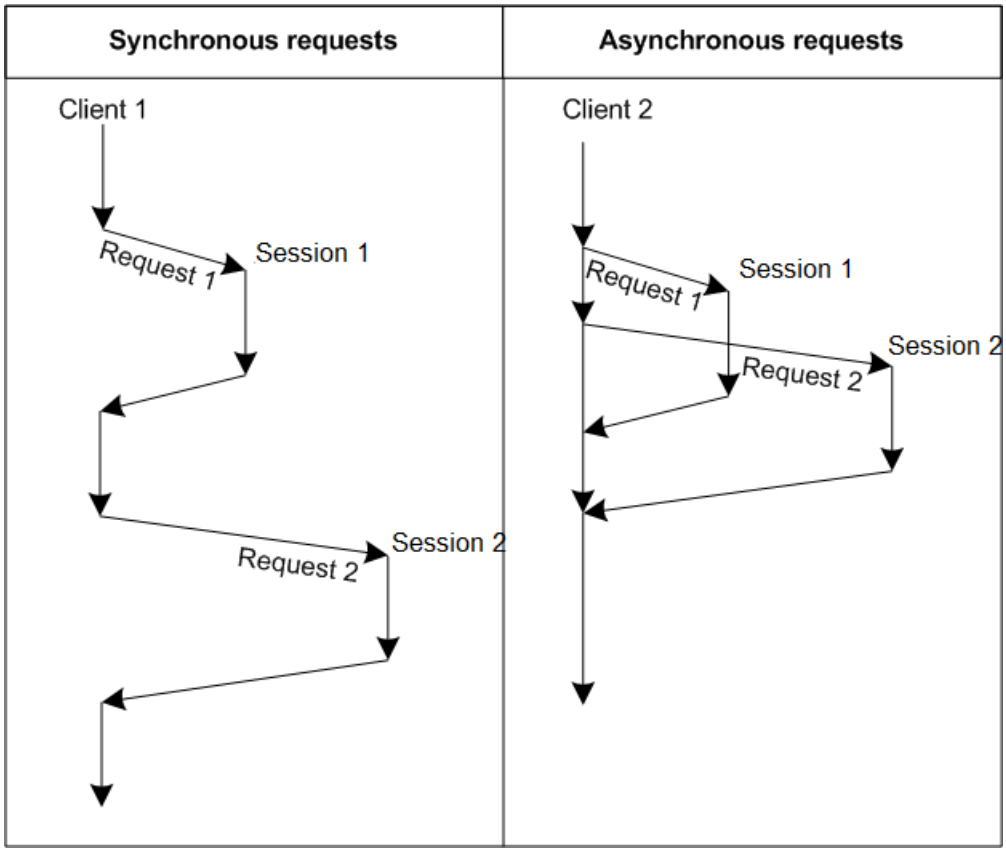
The ABL client also supports asynchronous requests to industry SOAP Web services, which work much like session-free asynchronous requests to an application server. For more information on ABL asynchronous requests to a SOAP Web service, see *OpenEdge Development: Web Services*.

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.

Figure 1: Synchronous and asynchronous requests



In this figure, one client (Client 1) makes synchronous requests and another client (Client 2) makes asynchronous requests to two PAS for OpenEdge sessions (Session 1 and Session 2). For Client 1, control flows in sequence between client and PAS for OpenEdge instance. For Client 2, control flows in parallel between client and PAS for OpenEdge instance. Assuming that the clients and PAS for OpenEdge instance perform the same work, Client 2 can conceivably complete the work in less time than can Client 1. The advantage for Client 2 comes with the parallelism that asynchronous requests can achieve. In this case, the actual advantage that Client 2 affords depends on its ability to respond to the results from the server sessions 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. A PAS for OpenEdge instance can handle both types of requests running in either a session-managed or session-free application model.

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 connection to a PAS for OpenEdge instance. OpenEdge also provides a mechanism for you to check the status of asynchronous requests for a session-managed or session-free connection.

Note: As suggested earlier, the ABL session agent that executes a request does not know if the request is synchronous or asynchronous. The session manager manages each request on behalf of the client, and dispatches the request for execution according to the application model. Consequently, the client can execute synchronous and asynchronous requests without regard to the application model.

Handling asynchronous request results

OpenEdge allows your client application to handle asynchronous request results using an asynchronous event procedure or method 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 callback event procedure or method receives any results from the asynchronous 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 or method signature must correspond to the signature of the remote procedure, you typically must define a unique event procedure or method to handle the results of 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 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 or session-free connection, OpenEdge maintains a queuing system to manage such requests between the client and the PAS for OpenEdge instance. OpenEdge handles multiple asynchronous requests in this queuing system differently, depending on the application model:

- **Session-managed connections** — The PAS for OpenEdge instance 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 connections** — The PAS for OpenEdge instance 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 application model.

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

- One *send queue* per application connection, for queuing asynchronous requests to the PAS for OpenEdge instance.
- One *response queue* per client connection, for queuing the results of each asynchronous request.
- The client *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 PAS for OpenEdge connections on a client.

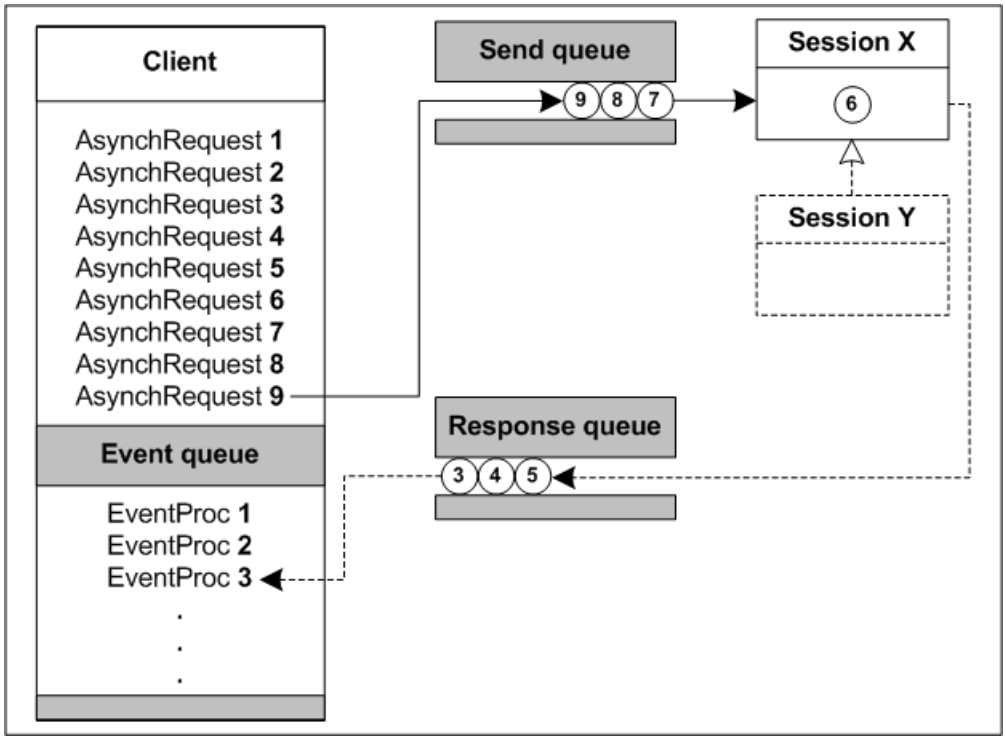
Understanding the session-managed queuing model

For a single session-managed PAS for OpenEdge 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 client 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 PAS for OpenEdge connection for asynchronous requests that have completed execution, but whose event procedure or method has not yet been executed. Finally, the client 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 for an unbound session-managed application, where **1** is the first asynchronous request (AsyncRequest **1**) a PAS for OpenEdge instance, 2 is the second asynchronous request (AsyncRequest 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 or method in response to the `PROCEDURE-COMPLETE` event handled from the ABL event queue. For simplicity, the example assumes that the PAS for OpenEdge instance as only a single multi-session agent with two sessions, Session X and Session Y, to execute requests.

On this unbound session-managed connection, if an asynchronous request is submitted for execution when there is already a request executing on the application server, 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 **7**, **8** and **9**. The running asynchronous request (in this case, 6) was dispatched to an available server session (Session X) for execution after the immediately prior request (in this case, 5) completed execution. Note that only one session at a time executes requests for a session-managed connection, in this case Session X. However, any available session (such as Session Y) can execute the next request for an unbound connection.

As each request completes execution, the response to the request is queued on the appropriate response queue. 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 or method can be executed in the context of a PROCESS EVENTS or blocking I/O statement. In the previous figure, the PROCEDURE-COMPLETE events have fired in order, and have already executed the event procedure or method for AsyncRequests **1** and **2**, is executing the event procedure or method for AsyncRequest 3. The responses for AsyncRequests **4** and **5** are still pending in the response queue.

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

Note that the order in which a session-managed client handles asynchronous responses from multiple PAS for OpenEdge connections is undefined. Thus, OpenEdge guarantees that the responses from asynchronous requests on the same PAS for OpenEdge connection are handled in the order they are sent to the client. However, it does not guarantee the order in which responses are handled from multiple PAS for OpenEdge connections.

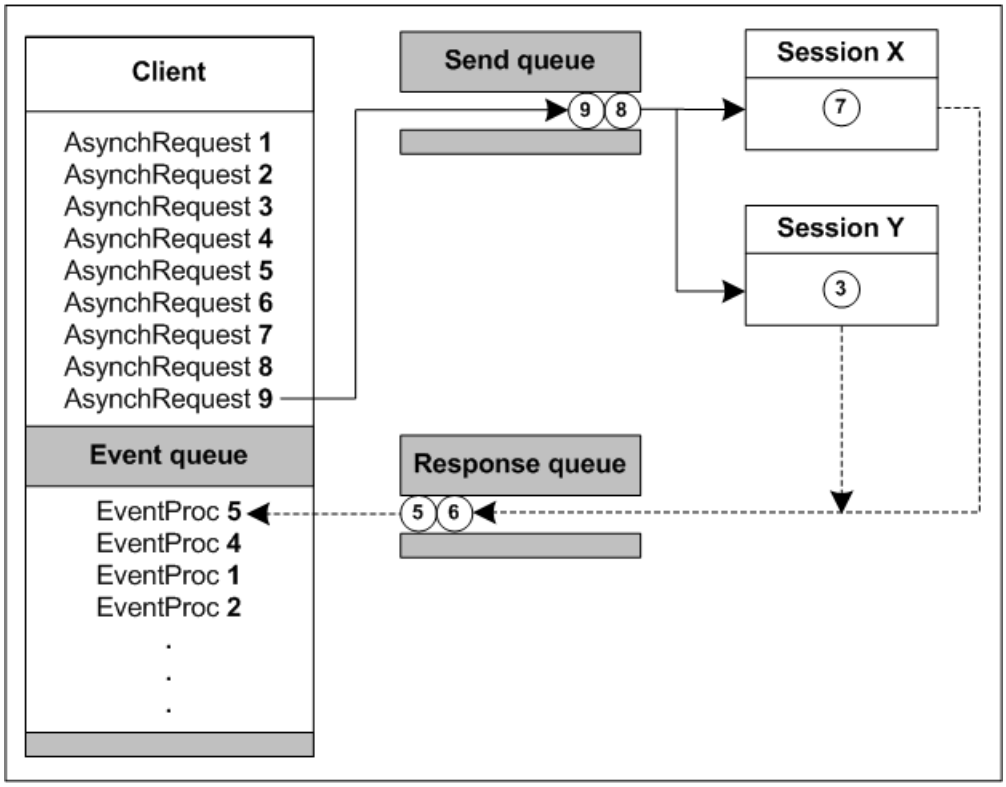
Understanding the session-free queuing model

For a single session-free PAS for OpenEdge connection, asynchronous requests are executed in no determined order, and the event procedures or methods for those requests are executed in no determined order. Unlike requests sent over session-managed connections, which are handled sequentially, requests sent over session-free connections are handled in parallel and completed in a manner that depends entirely on available PAS for OpenEdge 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 business application connection for requests that have completed execution. Finally, the client event queue queues the PROCEDURE-COMPLETE event for each completed request as it is received from the response queue. However, it is the availability of PAS for OpenEdge resources in its session pool that determines when and how these requests are actually processed.

The following figure shows how this works for a session-free application, with nine asynchronous requests submitted on the client. In this example, the requests are assumed to be sent for a single PAS for OpenEdge instance.

Figure 3: Session-free asynchronous request queuing



For simplicity, the example assumes that the PAS for OpenEdge instance has only two ABL sessions (Session X and Session Y) to execute requests. However, the principle is the same multiple PAS for OpenEdge instances executing requests sent through a Domain Name Server (DNS) load balancer. The main difference is that multiple PAS for OpenEdge instances are likely to provide higher general availability for the business application. AsyncRequest refers to the execution of an asynchronous `RUN` statement, and EventProc refers to the execution of an event procedure or method in response to the `PROCEDURE-COMPLETE` event handled from the ABL event queue. The requests are numbered in the order they are sent from the client.

On a session-free connection, if an asynchronous request is submitted for execution when all PAS for OpenEdge 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 the PAS for OpenEdge instance for execution once AsyncRequest **8** has been submitted to an available session for execution. All prior asynchronous requests have completed execution (**1**, **2**, **4**, **5**, and **6**) or are executing in the available sessions (**3** and **7**).

Note that although these requests have been submitted by the client in order from **1** to **9**, AsyncRequest **3** is still being executed, while 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 in session Y that AsyncRequests **4** through **6** have already completed execution in session X and AsyncRequest **7** is still executing in session X.

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 in session Y. Note that when this series of requests began, AsyncRequests **1** and **2** would have executed at the same time in sessions X and Y, but **2** clearly completed before **1** and passed its results back ahead of it because its results were the first to appear in the event queue.

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 sessions X and Y were running in two separate PAS for OpenEdge instances accessed through a DNS load balancer, the order of completion might be exactly the same, especially if the load balancer balanced the load evenly between them. The one feature that could very well change the order of completion between the two-server and the one-server scenarios is if the two-server scenario balanced the load significantly differently between the two PAS for OpenEdge instances.

Asynchronous requests and internal procedures

The parallelism of execution and the order of request completion shown in the previous example of session-free asynchronous request queuing 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 session bound to the client by the remote persistent procedure (see Context management and the session-free model on page 17.

The effect is essentially the same as if the requests were invoked on a session-managed connection (see the previous example of session-managed asynchronous request queuing). However, any other asynchronous requests to non-persistent external procedures over the session-free connection 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 for a PAS for OpenEdge connection.

The Async Queue Size parameter only applies when you create the PAS for OpenEdge 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 PAS for OpenEdge instance. Thus, you can maximize the PAS for OpenEdge performance by choosing a value based on the amount of data that you expect to flow between the client and the PAS for OpenEdge instance 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*.

2

Programming the Progress Application Server for OpenEdge

This chapter describes how to program procedures that execute on the Progress Application Server for OpenEdge. For details, see the following topics:

For details, see the following topics:

- Programming for a PAS for OpenEdge application model
- ABL for programming PAS for OpenEdge procedures
- Accessing the connection ID
- Accessing the name of the current remote procedure
- Using PAS for OpenEdge event procedures
- Controlling PAS for OpenEdge entry points
- Manage context for bound and unbound session-managed connections
- Manage client context for session-free and unbound session-managed connections
- Understanding remote procedure handles
- Handling conditions and returning values
- Managing transactions
- Programming for Open Client applications

Programming for a PAS for OpenEdge application model

The most fundamental decision to make when designing and programming a PAS for OE business application is what application model it will use, session-free or session-managed, or even if the application will use a mixed model. The decision of which application model to use, and how, depends on your particular application requirements, which cover all areas of the application, including network capabilities, available PAS for OpenEdge resources, and critical application functions.

Once you have determined the application model to use, you must design and build the application server procedures to support it. Typical applications (especially Open Client and SOAP Web service applications) start with development of a business application that you then make available to the client application using the appropriate ABL, Open Client Toolkit, or SOAP 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 application model, or models, that you choose fundamentally determines the options for application development that are available to you. For information on the differences between these two application models, see PAS for OpenEdge and Client Interaction on page 11.

Note: This chapter addresses only programming issues specific to application server procedures. It assumes that you are generally familiar with writing and executing ABL procedures. For more information on using ABL, see *OpenEdge Getting Started: Guide for New Developers*.

Session-free programming

You can develop a session-free application similar to an unbound session-managed application. However, a session-free application must follow these additional PAS for OpenEdge programming requirements and recommendations:

- Every request (external procedure) runs independently of every other request on the PAS for OpenEdge instance. 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 PAS for OpenEdge instance (with DNS load balancing) or ABL session.
- You cannot specify a Connect or Disconnect event procedure, and none is ever executed on the application server.
- You can specify Activate and Deactivate event 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 SOAP Web service ProcObjects) that are called and instantiated directly by the client forces a bound connection to the application server, which limits application performance and scalability (see Affecting application scalability on page 27). 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 PAS for OpenEdge 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 PAS for OpenEdge session in response to another client invocation, because the latest remote external procedure is likely to execute in a different PAS for OpenEdge session than the one in which any of the PAS for OE-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 SOAP Web service ProcObject). If enough clients simultaneously create and use instances of this persistent procedure, the PAS for OpenEdge instance might have no more ABL sessions available in its session pool to serve additional clients.

Thus, if all clients that use the business application are always required to create an instance of this remote persistent procedure, the business application 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 business application, any clients that do so reduce the availability of network resources to other clients and limit the scalability of the business application accordingly.

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

SESSION handle attributes and session-free programming

The `LOCAL-VERSION-INFO`, `CURRENT-REQUEST-INFO`, and `CURRENT-RESPONSE-INFO` attributes on the `SESSION` handle support client context management for the session-free model.

The functionality of the following `SESSION` handle attributes is modified or not supported with the session-free model:

- `SERVER-CONNECTION-BOUND-REQUEST`
- `SERVER-CONNECTION-BOUND`
- `SERVER-CONNECTION-ID`

For more information, see ABL for programming PAS for OpenEdge procedures on page 27.

Session-managed programming

The requirements for programming an application server for session-managed applications differ depending on whether a bound or unbound client connection is used. However, both types of client connections provide some means to manage session context internally on behalf of clients that are connected using the session-managed application model.

The remaining sections in this chapter indicate where a feature's function or availability depends on the application model and binding state However, in general, the main differences affect how you can manage client connection context for each model. For more information on the session-managed application, see PAS for OpenEdge and Client Interaction on page 11 .

For more information on programming for the session-managed model, see Connect and Disconnect procedures on page 37 and Manage context for bound and unbound session-managed connections on page 43.

ABL for programming PAS for OpenEdge procedures

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

Table 2: ABL for programming PAS for OpenEdge procedures

ABL element	Description
CLIENT-TYPE	A CHARACTER attribute on the SESSION system handle that returns the value, MULTI-SESSION-AGENT in a PAS for OpenEdge session. Valid only if the REMOTE attribute is TRUE.
CURRENT-RESPONSE-INFO	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.
DEFAULT-COMMIT	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.
DELETE OBJECT *handle*	A statement that you can use to delete certain session objects, including local and remote persistent procedures.
DELETE PROCEDURE *procedure-handle*	A statement that you can use to delete both local and remote procedure objects (persistent procedures).
EXPORT ([*list*])	A method on the SESSION system handle that creates and modifies the list of remote procedures provided by the PAS or OE instance.
FUNCTION ... IN SUPER	A statement that specifies the prototype definition for a user-defined function defined within a super procedure.
IS-OPEN	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.
LOCAL-VERSION-INFO	An object reference attribute on the SESSION system handle that references a Progress.Lang.OEVersionInfo object, which identifies the current instance of the OpenEdge server AVM, including its OpenEdge version.
OEClientType	A CHARACTER property on the Progress.Lang.OEVersionInfo class that returns the value, MULTI-SESSION-AGENT, when the reference to the OEVersionInfo object is returned as the value of the LOCAL-VERSION-INFO attribute on the SESSION system handle in a PAS for OpenEdge session, or as the value of the VersionInfo property on an OERequestInfo object returned by the appropriate attributes on a server object handle for a PAS for OpenEdge instance or on the SESSION handle in a PAS for OpenEdge session.
Parameters for procedures and user-defined functions	Procedure and user-defined function parameters can be defined as any ABL parameter data type, including class-based object

ABL element	Description
	types. (For more information see Passing class-based objects as parameters between a server and ABL client on page 84.) However, note that for any procedure or user-defined function that you include in an OpenEdge SOAP Web service, you cannot pass a static or dynamic ProDataSet™ parameter that includes parent-id data-relations (for example, data-relations as defined using the `PARENT-ID-RELATION` option of the `DEFINE DATASET`

ABL element	Description
	statement). Otherwise, ProxyGen fails to generate any proxies for the specified service definition.
PERSISTENT	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.
PROCEDURE ... IN SUPER	A statement that begins the prototype definition for an internal procedure defined within a super procedure.
PROCEDURE ... THREAD-SAFE	A statement that begins the prototype definition for a routine in a Windows DLL or UNIX shared library, where the THREAD-SAFE option declares that it as thread safe. You must know that the routine is, in fact, thread safe in order to specify this option, which simply informs OpenEdge that this is the case. ABL sessions of a PAS for OpenEdge instance can execute a given DLL or shared library routine only if it is thread safe.
Progress.Lang.OERequestInfo	A built-in ABL class that is passed as an object between a client and any ABL session that executes a client request on a PAS for OpenEdge instance, and contains information about the request, including identification, security, and context information. When the reference to an OERequestInfo object is returned by the CURRENT-REQUEST-INFO or CURRENT-RESPONSE-INFO attributes on the SESSION handle in a PAS for OpenEdge session, the AgentId, SessionId, and ThreadId properties return integer values that identify the multi-session agent, ABL session, and thread, respectively, that are executing the current client request.
QUIT	A statement that when executed as the last statement in the Disconnect event procedure of a bound session of a session-managed connection, cleans up the context of the session as the client disconnects, similar to a session-managed client disconnecting from an OpenEdge PAS for OpenEdge instance running in the session-managed application model.
REMOTE	A LOGICAL attribute on the SESSION system handle that is TRUE if the current session is running in the context of an PAS for OpenEdge instance. 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 PAS for OpenEdge session (as the result of a remote procedure call by a client application). Always FALSE if PROXY is TRUE.
SERVER-CONNECTION-BOUND	A LOGICAL attribute on the SESSION system handle that is TRUE if the PAS for OpenEdge session is bound to a particular client session in the session-free or an unbound session-managed model by client invocation of a remote persistent procedure or

ABL element	Description
	Open Client/Web service ProcObject. By default, it is FALSE. Valid only if the REMOTE attribute is TRUE.
SERVER-CONNECTION-BOUND-REQUEST	A LOGICAL attribute on the SESSION system handle that, when set to TRUE, requests that the PAS for OE session be bound to the current client connection identified by the SERVER-CONNECTION-ID attribute. When set to FALSE, requests that the PAS for OpenEdge session be unbound from the currently bound client connection pending deletion of all remote persistent procedures running in the session. In session-free, this attribute always has the Unknown value (?). Valid only if the REMOTE attribute is TRUE and the application model is session-managed. Any attempt to set this attribute in the session-free model raises a WARNING condition in the PAS for OpenEdge session, which writes a message to the PAS for OpenEdge 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.
SERVER-CONNECTION-CONTEXT	A CHARACTER attribute on the SESSION system handle that contains an application-determined value that you can set. OpenEdge passes this value to each server session 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 and the application model is session-managed.
SERVER-CONNECTION-ID	A CHARACTER attribute on the SESSION system handle that returns the run-time connection ID of the current client connection assigned to this Server session. Valid only if the REMOTE attribute is TRUE and the application model is session-managed. In the session-free model, this attribute has no meaning.
SERVER-OPERATING-MODE	A CHARACTER attribute on the SESSION system handle that returns the application model of the client request that the current

ABL element	Description
	PAS for OpenEdge session is running: "session-managed" or "session-free". Valid only if the REMOTE attribute is TRUE.
SET-COMMIT( )	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.
SET-ROLLBACK( )	A method on a transaction object handle that tells the transaction object to roll back any automatic transaction when the current request completes and returns execution to the client.
SINGLE-RUN	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.
SINGLETON	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.
THREAD-SAFE	A LOGICAL attribute on a call object handle that represents a Windows DLL or UNIX shared library routine that is TRUE if the PROCEDURE statement that declares the routine prototype is specified with the THREAD-SAFE option.
TRANSACTION	A HANDLE attribute on procedure handles and the THIS-PROCEDURE system handle that provides a handle to the current transaction object.
TRANS-INIT-PROCEDURE	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.

Accessing the connection ID

As described in ABL for programming PAS for OpenEdge procedures on page 27, OpenEdge creates a unique ID for each connection to a PAS for OpenEdge instance. For session-managed applications, there is a connection ID mechanism especially for a PAS for OpenEdge running in the unbound session-managed model. 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 PAS for OpenEdge instance.

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

For more information on using the SERVER-CONNECTION-ID, see Manage context for bound and unbound session-managed connections on page 43. For more information on using the ClientContextId property, see Manage client context for session-free and unbound session-managed connections on page 46.

Accessing the name of the current remote procedure

A PAS for OpenEdge instance runs certain configuration procedures when a client runs external or internal procedures. PAS for OpenEdge event procedures are ABL procedures that you specify when you configure a PAS for OpenEdge instance; the procedures run typically before and after invoking remote procedures during the lifetime of a PAS for OpenEdge instance and its sessions.

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 a PAS for OpenEdge instance running in session-managed and unbound application model.

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 PAS for OpenEdge instance, 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 PAS for OpenEdge instance's `SESSION` system handle. For more information on this attribute, see the Manage client context for session-free and unbound session-managed connections on page 46.

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 file name in the client's `RUN` statement, the property captures only the procedure file name.

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

Table 3: RUN requests and ProcedureName property values

RUN request on the ABL client	Procedure path or file name	Description
`RUN test/foo.p ON SERVER hdl.`	`test/foo.p`	The procedure name includes the relative path. The test directory is in the OpenEdge working directory on the PAS for OpenEdge instance.
`RUN C:/OpenEdge/WRK/test/foo.p ON SERVER hdl`	`C:/OpenEdge/WRK/test/foo.p`	The procedure name includes the absolute path.
`RUN bar.p ON SERVER hdl`	`bar.p`	No path is specified in the procedure name: `bar.p` is in the OpenEdge working directory on the PAS for OpenEdge instance.

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 PAS for OpenEdge

You can access the path or file name of a remote procedure running on a PAS for OE instance 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 PAS for OpenEdge instance 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 -- has an internal procedure others want to find. */
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 PAS for OpenEdge instance 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(" -URL http://localhost:3090/apsv","","") 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.
```

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 PAS for OE instance, 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"`
- `"h-UsefulProc.p&UsefulRoutine2"`

Using PAS for OpenEdge event procedures

PAS for OpenEdge event procedures are ABL procedures that you can specify when you configure an PAS for OpenEdge instance, and that run at specific times during the lifetime of a PAS for OpenEdge instance and its sessions. The three procedure types are described in the following sections.

- Startup and Shutdown procedures on page 35
- Connect and Disconnect procedures on page 37
- Activate and Deactivate procedures on page 39

Each of these procedures is run at specific times during the lifetime of a server session with the idea of simplifying the process of deploying an application with the PAS for OpenEdge instance.

Startup and Shutdown procedures

Startup and Shutdown procedures encapsulate logic that executes during the startup and shutdown of a PAS for OpenEdge multi-session agent and/or a PAS for OpenEdge ABL session. In effect, the logic in these procedures executes for each multi-session agent and in each ABL session created for an agent as it starts up and shuts down. Functionality that might go in a Startup or Shutdown procedure includes managing self-service database connections, loading or storing the contents of temp-tables, and instantiating or deleting certain local persistent procedures.

Note that on PAS for OpenEdge, all connections to a self-service database are created as single connections owned by each multi-session agent and shared by the ABL sessions managed by the agent. Each ABL session started by a multi-session agent that connects to the same database, whether by using the `-db` startup parameter or by executing the ABL `CONNECT` statement, becomes a separate user of the shared connection already established and owned by the multi-session agent. On the other hand, a network database connection started in a PAS for OpenEdge session is a separate user connection to the database owned by that session.

So, you can use Multi-session Agent Startup and Shutdown procedures to connect and disconnect all self-service databases used by its ABL sessions, and use Session Startup and use Shutdown procedures to connect and disconnect all network databases used by these sessions. At the same time, any self-service database that is connected in a Session Startup procedure shares that connection with self-service connection already established by the Agent Startup procedure, and if the Session Shutdown procedure disconnects that database, it has no affect on the self-service connection established by the Agent Startup procedure.

Multi-session Agent Startup procedure

PAS for OpenEdge uses a multi-session agent, which means that a single agent can concurrently handle requests from several clients, each in its own session, and each running its own application model. When a multi-session agent starts up, it runs any Agent Startup procedure that you have defined for the PAS for OpenEdge instance. This procedure can be used to perform tasks required by all server sessions that run in the PAS for OpenEdge instance. One common task is to create all self-service database connections that are shared by the server sessions created and managed by a given multi-session agent. Each multi-session agent is a user with its own self-service connection to a given database, and all of its server sessions share that same connection as separate users.

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

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

The session that executes an Agent Startup procedure runs effectively session-free. Any global data that is set in this session expires when the session terminates and is not available to other sessions of the multi-session agent. You use the following properties in `openedge.properties` to set them: `agentStartupProc` and `agentStartupProcParam`.

Note: Although all the ABL sessions started by a single multi-session agent can share a database connection, they cannot share data or running procedures.

Multi-session Agent Shutdown procedure

When a multi-session agent shuts down, it runs any Multi-session Agent Shutdown procedure that you have defined for the PAS for OpenEdge instance. This procedure can be used to perform tasks required when shutting down the agent and the server sessions that it manages. One common task is to shut down all self-service database connections that were started in the Multi-session Agent Startup procedure.

The session that executes an Agent Shutdown procedure runs effectively session-free. Global data that is set in this session expires when the session terminates and is not available to other sessions of the multi-session agent. You use the following property in `openedge.properties` to set them: `agentShutdownProc`.

Session Startup procedure

A Session Startup procedure executes as an server session starts up. You can specify the name of an PAS for OpenEdge Session Startup procedure using the OpenEdge Explorer or OpenEdge Management or by setting the `sessionStartupProc` property in the PAS for OpenEdge properties file (`openedge.properties`).

This 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 `sessionStartupProcParam` 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 PAS for OpenEdge session executes the Startup procedure.

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

If the Session Startup procedure completes with no error, the PAS for OpenEdge starts up successfully. If the Startup procedure completes with an error return value (`RETURN ERROR` statement), PAS for OpenEdge instance startup fails and the PAS for OpenEdge session is terminated.

Session Shutdown procedure

A Session Shutdown procedure executes just before the server session is shut down. A Session Shutdown procedure takes no parameters. You can specify the name of a PAS for OpenEdge Session Shutdown procedure using the OpenEdge Explorer or OpenEdge Management or by setting the `sessionShutdownProc` property in the PAS for OpenEdge properties file (`openedge.properties`).

Unlike the Session Startup procedure, the Session Shutdown procedure is run as a non-persistent procedure, and any errors propagated by the Shutdown procedure are simply ignored. The PAS for OpenEdge session terminates immediately after the Shutdown procedure executes.

Usage requirements

Both the Startup and Shutdown event procedures are optional. There is no requirement to specify either or both of them for multi-session agents or their ABL sessions. Additionally, the content of these procedures is completely application specific, and the PAS for OpenEdge architecture places no constraints on it.

Any ABL logic that is valid within a PAS for OpenEdge session that services a client request is valid for these event procedures.

Connect and Disconnect procedures

Connect and Disconnect procedures encapsulate logic that executes whenever a client application establishes and terminates a session-managed connection with a PAS for OE instance.

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, creating a bound connection between a client and the current PAS for OpenEdge session, or any other common actions required when each individual client connection begins or terminates for the PAS for OE instance.

Connect procedure

A Connect procedure executes in whatever PAS for OpenEdge session 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 `RETURN` *`string`* statement or the `RETURN ERROR` *`string`* 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:

- For ABL clients, see Connecting to a PAS for OpenEdge instance on page 71
- For SOAP Web Service clients, see *OpenEdge Development: Web Services*
- For .NET Open Clients, see *OpenEdge Development: .NET Open Clients*
- For Java Open Clients, see *OpenEdge Development: Java Open Clients*

You can specify the name of an PAS for OpenEdge Connect procedure by setting the `sessionConnectProc` property in the PAS for OpenEdge properties file (`openedge.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 a Web Service or Open Client executes a corresponding connection method.

You must define the procedure with three character input parameters. The following is an example of how you can define these input parameters:

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

When the PAS for OpenEdge 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 PAS for OpenEdge Connect procedure receives the Unknown value (?) for that parameter.

OpenEdge does not examine the values passed from the CONNECT( ) method to an PAS for OpenEdge Connect procedure. The name of each character input parameter might suggest the content you define for it. For example, cUserId can contain user name information, cPassword can contain password information, and cConnectInfo can contain other user-defined information needed by the PAS for OpenEdge 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 PAS for OpenEdge Connect procedure. After examining the values provided to the PAS for OpenEdge Connect procedure, you can reject the connection request from the client application by terminating the procedure with the RETURN ERROR statement.

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 typically begin by checking the cAppServerInfo procedure parameter to identify if this is to be a bound or unbound connection and set the SERVER-CONNECTION-BOUND-REQUEST attribute on the SESSION handle appropriately. You might then initialize context for the connection using an application-specific context database or using the SERVER-CONNECTION-CONTEXT attribute on the SESSION handle (see Manage context for bound and unbound session-managed connections on page 43). 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 marshal it into the SERVER-CONNECTION-CONTEXT attribute. Any PAS for OpenEdge procedure that executes on behalf of the connection can restore and re-save this context as appropriate. .

Disconnect procedure

A Disconnect procedure executes in whatever PAS for OpenEdge session handles a client disconnection request, but before the connection is terminated. You can use the PAS for OE 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 PAS for OpenEdge instances that were connected on behalf of the client.

You can specify the name of the PAS for OpenEdge Disconnect procedure by setting the sessionDisconnProc property in the PAS for OpenEdge properties file (openedge.properties). This procedure takes no parameters.

For a bound client connection, the Disconnect procedure runs after the PAS for OE instance returns the response to the disconnect request to the client application. For an unbound client connection, the procedure runs just before the PAS for OpenEdge returns this disconnect 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

Any Connect or Disconnect procedures that you specify run only for session-managed client connections

If the client is bound in a session-managed connection, the Connect procedure runs as a persistent procedure. If the client is unbound in a session-managed connection, the Connect procedure runs as a non-persistent procedure. The Disconnect procedure always runs as a non-persistent procedure.

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 client is bound in a session-managed connection, the persistent Connect procedure is automatically deleted after the client disconnects.

Note that if the client is unbound in a session-managed connection, there is no guarantee that the PAS for OpenEdge session that runs the Connect procedure for the client application will be the same one that actually processes each request for the client application. Thus, any context that a Connect procedure establishes for an unbound connection must be stored in a file, database, SERVER-CONNECTION-CONTEXT attribute of the SESSION handle (see Manage context for bound and unbound session-managed connections on page 43), 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 unless you need the client to be bound to the same PAS for OpenEdge session for the entire life of a session-managed connection.. Additionally, the content of these procedures is completely application specific, and the application server architecture places no constraints on it. Any ABL logic that is valid within an a PAS for OpenEdge session that services a client request is valid for these procedures.

Activate and Deactivate procedures

Activate and Deactivate procedures encapsulate logic that executes automatically for client requests that run over session-free or unbound session-managed connections.When a client application sends remote procedure requests over these types of connections, each request might execute in a different PAS for OpenEdge session instance. To maintain application continuity between sessions running over the same client connection, you might need to establish some application-specific resources or context before, and discard them after each request. The Activate and Deactivate procedures help to manage these resources and context more easily.

Note that if your application uses a mixed application model, you might need to manage the resources and context for each request differently, depending on the application model of the current client request. You can verify the application model of the current client request by testing the value of the SERVER-OPERATING-MODE attribute on the SESSION handle and execute the appropriate context management code accordingly.

Activate procedure

The Activate procedure executes immediately before a remote procedure request when the connection is session-free or session-managed in the unbound state. A typical use of Activate procedures for unbound session-managed connections is to retrieve connection context using an application-specific context database or the SERVER-CONNECTION-CONTEXT attribute on the SESSION handle (see Manage context for bound and unbound session-managed connections on page 43). 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 unmarshal 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 procedure on page 37.

For either session-free or unbound session-managed connections, 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 PAS for OE instance) 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 Manage client context for session-free and unbound session-managed connections on page 46.

You can specify the name of an PAS for OpenEdge instance Activate procedure by setting the sessionActivateProc property in the PAS for OE properties file (openedge.properties). If you specify this procedure, it accepts no parameters and runs as a non-persistent procedure in any PAS for OpenEdge session that executes a remote procedure request for an session-free or unbound session-managed connection.

Note: The Activate procedure only runs prior to remote procedure requests. It does not run in response to connect or disconnect 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 session-free or session-managed in the unbound state. A typical use of Deactivate procedures for unbound session-managed connections is to store connection context to be saved from the session that just completed the current request using an application-specific context database or using the SERVER-CONNECTION-CONTEXT attribute on the SESSION handle. For more information, see Manage context for bound and unbound session-managed connections on page 43. 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 its just completed request and store it in the context database or marshal it into the SERVER-CONNECTION-CONTEXT attribute.

For either session-free or unbound session-managed connections, 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 PAS for OE instance) 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 Manage client context for session-free and unbound session-managed connections on page 46.

You can specify the name of an PAS for OpenEdge Deactivate procedure by setting the sessionDeactivateProc property in the PAS for OpenEdge properties file (openedge.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 a session-managed connection is unbound when the request completes execution.

A session-managed 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 a session-managed 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 a session-managed 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 session-managed connections transition between bound and unbound states, see Manage context for bound and unbound session-managed connections on page 43.

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 session-free or session-managed connections in the unbound state. If a remote procedure request executes over a bound session-managed 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 PAS for OpenEdge architecture places no constraints on it. Any ABL logic that is valid within a PAS for OpenEdge session that services a client request is valid for these procedures.

Controlling PAS for OpenEdge entry points

You can control PAS for OpenEdge entry points at run time using the EXPORT( ) method on the SESSION handle. PAS for OpenEdge *entry points* are the pathnames of procedures in the PAS for OpenEdge 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 promote secure access to the business application running on the PAS for OpenEdge instance.

The scope of the export list set by the EXPORT( ) method is the PAS for OpenEdge session in which the method executes. As result, when and where you need to set the export list depends both on your application model 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 PAS for OpenEdge PROPATH as a remote procedure.

This section describes how the PAS for OpenEdge uses an export list to limit client access to the PAS for OpenEdge and how you can use the EXPORT( ) method to set them. For more information on how the application model can affect export list settings, see the information on PAS for OpenEdge security in Design and Implementation Considerations on page 107.

Export list operation

When a PAS for OpenEdge session 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 a PAS for OpenEdge 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 application model of your connection. For more information on setting export lists for different application models, see Design and Implementation Considerations on page 107.

Calling the EXPORT( ) method

There is no requirement to execute the EXPORT( ) method within a server 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 a PAS for OpenEdge Connect procedure. In this example, two procedures, `order.p` and `stock.p`, are added to the export list, as shown:

```
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 a server session that executes this PAS for OpenEdge 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:

```
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 server session 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:

```
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:

```
lReturn = SESSION:EXPORT(). /* Reset to empty */
lReturn = SESSION:EXPORT("special-order.p,special-stock.p").
```

At this point, the PAS for OpenEdge session 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 server session 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 working directory in which the PAS for OpenEdge session starts might affect how a procedure name is resolved because remote procedure requests and the PROPATH setting might contain unqualified (relative) pathnames.
- A PAS for OpenEdge session runs in the same directory as the multi-session agent that starts it. Therefore, carefully consider what directory the multi-session agent starts in because procedure names might be resolved relative to this directory.

For more information on the PROPATH setting for a PAS for OpenEdge instance, see *OpenEdge Application Server: Administration*. For more information on the standard name resolution rules, see the RUN statement entry in *OpenEdge Development: ABL Reference*.

Manage context for bound and unbound session-managed connections

On a bound session-managed connection, all remote procedures run in the session of a PAS for OpenEdge instance. The context at the initiation of a client request is exactly as it was when the previous request from the same client completed.

Unbound session-managed connections generally require more complex context management than for bound connections. The trade-off is potentially higher throughput for the effort. Because any ABL session in a PAS for OpenEdge session pool can handle any remote procedure request on any unbound connection, the context created by one remote procedure request is generally unavailable to the next request in the same connection sequence. To help you manage context for an unbound session-managed connection, OpenEdge provides the following ABL elements:

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

Other than in Use the SERVER-CONNECTION-BOUND attribute on page 45, most of the remaining information on context management applies only to session-managed connections in the unbound state.

To support context management for either session-free or unbound session-managed connections, OpenEdge also provides the `ClientContextId` property on the `Progress.Lang.OERequestInfo` class. For more information, see Manage client context for session-free and unbound session-managed connections on page 46.

Use the SERVER-CONNECTION-CONTEXT attribute

When a client application establishes a session-managed connection with a PAS for OE instance, the instance session manager creates a connection context area in memory for the connection. A PAS for OpenEdge procedure can access this context area using the `SERVER-CONNECTION-CONTEXT` attribute of the `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 in a session over a session-managed connection, it can set the `SERVER-CONNECTION-CONTEXT` attribute to any application-specific value. Each time OpenEdge assigns a request from a session-managed client to a PAS for OpenEdge session, OpenEdge ensures that the `SERVER-CONNECTION-CONTEXT` attribute for that session is set to the last value assigned in a previous server session 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 PAS for OpenEdge instance. Thus, each ABL session that services a an unbound session-managed client connection can pass context information to the next.

Use the SERVER-CONNECTION-ID attribute

For more complex 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 unbound session-managed connection. A context database is an application database that is connected to and accessible from each that services a session-managed client..

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 session-managed request is executed within a client connection, 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 PAS for OpenEdge instance.

Use the SERVER-CONNECTION-BOUND-REQUEST

A session-managed client connection to a PAS for OpenEdge instance is in one of two states:

- **Unbound** — Where remote procedure requests over the connection can be handled by any available PAS for OpenEdge session
- **Bound** — Where remote procedure requests over the connection are handled by a single PAS for OpenEdge session that is dedicated to the connection

By default, a session-managed connection is unbound.

Bound connections

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 63) or instantiate an Open Client ProcObject, in the case of an Open Client or SOAP Web service client. This call runs and instantiates a persistent procedure in the PAS for OpenEdge session that handles the remote procedure request. The connection thus becomes bound, dedicating this session to handle all future client requests over the bound connection.

  Note: 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.

- A PAS for OpenEdge session 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 session that sets this attribute to handle all future client requests over the bound connection.

Unbound connections

You can request that a bound 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 PAS for OpenEdge session.

Once a connection transitions from the bound to the unbound state, the PAS for OpenEdge session can handle the next request sent by any connected client, whether it is session-managed or session-free. Conversely, the next request sent by the client on the previously bound connection can be handled by any available PAS for OpenEdge session.

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 bound session 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.

Use the SERVER-CONNECTION-BOUND attribute

A session-managed 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 a session running in a session-managed connection, 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`.

Manage client context for session-free and unbound session-managed connections

In the session-free or the (especially unbound) session-managed application models of the PAS for OpenEdge instance, any available PAS for OpenEdge session handles each remote request from a PAS for OpenEdge client. The session 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 (acting as a client) the session can make its own request to another PAS for OE instance, passing the same client context identifier on behalf of the original client.. Sessions across one or more PAS for OpenEdge instances can thus use this same client context identifier to access the context store on behalf of the original client request..

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 a PAS for OpenEdge instance. The client can then assign its own value for this identifier if necessary, possibly one that has been previously initialized for a different PAS for OpenEdge connection.

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

Pass 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 both on the server object handle that an ABL client uses to make remote requests, and on the `SESSION` system handle of the server session 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 server 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 server associated with this handle
- On the server session's `SESSION` handle:
  - `CURRENT-REQUEST-INFO` — Referencing context information that is received from the current client request to the server. 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 server 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 server. 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 client that sends the request or the server that returns the response, depending on the handle and 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 server, it again propagates the client context identifier (and related information) from the REQUEST-INFO attribute on the server handle, and the server session 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 ABL Reference*.

For more information on using the ProcedureName property, see 'Accessing the name of the current remote procedure' in *Develop ABL Services*.

Generate 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:

```
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 clients (ABL, Java, and .NET) to automatically generate the value and send it to the server. Optionally, the client or the server session 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 on the same server 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 49 and the topics on authenticating the user identity for multi-tier applications in *OpenEdge Programming Interfaces*.

Manage context from the server with OERequestInfo objects

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

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

OpenEdge context information accessible by a server session provides:

- **OpenEdge version information for the current server session** — The server 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.
- **Server version information returned to a client** — This identifies the OpenEdge Release information for the server returned to the client from the server session in response to the current client request. This is the same information provided to the agent using its own `LOCAL-VERSION-INFO` attribute. The server session provides this information to the client using the `VersionInfo` property on the `OERequestInfo` instance referenced by the `CURRENT-RESPONSE-INFO` attribute on the session'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 session 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 server session 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 server session 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 server session handling the request 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 session-free connections, the typical way for both the client and server to manage context is for both the client and server to manage context. A client request can be executed, not only on any available session on the server, but on any one of many servers that support the business application for a single user login session. You can also use this same configuration for one or more servers that support a single user login session for an unbound session-managed application..

Because none of the context management attributes provided for a session-managed application work for a session-free application (except for the SERVER-CONNECTION-BOUND attribute), the context database must:

- Be shared by all server instances that support the user login session
- Use an entirely application-defined means (such as the ClientContextId property) to identify and distinguish user login session 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 each session handling the remote requests.

For more information on managing context from the ABL client, see Accessing client context regardless of application model on page 78.

Implementing multi-tier security models

A Server session can act as a client of another server 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 Manage context from the server with OERequestInfo objects on page 48), this "server session as client" can also execute remote procedure calls using the following security models:

- **Delegation** — The server session establishes its application context in a remote login procedure by propagating the client context identifier from the originating client to the authentication server.
- **Impersonation** — The server session 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 server, thus allowing the other server to establish the proper client context for the call.
- **Single sign-on (SSO)** — The server session sets the client context identifier of a server handle for a Server B based on the context established by executing the remote login procedure on the server handle for a Server A (the authentication server). When the initial server session connects to Server B, the server'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 Server A.
- **WebSpeed® session** — This replaces the server session along with a Web browser serving as the originating client and the WebSpeed application possibly calling out to a separate authentication server. 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 a PAS for OpenEdge 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 server session where the procedure runs. 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 server session 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 107.

Deleting remote procedures

As with local persistent procedures, a remote persistent procedure context remains active within a server session (binding that session to the client) until it is deleted using the DELETE OBJECT or DELETE PROCEDURE statement. You can thus delete a remote persistent procedure in the server 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 server session 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 server session, and a singleton procedure can be running in more than one server session, where it can be deleted using ABL code running in the session.

Handling conditions and returning values

Conditions raised in a server 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 a server session to an ABL client. The following sections describe how each of these standard ABL mechanisms can affect the server 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 session running a session-free or unbound session-managed request 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 remote user-defined function.

Throwing error objects to the client

Error objects can be thrown from a PAS for OpenEdge instance 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 server instance and then reconstructed on the client, and they are subject to the same rules for serialization as all other class-based objects passed between a server 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 a server and ABL client on page 84 for more information.

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

Raising the ENDKEY condition

Like an ABL client, a server 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 in a connected PAS for OpenEdge instance, OpenEdge also raises the STOP condition in the context of the server session that is 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 server 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 53.

A STOP condition raised in an Activate procedure for a session running a session-free or unbound session-managed request 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 ABL session 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 a PAS for OpenEdge instance, the time limit for the procedure is enforced in the client session. If the timeout is exceeded, the STOP condition is raised by the client. In this context, a STOP message is sent to the server session running the remote procedure after the specified time limit has 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 a PAS for OpenEdge instance:

```
/* 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 http://localhost:3090/apsv").
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 a PAS for OpenEdge instance running on the local host, 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 session begins timing this when the remote procedure call is made. If the call to srvrNested3.p does not return in the client session within the 5 second time limit, the client session will issue a STOP message to the server session running srvrNested3.p. When the PAS for OpenEdge instance receives the STOP message, the STOP condition will be raised in the server session, as shown:

```
/* 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 PROCEDURE.
```

In `srvrNested3.p` running in the server session, 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 session 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 session. The timeout is consequently propagated back to the client session.

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.

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 53.

A `QUIT` condition raised in an Activate procedure for a session running a session-free or unbound session-managed request 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 a server session has no effect on the client application for which it is handling a request. 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` , respectively, in a PAS for OE session, have definite effects on any client application for which the server session is handling a request.

Any unhandled STOP causes the remote procedure request (persistent or non-persistent) to terminate and propagate the STOP condition to the client application.

Any unhandled 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 server.

Managing transactions

You can manage transactions in a PAS for OpenEdge 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 PAS for OpenEdge sessions are entirely separate, including their transaction contexts, the client normally has no direct ability to influence the course of a server transaction that begins and ends within the scope of a single remote procedure request. To allow finer-grained control of transactions running on PAS for OpenEdge, OpenEdge provides an additional transaction type that allows a transaction on a PAS for OpenEdge instance to span multiple client requests.

Types of a PAS for OpenEdge instance transactions

A PAS for OpenEdge server can allow a client to invoke two different types of transactions in a PAS for OpenEdge instance. 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 server session 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. For more information on managing normal ABL transactions on a PAS for OpenEdge instance, see the Implementing normal ABL transactions on page 55.

Automatic transactions

An *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 PAS for OpenEdge session. To control this type of transaction, the OpenEdge provides attributes and methods that you can use to specify how and when you want the transaction to end in the PAS for OpenEdge session.

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 client 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 server 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 server, 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 55.

Implementing normal ABL transactions

You can implement a normal ABL transaction in any PAS for OpenEdge procedure that the client runs either directly or indirectly. The ABL for starting a normal ABL transaction in a server 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 server 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 for a client connected in any application model (session-managed or session-free).

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 PAS for OpenEdge session 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 server 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. Also, because this type of transaction must be initiated using a persistent procedure, you can implement an automatic transaction most appropriately for a client connected in the session-managed application model, which is then bound to the server session at least until the persistent procedure is deleted.

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 server session and can remain open after the remote procedure call returns to the client. The context for this open automatic transaction is the entire server session, not (as you might think) the context internal to the transaction initiating procedure instantiated in the server session. Thus, any remote procedure (persistent or non-persistent) that is subsequently executed in the same server session 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 handle of any procedure executing in the server 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

Method/Attribute	Return/Data type	Description
DEFAULT-COMMIT	LOGICAL	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.
IS-OPEN	LOGICAL	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.
SET-COMMIT( )	LOGICAL	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.

Method/Attribute	Return/Data type	Description
SET-ROLLBACK( )	LOGICAL	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.
TRANS-INIT-PROCEDURE	HANDLE	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.)

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 in the server session 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 server session. However, if no transaction is open, any such attempt to call these internal procedures from within the server session 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 58 for more information).

  Note: You can determine whether an automatic transaction is open in a server 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 server session by deleting the proxy procedure handle (on an ABL client) or the remote persistent procedure handle (on the application server) 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 server 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 server 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 a PAS for OE instance 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 107.

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

```
/* 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 server. 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 server session automatically starts a new transaction with no need for the client to call a procedure (`startme`) to initiate it.

a-txclnt.p

```
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 63.

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 PAS for OpenEdge procedures for Open Client (Java, .NET, and SOAP Web service client) applications is not much different than programming for ABL clients. However, for each remote 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 SOAP Web service interface for use by Open Client and SOAP 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 ABL editor in Progress Developer Studio for OpenEdge

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 of the ADE, especially if you build them using SmartObjects™.

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. The **Prototype Generator** dialog box opens.
3. In the **Prototype Generator** dialog box, enter the file name 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 the manual, *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.

3

Programming ABL Client Applications

This chapter describes how to program ABL procedures that execute in an ABL session that runs as a client of a PAS for OpenEdge instance.

For details, see the following topics:

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

Programming for the application model

If you are writing a client for a new PAS for OpenEdge business application, the decision of what application model to use most immediately impacts the server and its configuration and programming. Once this decision is made for a new business application or if you are writing the client for an existing application, the application 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 business application 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.

Session-managed applications provide various means to manage context depending on the binding set up between clients and PAS for OpenEdge sessions. Where there are any responsibilities involved for the ABL client, this chapter describes them as appropriate for the client binding.

Otherwise, this chapter describes the differences in functionality available explicitly for each application model, as appropriate. For more information on the differences between the two application models, session-free and session-managed, see PAS for OpenEdge and Client Interaction on page 11.

Note: This chapter addresses programming issues specific to ABL client procedures of a PAS for OpenEdge instance. It assumes that you are generally familiar with writing and executing ABL procedures. For information on using ABL, see *OpenEdge Getting Started: Guide for New Developers*. While there is some reference to Open Clients, for complete information on programming Java, .NET, and SOAP Web service Open Clients, see *OpenEdge Development: Open Client Introduction and Programming*.

ABL for programming PAS for OpenEdge client procedures

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

Table 5: ABL for programming PAS for OpenEdge client procedures

ABL element	Description
`ASYNC-REQUEST-COUNT`	One of the following: 1) An `INTEGER` attribute on the server object handle that returns the number of active asynchronous requests submitted to this server. 2) An `INTEGER` attribute on the procedure handle that returns the number of currently outstanding asynchronous requests for this procedure. Can be non-zero only if the `PROXY` and `PERSISTENT` attributes are both set to `TRUE`.

ABL element	Description
Asynchronous request object handle	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.
`CANCELLED`	A `LOGICAL` attribute on the asynchronous request object handle that indicates if the asynchronous request was canceled using either the `CANCEL-REQUESTS( )` method or the `DISCONNECT( )` method on the associated server handle.
`CANCEL-REQUESTS( )`	A method on the server object handle that for a session-managed application raises a `STOP` condition in the context of the currently running asynchronous request and purges the send queue of any asynchronous requests that have not been executed. For a session-free application, it causes a `STOP` condition to be raised for all currently running asynchronous requests, and purges the send queue of any asynchronous requests that have not been executed.
`CLIENT-CONNECTION-ID`	A `CHARACTER` attribute on the server object handle that for a session-managed application returns the connection ID for the PAS for OpenEdge connection associated with this server handle.
`COMPLETE`	A `LOGICAL` attribute on the asynchronous request object handle that indicates if the asynchronous request is completed and its result is processed on the client.
`CONNECT( [`*`connection-parameters`*`] [ , userid] [ , password] [ , appserver-info] )`	A method on the server object handle that connects and associates a PAS for OpenEdge instance with the server handle.
`CONNECTED( )`	A method on the server object handle that returns `TRUE` if a PAS for OpenEdge instance is currently connected and associated with the server handle. **Note:** The `CONNECTED( )` method relies on information from the operating system, which may not always be reliable. The method may return `TRUE` when a connection has been lost.
`CREATE SERVER `*`server-handle`*	A statement that creates a server object handle and stores it in a `HANDLE` variable.

ABL element	Description
DEFINE INPUT PARAMETER *parameter*	A statement for defining INPUT parameters for an event procedure or method. Each INPUT parameter must correspond to an OUTPUT or INPUT-OUTPUT parameter of the associated asynchronous remote procedure.
DELETE OBJECT *handle*	A statement that you can use to delete certain objects, including server objects, and persistent procedures (local and remote), and asynchronous request objects.
DELETE PROCEDURE *procedure-handle*	A statement that you can use to delete both local and remote procedure objects (persistent procedures).
DISCONNECT( )	A method on the server object handle that disconnects from and removes all reference to the PAS for OpenEdge instance currently associated with the server handle. Any running or pending asynchronous requests submitted by this client are also canceled.
ERROR	A LOGICAL attribute on the asynchronous request object handle that indicates that an ERROR condition was returned from the PAS for OpenEdge instance as a result of processing the request.
EVENT-HANDLER	A CHARACTER attribute on the asynchronous request object handle that contains the name of the internal procedure or method to be run as the event procedure for this asynchronous request.
EVENT-HANDLER-OBJECT	A Progress.Lang.object attribute on the asynchronous request object handle that contains the procedure handle of the event procedure context or the object name containing the event method defined as the asynchronous request callback handler.
EVENT-PROCEDURE	A CHARACTER 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.
EVENT-PROCEDURE-CONTEXT	A HANDLE 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.
FIRST-ASYNC-REQUEST( )	A method on the server object handle that returns the first entry in the list of all current asynchronous request handles for the specified PAS for OpenEdge instance.
FIRST-PROCEDURE	A HANDLE attribute on the server object handle that returns the first entry in the list of remote persistent procedures running on the connected PAS for OpenEdge instance.
FIRST-SERVER	A HANDLE attribute on the SESSION system handle that returns the handle to the first entry in the chain of server handles for the session.

ABL element	Description
FUNCTION ... IN *procedure-handle*	A statement that defines a forward reference to a remote user-defined function prototype. When *procedure-handle* is a proxy procedure handle, the function is defined on the connected PAS for OpenEdge instance in the specified procedure object.
LAST-ASYNC-REQUEST( )	A method on the server object handle that returns the last entry in the list of all current asynchronous request handles for the specified PAS for OpenEdge instance.
LAST-PROCEDURE	A HANDLE attribute on the server object handle that returns the last entry in the list of remote persistent procedures running on the connected PAS for OpenEdge instance.
LAST-SERVER	A HANDLE attribute on the SESSION system handle that returns the handle to the last entry in the chain of server handles for the client session.
LOCAL-VERSION-INFO	An object reference attribute on the SESSION system handle that references a Progress.Lang.OEVersionInfo object, which identifies the current instance of the OpenEdge client AVM, including its OpenEdge version.
NAME	A CHARACTER attribute on the server object handle that uniquely identifies the PAS for OpenEdge instance. It returns the connection ID for the PAS for OpenEdge instance associated with the server handle.
NEXT-SIBLING	One of the following: 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. 2) A HANDLE attribute on the proxy procedure handle that returns the next entry in the list of proxy procedure handles. 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 PAS for OpenEdge instance.
PERSISTENT	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.
PERSISTENT-PROCEDURE	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.

ABL element	Description
PREV-SIBLING	One of the following: **1.** A HANDLE attribute on the server object handle that returns the previous entry in the list of server handles created for the current ABL session. **2.** A HANDLE attribute on the proxy procedure handle that returns the previous entry in the list of proxy procedure handles. **3.** A HANDLE 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 PAS for OpenEdge instance.
PROCEDURE-COMPLETE	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 EVENT-PROCEDURE and EVENT-PROCEDURE-CONTEXT attributes.
PROCEDURE-NAME	A CHARACTER attribute on the asynchronous request object handle that provides the name of the remote procedure executed to instantiate this asynchronous request handle.
PROCESS EVENTS	A statement that you can use to handle any pending PROCEDURE-COMPLETE events for asynchronous requests. You can also use any blocking I/O statement, such as the WAIT-FOR statement.
PROXY	A LOGICAL attribute on a procedure handle that is TRUE if the procedure handle is a proxy handle for a persistent, single-run, or singleton procedure running remotely in the context of a PAS for OpenEdge instance.
Proxy procedure handle	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.
QUIT	A LOGICAL attribute on the asynchronous request object handle that indicates a QUIT condition was returned from the PAS for OpenEdge instance as a result of processing the request.
REQUEST-INFO	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 PAS for OpenEdge instance connected using the associated server object handle.

ABL element	Description
RESPONSE-INFO	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 PAS for OE instance connected using the associated server object handle.
RUN ... [PERSISTENT | SINGLE-RUN | SINGLETON...] ON [SERVER] *server-handle* [TRANSACTION DISTINCT] [ASYNCHRONOUS ...] ...	The statement that executes an external (remote) procedure on a connected PAS for OpenEdge instance 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 on a session-free or unbound session-managed connection, the procedure is instantiated on the server without causing the client to be bound to the server session. 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.
RUN ... IN *procedure-handle* [ASYNCHRONOUS ...] ...	The statement that executes an internal (remote) procedure on a connected server, where *procedure-handle* specifies a remote persistent, single-run, or singleton procedure that defines the internal procedure on the PAS for OpenEdge instance. 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 or method 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.
SELF	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.
SERVER	One of the following: **1.** A HANDLE attribute on a remote persistent procedure handle that returns the server object handle to the PAS for OpenEdge instance on which the specified remote persistent procedure runs. Valid only if the PROXY and PERSISTENT attributes are both TRUE. **2.** A HANDLE attribute on the asynchronous request object handle that returns the server handle of the PAS for OE instance 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 the SESSION handle.

ABL element	Description
Server object handle	A type of handle that provides a connection to a PAS for OE instance in an ABL client application. This handle provides server object methods and attributes that allow you to connect and manage the PAS for OpenEdge instance.
SINGLE-RUN	A LOGICAL attribute on procedure handles and the THIS-PROCEDURE system handle (in a server session) that is TRUE if the specified procedure is running as a single-run object.
SINGLETON	A LOGICAL attribute on procedure handles and the THIS-PROCEDURE system handle (in a server session) that is TRUE if the specified procedure is running as a singleton object.
STOP	A LOGICAL attribute on the asynchronous request object handle that indicates a STOP condition was returned from the PAS for OpenEdge instance as a result of processing the request.
TYPE	One of the following:1) A CHARACTER attribute on the server object handle that returns the handle type, which is "SERVER" for a server object handle. 2) A CHARACTER attribute on the asynchronous request object handle that provides the handle type, which is "ASYNC-REQUEST" for an asynchronous request object handle.
WAIT-FOR ...	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.

Accessing PAS for OpenEdge resources from a client

In order to run remote procedures on a PAS for OpenEdge instance, a client application must first establish a session-managed or session-free connection to the required instance. When done with the connection, the client application can then disconnect from the instance.

Steps for accessing PAS for OpenEdge resources

To allow your client application to access and use a PAS for OpenEdge instance:

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

4. When all remote requests to the PAS for OpenEdge instance have completed, use the `DISCONNECT( )` method on the server object handle to disconnect the client application from the instance.
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-PAS for OpenEdge interactions

Several features describe the relationship between a client application and the PAS for OpenEdge environment. For example:

- A client application can have connections to multiple PAS for OpenEdge instances simultaneously.
- A session-free client application can connect specifically to one OE ABL Web application running a given business application on a single PAS for OpenEdge instance or to multiple PAS for OpenEdge instances running the same business application using a DNS load balancing service.
- How the the session manager of a PAS for OpenEdge instance allocates server sessions to service client requests that it receives depends on the application model that the client uses to connect to the PAS for OpenEdge instance.

  Note: Just as you must write the business application according to the application model (or models) that it supports, you must accordingly to write the client application to access and manage responses from PAS for OpenEdge.

- A PAS for OpenEdge business application can manage transactions in a number of ways. The transaction management mechanism used by PAS for OpenEdge sessions is completely transparent to the client. The client has no direct control over the transaction mechanics used by the PAS for OpenEdge business application, but the business application implements a transaction model that works with the application model that it expects to run in.

Connecting to a PAS for OpenEdge instance

The connection mechanism for both session-managed and session-free connections is the same. For both application models you use the `CONNECT( )` method on a server object handle to create the appropriate connection to an OE ABL Web application running on a particular PAS for OpenEdge instance using an HTTP or HTTPS-based URL.

To connect an ABL client to an OE ABL Web application:

1. Create a server object handle.
2. Execute the `CONNECT( )` method on the server object handle, passing the parameters necessary to identify the URL to the Web application on a given PAS for OpenEdge instance, the application model to use, and additional information based on the URL scheme (HTTP or HTTPS) and the chosen application model.

Creating a server object handle

To create a server object handle,you must define a HANDLE variable and execute the CREATE SERVER statement to instantiate and reference the new server object with the handle. For example::

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

You can then use this handle to physically connect to a PAS for OpenEdge instance.

Establishing a connection with the CONNECT( ) method

The CONNECT( ) method takes up to four character string arguments in the following order, only one (*connection-parameters*) is required, depending on application model of the connection and the parameter requirements of any configured Connect procedure for the PAS for OpenEdge instance:

1. *connection-parameters*
2. *userid*
3. *password*
4. *appserver-info*

You use the *connection-parameters* argument to specify the location and connection configuration for a PAS for OpenEdge instance that supports a given business application. For more information, see Connection parameters argument on page 73.

The *userid*, *password*, and *appserver-info* arguments apply only to session-managed configurations and are passed from the client application to the Connect procedure (if configured) on the PAS for OpenEdge instance. Note that these parameters are never validated by the APSV transport or otherwise handled by the PAS for OE instance except to pass them to the Connect procedure. If a Connect procedure is not defined for the connected PAS for OpenEdge instance, or the connection is session-free, these arguments are discarded. The actual use and meaning of these parameters depends entirely on the Connect procedure and the associated business application running on the PAS for OE instance. For more information, see Session-managed application arguments on page 75.

A connection request can fail if:

- The server handle is invalid.
- One of the CONNECT( ) method parameters contains an invalid value.
- One of the values, such as the application model, specified in the *connection-parameters* parameter is invalid.
- There are not enough PAS for OpenEdge resources (for example, server sessions) available to service the connection.
- The Connect procedure fails.
- The Connect procedure terminates with a STOP condition, a QUIT condition, or after executing a RETURN ERROR statement.

A run-time error is generated when a connection is rejected by the server. If any error causes the connection request to fail, the CONNECT( ) method returns FALSE; otherwise it returns TRUE.

A successful connection lasts until the client application invokes the server handle `DISCONNECT( )` method or until the server session detects any failure conditions that automatically terminate the connection. For more information on disconnecting from a PAS for OpenEdge instance, see Disconnecting from a PAS for OpenEdge instance on page 90.

Connection parameters argument

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

- A set of optional connection parameters to specify information about the connection, such as the application model
- A single required connection parameter (`-URL`) to specify the Web path to the PAS for OpenEdge instance and the OE ABL Web application to which you are connecting

The following table describes these connection parameters to connect to a PAS for OE instance, regardless of the application model.

Table 6: PAS for OpenEdge connection parameters

Connection parameter	Description
`-NoSessionReuse`	(Optional) If specified, the connection does not reuse the TLS session ID when reconnecting over HTTPS to the same PAS for OpenEdge instance.
`-nohostverify`	(Optional) If specified, turns off host verification for an HTTPS connection. Without this parameter specified, the client compares the host name specified in the connection with the Common Name specified in the server certificate, and raises an error if they do not match. With this parameter specified, the client never raises the error. For more information, see *OpenEdge Getting Started: Core Business Services - Security and Auditing*.
`-pf` *`filename`*	(Optional) Where file name specifies a text file containing any of the other PAS for OE connection parameters described in this table. If this file contains any other OpenEdge startup parameters, the method ignores them.
`-sessionModel` *`option`*	(Optional) Where *`option`* specifies the application model in which PAS for OpenEdge sessions must run for this ABL client connection, and can have 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. (The default value is `Session-managed`.)

Connection parameter	Description
`-URL` *`web-path`*	(Required) Where *`web-path`* specifies the URL to connect the ABL client to a PAS for OpenEdge instance. This URL uses the APSV transport to connect to PAS for OpenEdge. For more information on the syntax of this URL, see Connection URL syntax for the APSV transport on page 74.
`-httpBasicUid`	Specifies the fully qualified name of the user to connect to a Web server that hosts PAS for OpenEdge or AIA web application.
`-httpBasicPwd`	Specifies the password for the user name.
`-clientConnectTimeout` *`nMilliseconds`*	(Optional) The number of milliseconds to wait before timing out each connection attempt.

Connection URL syntax for the APSV transport

Following is the URL syntax for the APSV transport, which OpenEdge supports to connect ABL clients or Java and .NET Open Clients to a PAS for OpenEdge instance:

Syntax

`scheme``://`[*`username`*`:`*`passphrase`*`@`]*`host`*`:`*`port`*`/`[*`web-app-name`*`/`]`apsv`

Where:

`scheme`

Specifies either `HTTP` or `HTTPS`.

Note: Internet-secure connections to a PAS for OpenEdge instance using HTTPS require the management of a certificate store for public key certificates on the client (TLS client) and a certificate and key store for private keys and certificates on the PAS for OE instance (TLS server). For information on configuring both OpenEdge clients and PAS for OpenEdge instances for HTTPS connections, see Digital certificate management on page 131 in this manual.

[*`username`*`:`*`passphrase`*`@`]

Specifies the user login credentials required to connect to the specified OE ABL Web application on the PAS for OpenEdge instance, according to the authentication model configured for the Web application. If the authentication model is Anonymous, no user credentials are required. If the model is HTTP Basic Authentication, you must provide a valid username (*`username`*) and password (*`passphrase`*) known to the user account systems configured for the Web application and its APSV transport.

Note: The APSV transport does not support HTTP Forms authentication.

host

Specifies the name or domain of the PAS for OpenEdge host.

port

Specifies the port for the host connection. Typically, HTTP uses 8810 and HTTPS uses 8811.

[*web-app-name/*]apsv

Specifies the OE ABL Web application and ABL client (or Open Client) transport (apsv) to connect with, where *web-app-name* is the name of an OE ABL Web application. If web-app-name is not specified, a connection to the default OpenEdge ABL web application (ROOT) is used.

For example, assuming a given a PAS for OpenEdge instance (*scheme*, *host*, and *port* (.)), an ABL client connection to an OE ABL Web application deployed in a WAR file named inventory.war is represented as:

```
./inventory/apsv
```

An ABL client connection to the default OE ABL Web application on the same host can be represented :

```
./apsv
```

Session-managed application arguments

Passing arguments for the *userid*, *password*, and *appserver-info* parameters is optional. However, the requirement to specify these arguments, and to provide specific values for any of them, depends on if and how the PAS for OpenEdge Connect procedure is being used on the server you are connecting.

Note: These arguments are ignored for a session-free application, because PAS for OpenEdge Connect procedures do not execute when a client connects with the session-free application model.

If a Connect procedure is defined for the PAS for OpenEdge instance, you must know how the Connect procedure uses its parameters to know how to specify the argument values. For example, the *userid* and *password* parameters can pass user credentials for a second level of authentication by the business application, or they can be used to pass any other information that the Connect procedure requires.

If you do not specify a value for an argument, OpenEdge passes the Unknown value (?) to the Connect procedure for that argument. OpenEdge only passes argument values to the Connect procedure from the CONNECT( ) method, and PAS for OpenEdge performs no evaluation of these arguments what so ever. It is the Connect procedure that actually evaluates the values that are passed.

Note: When determining what arguments to pass to the CONNECT( ) method, understand that a PAS for OpenEdge instance accepts a connection request only if any configured Connect procedure executes successfully.

If no Connect procedure is defined for the PAS for OpenEdge instance, you do not need to pass any arguments to these three parameters of the CONNECT( ) method. If you happen to provide them, they are ignored.

In the session-managed model, the PAS for OpenEdge Connect procedure has the ability to return a user-defined string if it uses the RETURN *string* or RETURN ERROR *string* statements. To access the string on the client, invoke the ABL RETURN-VALUE function immediately after invoking the CONNECT( ) method. An Unknown value (?) returned by this function indicates that the Connect procedure did not return a string.

For more information on Connect procedures, see Connect and Disconnect procedures on page 37.

Connection examples

The following code examples show how a client application specifies the connection and application arguments to connect a PAS for OpenEdge instance .

Connecting to a session-managed PAS for OpenEdge instance

In this session-managed connection example, the -URL parameter for the local PAS for OE instance passed as the *connection-parameters* argument to the CONNECT( ) method. Specific *userid* and *password* values are also passed as arguments to the Connect procedure configured for the instance. A value is not supplied as a *appserver-info* argument, which the procedure receives as the Unknown value (?). For example:

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

CREATE SERVER hAppSrv.
lReturn = hAppSrv:CONNECT
  ("-URL
http://slater:OuterLimits64@localhost:8810/inventory/apsv","SMITH","StarShip90").
IF NOT lReturn THEN DO:
  DELETE OBJECT hAppSrv NO-ERROR.
  RETURN ERROR "Failed to connect to the inventory ABL web application: " + RETURN-VALUE.
END.
. . .
lReturn = hAppSrv:DISCONNECT().
DELETE OBJECT hAppSrv NO-ERROR.
```

This code tries to connect to the `inventory` OE ABL Web application running on the local instance and authenticates the connection for user `slater` with the password, `OuterLimits64`.

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

Connecting to a session-free PAS for OpenEdge instance

In this session-free connection example, the *connection-parameters* argument to the CONNECT( ) method contains the -URL and -sessionModel parameters for a remote PAS for OpenEdge instance with the host name, ABLServices, and port, 8810:

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

CREATE SERVER hInventory.
lReturn = hInventory:CONNECT("-URL
http://slater:OuterLimits64@ABLServices:8810/inventory/apsv
  -sessionModel Session-free").
IF NOT lReturn THEN DO:
  DELETE OBJECT hInventory NO-ERROR.
  RETURN ERROR "Failed to connect to the inventory ABL web application".
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 OE ABL Web application running on the remote instance and authenticates the connection for user slater with the password OuterLimits64.

Accessing the connection ID on a session-managed client

On a session-managed ABL client, a connection ID for a PAS for OpenEdge 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 PAS for OpenEdge 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*).

Note: The CLIENT-CONNECTION-ID is provided for backward compatibility only. In order to maintain client connection context for an application regardless of model, use the ClientContextId property on the Progress.Lang.OERequestInfo class. For more information on using this property, see Accessing client context regardless of application model on page 78.

For a client, the connection ID of a given PAS for OpenEdge connection remains the same until the client disconnects from the PAS for OpenEdge instance. If the same client reconnects to the same instance, 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 the session-managed application model.

The CLIENT-CONNECTION-ID value is also shared with the SERVER-CONNECTION-ID attribute on the PAS for OE SESSION handle. This value can therefore be used by both the client and server to maintain a consistent client context for a session-managed connection. This value can also be useful to provide a common audit trail ID for any server and client running a session-managed application.

For information on how to use this value to manage connection context, see Manage context for bound and unbound session-managed connections on page 43. For information on how to use this value in audit trails, see Audit trails on page 130.

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 client connection to the server in a session-free application.

Accessing client context regardless of application 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 PAS for OpenEdge session'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 PAS for OpenEdge session, where it initializes the same properties in a separate instance of `OERequestInfo` that is referenced by the `CURRENT-REQUEST-INFO` attribute on the session's `SESSION` handle. When the server session 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 session 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 server 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 PAS for OpenEdge instance 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 servers 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 server, see Manage client context for session-free and unbound session-managed connections on page 46. 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.
- **PAS for OpenEdge version information on the current ABL client** — This identifies the OpenEdge Release information for a PAS for OpenEdge instance that handles a remote request for the client. The client can access the Release information for a connected PAS for OpenEdge instance 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 a PAS for OpenEdge session in order to propagate the originating client context identifier to yet another PAS for OpenEdge instance 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 PAS for OpenEdge response. The next PAS for OpenEdge session 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 server response** — The client can access the client context identifier returned by the server 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 server session code 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 PAS for OpenEdge session 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 PAS for OpenEdge 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 49.

Running and managing remote procedures

Once a client application connects to a PAS for OpenEdge instance, the client application can invoke requests to do the following:

- Run remote procedures as persistent, single-run, singleton, or non-persistent on the server 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 PAS for OpenEdge 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 in a server session. You can then access the results of the remote request in an event procedure or method 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 an ABL 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 hOrderON SERVER hAppSrv TRANSACTION DISTINCT.
```

  A synchronous remote persistent procedure executes similarly to the same persistent procedure run locally. The server 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 client connection or until the persistent procedure is explicitly deleted (see Deleting remote persistent procedures on page 86.) 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 server 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 85.

- Run a remote external procedure over a session-free or unbound session-managed connection, using the ON SERVER option and either SINGLE-RUN SET *handle-variable* (as shown below) or SINGLETON SET *handle-variable*

```
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 server session as it is when RUN PERSISTENT is used. Single-run and singleton operations offer the potential for greater server throughput (since the lack of binding increases the availability of sessions) and improved application performance (because the number of messages that the client must send to the PAS for OpenEdge instance 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 server, 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 client session sends the server 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 server access is being controlled by a PAS for OpenEdge export list, procedures run using the SINGLE-RUN and SINGLETON options should be added using the EXPORT( ) method. See Controlling PAS for OpenEdge entry points on page 41 for more information.

The difference between SINGLE-RUN and SINGLETON lies in how the client session 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 in a server session on the first call to an internal procedure or user-defined function, that instance remains in session memory until it is explicitly deleted or the session ends. This singleton instance is used for all subsequent calls into the remote procedure running in this server session, 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 raises 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 session when the request completes, and a singleton procedure can be running on more than one server session where it is separately instantiated, and where it can be deleted using ABL code running in the session.

- Run a synchronous internal procedure or user-defined function using the IN *handle-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 session-free and unbound session-managed connections, the same functionality is available by using the IN *handle-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 server session executing the request.

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 server 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 raises 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 93.

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

Passing class-based objects as parameters between a server and ABL client

Objects can be passed as INPUT, OUTPUT, and INPUT-OUTPUT parameters between a server and an ABL client. Because parameters are passed by value between a server 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 client session will raise a run-time error on the RUN statement. (Any logic associated with class members, like the GET and SET methods of a property, can differ on the client and server sides without raising an error, as long as their signatures 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 side 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 a server 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 server 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 server 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 server sessions. For information, see the sections on transaction management in Programming ABL Client Applications on page 63.

- Can create other objects in the server session, such as other persistent procedures. These objects are only available locally in the server session in which they are created. They cannot be directly shared between an ABL client application and the server 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 server 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 server session 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 107.

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 a server 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:

```
DEFINE VARIABLE hProxy  AS HANDLE NO-UNDO.
DEFINE VARIABLE hServer AS HANDLE NO-UNDO.

/* Create server handle for hServer and connect to a server. Call remote
persistent procedure on the server. */
hProxy = hServer:FIRST-PROCEDURE. /* First proxy handle */

/* Display the pathname of the file that contains the remote procedure active
on the server 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 server session 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 79.

Deleting remote persistent procedures

As with local persistent procedures, a remote persistent procedure context remains active within an server 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 PAS for OpenEdge instance where the procedure is active (see Disconnecting from a PAS for OpenEdge instance on page 90). 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 server by using the DISCONNECT( ) method on the corresponding server handle. This deletes all remote persistent procedures in the server session.

If the delete occurs in the context of another remote procedure request to the server, 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 server session 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 server session.

Remote procedure code examples

This section presents examples of ABL clients connected in the session-managed application model with a PAS for OpenEdge instance running on the remote host, zeus.

The first and fourth examples present 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 a PAS for OpenEdge instance 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(" -URL http://slater:OuterLimits64@zeus:8810/inventory/apsv",

                          "SMITH", "StarShip90").
IF NOT lReturn THEN DO:
  DELETE OBJECT hAppSrv NO-ERROR.
  RETURN ERROR "Failed to connect to server".
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 PAS for OpenEdge 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 PAS for OpenEdge instance:

```
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("-URL http://slater:OuterLimits64@zeus:8810/inventory/apsv",

                          "SMITH", "StarShip90").

IF NOT lReturn THEN DO:
  DELETE OBJECT hAppSrv NO-ERROR.
  RETURN ERROR "Failed to connect to server".
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 PAS for OpenEdge instance and session 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 a PAS for OpenEdge instance. 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:

```
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("-URL http://slater:OuterLimits64@zeus:8810/inventory/apsv",

                          "SMITH", "StartShip90").
IF NOT lReturn THEN DO:
  DELETE OBJECT hAppSrv NO-ERROR.
  RETURN ERROR "Failed to connect to server".
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 server. This automatic disconnection can also occur if the server machine looses 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`, that passes a class-based object parameter to the PAS for OpenEdge instance identified as `hAppSrv`. The object `rCustObj` is an instance of the class `myObjs.CustObj`, which is created for customer number 101, passed as numeric argument to the constructor. This class 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 rCustObj AS CLASS CustObj.
CREATE SERVER hAppSrv.
lReturn = hAppSrv:CONNECT("-URL http://slater:OuterLimits64@zeus:8810/inventory/apsv",

                          "SMITH", "StarShip90").
IF NOT lReturn THEN DO:
  DELETE OBJECT hAppSrv NO-ERROR.
  RETURN ERROR "Failed to connect to server".
END.
IF ERROR-STATUS:ERROR THEN DO:
  DELETE OBJECT hAppSrv NO-ERROR.
  RETURN ERROR RETURN-VALUE.
END.
. . .
rCustObj = NEW CustObj(101).
rCustObj:CustCredit = 0.
RUN updatecredit.p ON hAppSrv (INPUT-OUTPUT rCustObj).
rCustObj: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 `rCustObj:CustCredit` is initialized with 0. When the `RUN` statement is executed, a remote procedure request to run `updatecredit.p` is sent to the PAS for OpenEdge instance that is connected to 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 a PAS for OpenEdge instance

You can disconnect a PAS for OpenEdge 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 PAS for OpenEdge instance by sending a disconnect request to the connected server. When the server receives the disconnect request, the following occurs:

- **For a session-managed connection**— Any configured Disconnect procedure executes in any bound or otherwise available server session, then control returns to the client application.
- **For a session-free connection** — No Disconnect procedure executes and client control continues without interruption.
- **For any server with pending asynchronous requests** — All running or pending asynchronous requests are canceled and the corresponding event procedure or method associated with each request is called. The CANCELLED attribute on the asynchronous request handle for all such canceled requests is set to TRUE. For more information on asynchronous requests, see Managing asynchronous requests on page 93.

Disconnecting a session-managed client

The following example shows how a session-managed client can disconnect from a PAS for OpenEdge instance:

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

CREATE SERVER hAppSrv.
lReturn = hAppSrv:CONNECT(" -URL http://slater:OuterLimits64@zeus:8810/inventory/apsv",

                          "SMITH", "StarShip90").
IF NOT lReturn THEN DO:
  DELETE OBJECT hAppSrv NO-ERROR.
  RETURN ERROR "Failed to connect to Inventory on zeus".
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 PAS for OpenEdge instance referenced by the hAppSrv handle. When the disconnection request completes, you can send no more remote requests using the disconnected server handle.

Disconnecting a session-free client

The following example shows how a session-free client can disconnect from a PAS for OE instance:

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

CREATE SERVER hInventory.
lReturn = hInventory:CONNECT("-URL http://slater:OuterLimits64@zeus:8810/inventory/apsv

  -sessionModel Session-free").
IF NOT lReturn THEN DO:
  DELETE OBJECT hInventory NO-ERROR.
  RETURN ERROR "Failed to connect to Inventory on zeus".
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.
```

When the `DISCONNECT( )` method executes, the client application disconnects from the PAS for OpenEdge instance 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 PAS for OpenEdge instance. Once you have disconnected a server, you can release the associated server object handle by executing the `DELETE OBJECT` statement, as shown in the previous code examples.

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

Handling conditions and return values

Typically, you pass context across the process barrier between client and PAS for OE 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 server session where a `STOP` condition is then raised.

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 Progress Application Server for OpenEdge on page 25.

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 propagated back to the client in the context of the event procedure or method. The result can be referenced within the event procedure or method 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 or method itself are handled in the context of the PROCESS EVENTS or blocking I/O statement that executes the event procedure or method. For more information on handling conditions for asynchronous remote procedures, see Handling the response from an asynchronous request on page 94.

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 remote user-defined function.

Managing asynchronous requests

Asynchronous requests allow an ABL client to execute a remote procedure and continue execution before a server session 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 server 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 server 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 session-managed 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 PAS for OpenEdge and Client Interaction on page 11.

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 or method to use as the event procedure or method, to handle the response from the asynchronous request. For more information on the RUN...ASYNCHRONOUS statement, see the RUN statement entry in *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 or method, which returns the values as input parameters to the event procedure or method. Thus, the output parameters on the RUN...ASYNCHRONOUS statement serve only as placeholders to map the signature of the remote procedure on the PAS for OpenEdge instance. For more information, see Handling the response from an asynchronous request on page 94.

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 SET *async-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 server.
- Accessing the SELF system handle within an event procedure or method specified as a callback in response to the asynchronous request complete event.

For more information on using the asynchronous request handle, see Handling the response from an asynchronous request on page 94.

Handling the response from an asynchronous request

When an asynchronous request completes execution on the server, it sends a response to the 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 or method can be executed. ABL executes an ABL callback procedure or the event procedure or method in the context of a blocking I/O statement much like it executes the trigger for an ABL 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 or callback procedure. You also do not have to define a trigger for the event (for example, using an ON statement) because you specify an event procedure or method in the RUN...ASYNCHRONOUS 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 server, and stores any error information returned from the server 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 PAS for OpenEdge instance
- Attempts to execute the event procedure or method specified by the EVENT-HANDLER and the EVENT-HANDLER-CONTEXT attributes for the asynchronous request handle, if EVENT-HANDLER is not the empty string (""), and:
  - Sets each INPUT parameter for the event procedure or method 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 or method 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 or method fails to execute for any reason

Event procedures or methods

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

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

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

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

Note: You can overload the callback method and the AVM will identify the most appropriate method to run by using the the signature of the method. The signature of the method is determined based on the output and input-output parameters passed to the procedure executed by the RUN statement.

You must define the specified internal procedure or method 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 or method, 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 or method
- 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 client session 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 INPUT parameters do not match in order and type with the output parameters from the remote request.

Obtaining parameter values

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

Note that INPUT parameters in an event procedure or method observe the same rules of definition and scoping as any internal procedure or method, 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(-URL http://slater:OuterLimits64@zeus:8810/inventory/apsv).

RUN order.p ON SERVER hAppSrv ASYNCHRONOUS SET hAsync
  EVENT-HANDLER GetOrderRecords EVENT-HANDLER-CONTEXT 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 ttOrder temp-table (defined in the outer block) receives its content directly from the ttOrder INPUT parameter of the GetOrderRecords event procedure. However, you must explicitly assign the iCount variable (defined in the outer block) to the piOrderCount parameter in order to make the parameter value available to the outer block. (That is, if you define the INPUT parameter as iCount instead of piOrderCount, ABL does not automatically make the parameter value available as iCount in the outer block. Only the 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 order.p using the server session 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 or method that handles the asynchronous request complete event. 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 ("").

Canceling asynchronous requests

You can cancel all asynchronous requests that are executing or pending on behalf of the client on a particular server 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 or methods execute for all canceled requests the next time the client blocks for I/O or executes the PROCESS EVENTS statement. Each event procedure or method 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 or method has not yet run** — The event procedure or method 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.
- **Any request that is executing and that is stopped in response to the CANCEL-REQUESTS( ) method** — The event procedure or method input parameters are set to the 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 in the server session running the request.
- **Any request that is removed from the send queue prior to its execution** — The event procedure or method 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 canceled while waiting for execution.

Note: If you disconnect a server using the DISCONNECT( ) method, this also cancels all asynchronous requests still running or pending on the server for this particular client. For more information, see Disconnecting from a PAS for OpenEdge instance on page 90.

Canceling asynchronous requests after a specified time

The CANCEL-REQUESTS-AFTER( ) method allows you to 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 ensures that all requests currently running, or that are queued to run, for this client on the server 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 CANCEL-REQUESTS-AFTER( ) timer running, the timer is restarted with the new time interval, measured from the moment of method execution.

If the method is called with a parameter whose value is less than or equal to zero, then the timer on the 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 client example uses the CANCEL-REQUESTS-AFTER( ) method:

```
DEFINE VARIABLE hSrv AS HANDLE  NO-UNDO.
DEFINE VARIABLE bool AS LOGICAL NO-UNDO.

CREATE SERVER hSrv.
hSrv:CONNECT("-URL http://slater:OuterLimits64@zeus:8810/inventory/apsv -sessionModel
Session-free").

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 more complex example with both client and remote procedures that demonstrate use of both the CANCEL-REQUESTS-AFTER( ) method and the STOP AFTER phrase, which starts a timer on DO blocks on both the client and server. This is the remote procedure, srvrNested3.p:

```
/* 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 PROCEDURE.
```

This is the client procedure that asynchronously invokes the remote procedure (srvrNested3.p):

```
/* 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.

DEFINE FRAME foo
  S1 AT 1 WITH NO-LABELS.

DEFINE 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.

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 http://slater:OuterLimits64@zeus:8810/inventory/apsv -sessionModel
Session-free".
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-HANDLER "remoteProcHandler"
      EVENT-HANDLER-CONTEXT 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 server 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 session calls CANCEL-REQUESTS( ) method on the server handle. This will cause a STOP message to be sent to the server 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.

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 server. 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 client session 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 server session at the time of the delete request, the delete request is queued on the server until the server session 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 or method, using the SELF handle (that is, DELETE OBJECT SELF).

Mixing synchronous and asynchronous requests

If you attempt to execute a synchronous request on a server connection with outstanding asynchronous requests, The client session raises the ERROR condition. You can determine if there are any outstanding asynchronous requests for a server object 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 or method specified on the RUN...ASYNCHRONOUS statement as though it were responding to a PROCEDURE-COMPLETE event from a server.

Thus, using the SESSION handle, the event procedure or method 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 or method 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 a server 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 PAS for OpenEdge instance when a network PAS for OpenEdge instance is not available. For more information on running a local PAS for OpenEdge instance, see *Progress Application Server for OpenEdge: Administration Guide*. 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 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.

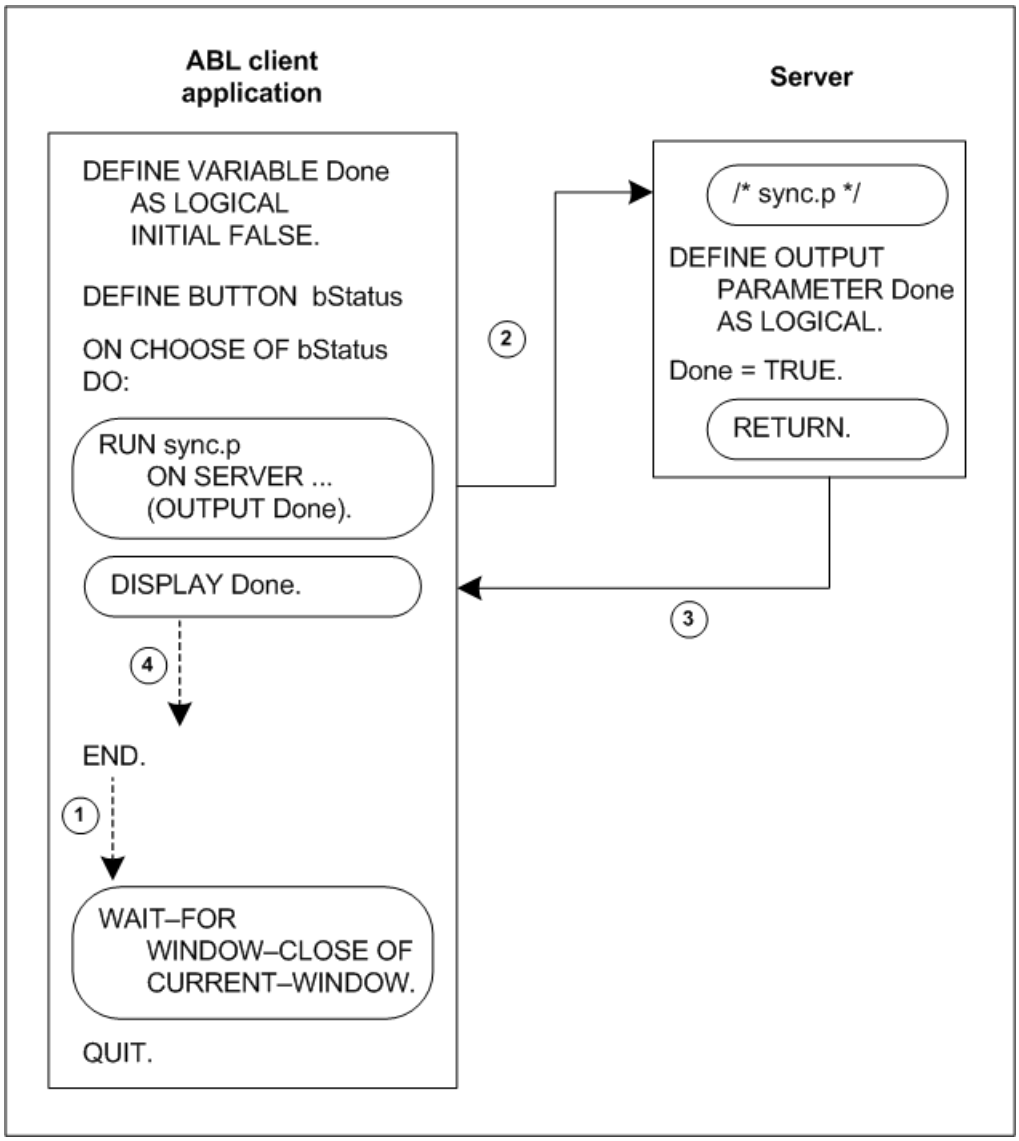
Synchronous request execution model

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 server 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.

Figure 4: Synchronous requests



Asynchronous request execution model

The following figure shows a remote procedure executed asynchronously. As in the synchronous request example, 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 server 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 a different server connection.

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 client session 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 this asynchronous request example provides the same functionality as the previous synchronous request example. In fact, `sync.p` and `async.p` are identical, except for their names, which are changed for readability. The PAS for OpenEdge instance 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 asynchronous request example tests the status of the remote request and performs actions based on whether the request finished, rather than invoking the request itself, as in the synchronous request example. 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 a server. 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 `OnProcedureComplete` event method defined in the *`callbackHandlerObj`* ABL object, which is an instance of the *`CallbackHandler`* class. The `callbackHandlerObj` method displays the customer number. After the callback method executes, the *`client.p`* procedure deletes the asynchronous request handle *`asyncRequest`*, the server connection handle *`appserverConnection`* and the callback method object instance *`callbackHandlerObj`*.

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 terminates after executing the callback method registered for the `PROCEDURE-COMPLETE` event.

client.p

```
var character customerName = "WILNER".
var handle asyncRequest.
var handle appserverConnection.
var logical lReturn.
var CallbackHandler callbackHandlerObj.
callbackHandlerObj = new CallbackHandler().

create server appserverConnection.
lReturn = appserverConnection:connect("-URL http://localhost:8810/apsv").

run server.p on appserverConnection asynchronous set asyncRequest
event-handler "OnProcedureComplete"  event-handler-context callbackHandlerObj
(input customerName, output customerNum as integer).

wait-for procedure-complete of asyncRequest.

delete object asyncRequest no-error.
appserverConnection:disconnect().
delete object appserverConnection no-error.
delete object callbackHandlerObj.
end.
```

server.p

```
define input parameter customerName as character no-undo.
define output parameter customerNum as integer no-undo.

for first customer where customer.name = customerName no-lock:
 customerNum = customer.cust-num.
end.
```

CallbackHandler.cls

```
class CallbackHandler:
    method public void OnProcedureComplete( customerNum as integer):
        display customerNum.
    end.
end.
```

4

Design and Implementation Considerations

This chapter highlights specific performance, transaction management, deployment, security, and internationalization issues related to planning and implementing a distributed application with PAS for OpenEdge. It explains some programming techniques, especially those commonly used by both ABL client and PAS for OpenEdge 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 three fundamental performance reasons for implementing an application using PAS for OpenEdge:

- 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

- To support critical business functions for a maximum client load across the Internet

These three points can work together to provide maximum performance benefits. The following sections discuss each point separately.

Minimizing network traffic

When you deploy a PAS for OpenEdge instance 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 a PAS for OpenEdge instance on the database machine, 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 PAS for OpenEdge, 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 PAS for OpenEdge instance uses the same query that was used when the application was executed using the traditional client/server model. However, using PAS for OE, 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 PAS for OpenEdge 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 PAS for OpenEdge 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 PAS for OpenEdge.

Choosing an application model

You can choose to build a PAS for OpenEdge application that runs in one of the two application models, session-managed or session-free. You can also build an application that runs in a combination of the two models, where a client can connect using one model or the other, depending on the application features that the client needs to access on a PAS for OpenEdge instance.

In addition, the session-managed model supports two connection states, bound and unbound, that are determined by your PAS for OpenEdge application at run time. The session-free model also provides support for bound connections, but PSC recommends that you avoid them. For information on how each application model affects the PAS for OpenEdge and client operation, see PAS for OpenEdge and Client Interaction on page 11

Application characteristics to consider

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

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

The best application model 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 PAS for OpenEdge machine (or machines)

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

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

Table 7: Characteristic application model trade-offs

Application Model	Design complexity	Resource consumption	Request throughput (requests/second)	Average response time (seconds/request)
Session-managed	Depending on the connection binding, shared client and PAS for OE session context can be established and managed for each request using the Activate procedure, then reset using the Deactivate procedure. When an unbound connection state is used, tends to be extremely complex if context is complex.	Many clients per PAS for OpenEdge session (unbound); One client per PAS for OpenEdge session (bound)	Maximum throughput under heavy client load where context is maintained, computing resources are limited, and requests are short. Assumes none or minimal use of bound connections. Bound connections of long duration can significantly limit throughput. Even when connections are unbound, large complex connection context can also limit throughput.	With a bound connection, a large shared context, and the same computing resources, performs requests faster for the bound client than for clients making requests over an unbound connection.
Session-free		Varies directly with client load, but tends to support many more clients per PAS for OpenEdge instance and session	Highest throughput under heavy client load, especially where multiple PAS for instances are available to support a single business application using a DNS load balancing service. Offers the greatest scalability, especially where no or minimal shared context is maintained between clients and sessions.	Has best possible response time for heavy client load with none or minimal shared context, and using the same computing resources as a session-managed connection. Performs less well for a given client than using a bound session-managed connection, especially given significant shared context.

Application Model	Design complexity	Resource consumption	Request throughput (requests/second)	Average response time (seconds/request)
	Easiest to program where little or no context is maintained. Any shared client and PAS for OpenEdge context must be established and managed for each request using the Activate procedure. You can reset the context using the Deactivate procedure. This application model tends to be the most complex if context is complex. Because no Connect procedure executes for the client connection, requires a separate login verification in the Activate procedure to establish or retrieve the current login session for each client request.			

Managing larger amounts of context

If you application has a large amount of context to manage between each request, then the session-managed application model in a bound state is the most appropriate choice. An unbound connection 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.

Note that regardless of application model, for a given PAS for OpenEdge instance, all self-service database connections are shared among all of its ABL sessions and must be started up in the Multi-session Agent Startup procedure and shutdown in the Multi-session Agent Shutdown procedure.

.

Managing smaller amounts of context

If there is none or 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 session-free application model might be the appropriate choice for your application. This increases your ability to service a large number of clients, because no server session is dedicated to any particular client. The minimal context that needs to be maintained between requests can be stored within an application-specific context database. No matter where you store the context, you do need to use the Activate and Deactivate procedures to manage that context for each request.

Considering ample computer resources

Note that for the most part, using the unbound session-free, and especially the session-free application model, makes the best sense when you must provide high throughput with minimal computing resources. On the other hand, if you only have minimal context that needs to be maintained between requests, and you have plenty of computing resources relative to your current and expected client load, there is no inherent advantage to using the session-free, or even the unbound sesssion-managed model. Under these more favorable conditions, even if there is only minimal context to manage, using the session-managed model in a bound connection might provide a somewhat simpler programming model.

Considering the future

When considering whether to use the unbound session-managed or session-free application model, be sure to consider your current and future performance goals. Redesigning a bound session-managed application to be unbound session-managed or session-free can be very complicated. On the other hand, if your current goals do not require either the unbound session-managed or session-free model, one of these modes might still be the right choice based on your future performance requirements. Note especially that session-free is the most scalable application model of all.

Setting the number of multi-session agents, sessions, and connections

For best performance, you must determine the optimal settings for the PAS for OpenEdge properties that control the number of multi-session agents, sessions, and client connections that a session manager manages:

- **Initial number of multi-session age to start** (`AppServer.SessMgr.numInitialAgents` property in `openedge.properties`)
- **Maximum number of multi-session agents to start** Instances (`AppServer.SessMgr.maxAgents` property in `openedge.properties`)
- **Maximum ABL sessions per multi-session agent** (`AppServer.SessMgr.maxABLSessionsPerAgent` in `openedge.properties`) o
- **Maximum connections per multi-session agent** (`AppServer.SessMgr.maxConnectionsPerAgent` in `openedge.properties`)

In general, you must consider the number of clients, the design of your application (including the application model), and the hardware resources that run your PAS for OpenEdge instance. At a minimum, follow these guidelines in setting these values:

- **For a session-managed application** — You must have one ABL session for each client that connects to the PAS for OpenEdge instance. That is, you need as many ABL sessions as clients that connect concurrently to the PAS for OpenEdge instance.

- **For a session-free application** — You can have one session work on requests for multiple clients. At a minimum, Progress Software Corp. recommends one server session for each CPU in the machine. You can expect each multi-session agent to handle requests from up to 20 PAS for OpenEdge 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-session client load to match your total PAS for OpenEdge client load. The per-session client load also depends on the length of time it takes to complete an average client request. The longer it takes for an session to complete session requests, the fewer sessions are supported by each session, and the more sessions you need to handle the total PAS for OpenEdge client load. Thus, the total number of clients required is very application specific.

For more information on setting the server pool parameters, see Progress Application Server for OpenEdge: Administration Guide *OpenEdge Application Server: Administration*.

Choosing a transaction model

The client application and PAS for OpenEdge run in two separate sessions. Transactions are not propagated between these sessions. However, PAS for OpenEdge 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 server session 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 server 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 server session.

For more information on trade-offs and issues for both types of transactions, see Transaction and record management considerations on page 116.

Using load balancing

For a session-free application with a large client load, you might want to use a Domain Name System (DNS) load balancing service to distribute client requests among several identical PAS for OpenEdge instances hosting the same OE ABL Web application. You typically connect to the DNS through a port that is configured to accept requests for your Web application hosted on these instances. The DNS then distributes client requests depending on the load balancing model that is used. Some DNS systems consider the actual client load reported by each server (PAS for OpenEdge instance) that services client requests, or like Apache Camel, allow you to configure access to servers (endpoints) according to one of a variety of load balancing policies. For more information, see the documentation for a given DNS load balancer.

Secondary performance considerations

This section highlights additional specific performance considerations that you should review and understand as you design and implement an application using PAS for OE.

Connect and disconnect performance issues

This section discusses performance issues related to connecting to and disconnecting from a PAS for OpenEdge instance.

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 a PAS for OpenEdge instance Connect procedure.

Using the PAS for OpenEdge Server Connect procedure

The PAS for OpenEdge 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 client connection using the session-free application model never executes a Connect procedure. To perform initial application tasks that apply to all session-free clients of a PAS for OpenEdge instance, you must use the Multi-session or Session Startup procedure, as appropriate.

You might want to perform any of the following activities independently of each other, or in combination with any other items:

- Authenticate and authorize access to PAS for OpenEdge resources based on additional parameters you can pass to the Connect procedure (*user-id*, *password*, and/or *app-server-info*)
- For a session-managed application, provide security for procedure execution by using the EXPORT( ) method to define the allowed export list for client applications

  ---

  Note: For a session-free application, you must set the export list in the Session Startup procedure.

- Connect to one or more databases, or to the other OpenEdge application servers

  ---

  Note: To make a self-service database connection for a PAS for OpenEdge instance, you must use the Multi-session Startup procedure.

- 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 125 for more information.

Performance-related issue for the CONNECT( ) method

Connecting to session-managed business applications can be expensive depending on session startup and initialization tasks defined in your PAS for OpenEdge Connect procedure. Connecting to a session-free business application can be expensive depending on the client load for the available server resources on the network. For these reasons, you might want to connect to servers at client application startup. Also, in most designs, it is advisable that you do not disconnect from servers if you intend to use services offered by a particular PAS for OpenEdge instance more than once prior to ending the client application session.

Performance-related issues for the DISCONNECT( ) method

On a bound session-managed connection, the DISCONNECT( ) method has no significant performance impact on your client application regardless of whether there is an server Disconnect procedure defined. The server Disconnect procedure is run by the PAS for OpenEdge server after the client connection is terminated and DISCONNECT( ) method execution has completed. That is, unlike other procedures executed in a server session, the server Disconnect procedure is run for a bound session-managed connection concurrently with processing occurring in the client application.

For a bound session-managed connection, you might be able to take advantage of the way the PAS for OpenEdge Disconnect procedure is run to improve overall application performance. When designing your distributed application, consider tasks to be performed by a PAS for OpenEdge instance that could be performed independently of the connection between the client application and the PAS for OpenEdge instance. 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 PAS for OpenEdge Disconnect procedure, these actions can be performed concurrently with other client application processing, improving overall application performance.

On an unbound session-managed connection, 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 the server session to handle other client requests.

In a session-free application, the DISCONNECT( ) method has minimal performance impact because no Disconnect procedure is executed.

Compiled compared to precompiled code

The execution process for non-compiled code on PAS for OpenEdge is identical to the process for regular run-time compiled code on an ABL client. Standard ABL compilation processing activities and behaviors occur.

Note: The PAS for OpenEdge production server does not compile ABL, and therefore cannot run non-compiled code in a deployment environment. You can only run non-compiled code using the PAS for OpenEdge development server.

Therefore, when non-compiled code must be used, from a performance perspective, there is the possibility of some improvement gains when the server is running on a more powerful machine than the client application.

For example, if the server you are running is on a faster machine, then the compilation process might occur faster for code compiled on it. Also, if PAS for OpenEdge sessions 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 a PAS for OpenEdge instance the following actions must occur:

1. A query running on the server 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 server session, 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 server session building the original query is bound to the client application in a session-managed connection.

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 server session 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 PAS for OpenEdge applications, garbage collection will run always before a switch in the connection context. Thus, PAS for OpenEdge applications running in any application model or connection binding are protected by garbage collection. For more information on garbage collection for 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. For a session-free application, all transactions must be normal ABL transactions: 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:

```
RUN startTransaction.p ON SERVER ...
RUN commitTransaction.p ON SERVER ...
```

Another example is the interface provided by `a-txtest.p` (see Programming the Progress Application Server for OpenEdge on page 25).

However, you might better simplify this process for the client with a more abstract interface, called from the client, as shown:

```
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 PAS for OpenEdge is their small scope, requiring no communication with the client for the transaction to complete. In general, to implement a normal ABL transaction in a server 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 server session 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 a PAS for OpenEdge 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 server transaction status 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 server 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 server 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 server

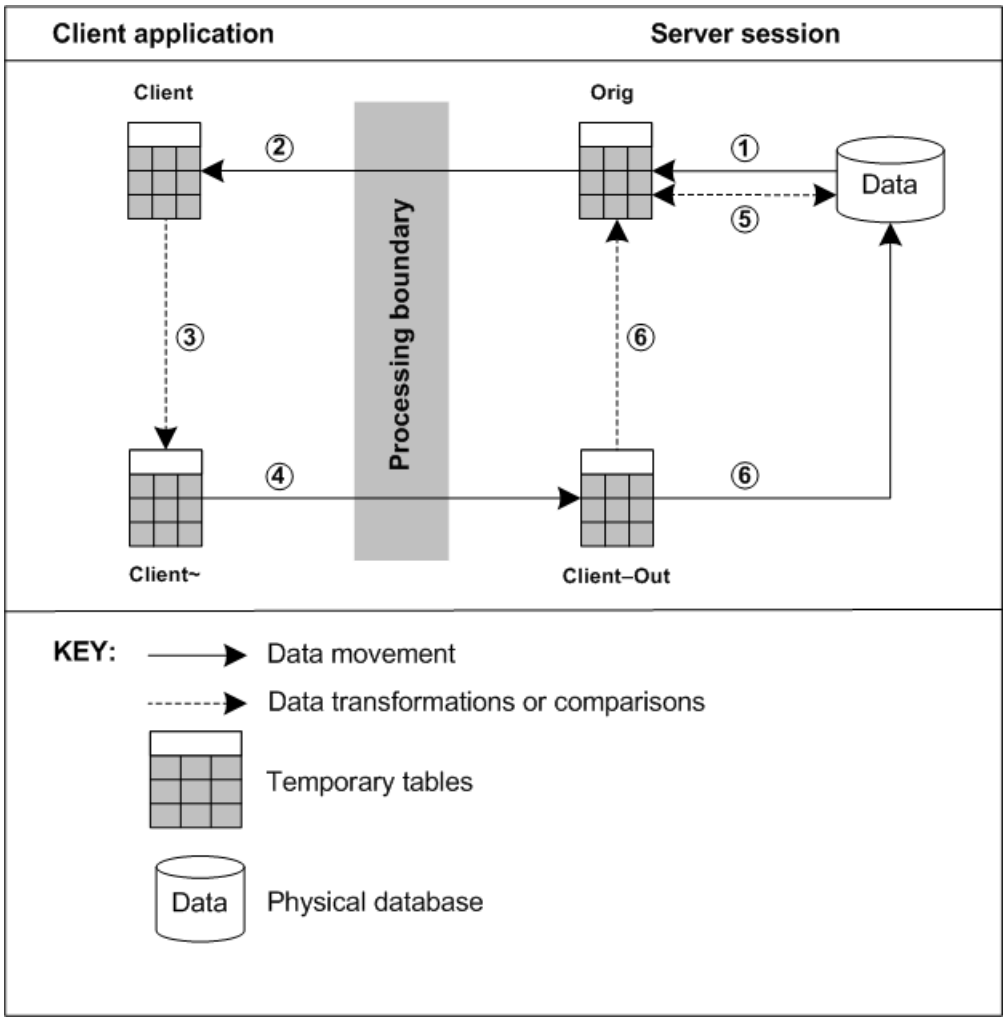
One disadvantage of using normal ABL transactions on the PAS for OpenEdge is that, because a normal ABL transaction completes and releases all locks before returning to the client, you may have to do extra work to ensure that the transaction completes as the client expects.

A typical scenario where this behavior might pose a problem is where the server returns data derived from a database to a client application. The client then modifies the data and returns it to the server 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 server, the server 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 to have the server keep a copy of the original data that is returned to the client application. In a server session, you can maintain a separate buffer or temp-table to hold a copy of the data. 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 unbound session-managed connection context in Programming the Progress Application Server for OpenEdge on page 25.

When it is time to apply the updated data, the server session 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 5: Transaction data flow



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 server to request the data. The PAS for OpenEdge instance 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 server 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 server by running an internal procedure of the persistent procedure instantiated in Step 1. The server stores its copy of the table in **Client-Out**.

5. Within a single transaction, the server compares the records that are in the **Orig** temp-table to the data in the database. If there are any differences, the server session 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 server 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 server 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 server 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 server, and skip that when applying the changes in Step 6. Only you can decide what approach is appropriate for your application.

Note: You can greatly simplify this process by using an ABL dataset object (ProDataSet) to hold the temp-table or temp-tables involved in the transaction, with all of them defined using the `BEFORE-TABLE` option. This option provides before-image support in the ProDataSet. By exchanging this ProDataSet as an `INPUT-OUTPUT` parameter of the remote procedure, the server can apply and manage database changes using methods and events of the ProDataSet, and the client can also respond to any errors that are returned for each changed record on the server. For more information, see *OpenEdge Development: ProDataSets*.

Managing conflicts between client and server

All server 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 server access the same database, each one accesses the database as a different user, whether or not the client is connected to the server. As a result, any combination of server clients and server 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 server 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 server, 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 server session (specified as part of the `sessionStartupParam` property value in your PAS for OpenEdge `openedge.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 server. The data source is managed solely by the server, 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 121

- On the server on page 121

On the client

When a lock wait time out occurs on the server because of a lock conflict that exceeds the time specified by the Lock Wait Timeout (-lkwtmo) startup parameter, a STOP condition is raised on the client side.

You cannot trap this stop condition simply by checking ERROR-STATUS:ERROR for a RUN statement that executes a remote procedure. The only way to handle this situation is to use the 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("-URL http://localhost:8810/dbmanager/apsv").
    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 lockrecords.p terminates on the server because of a lock wait time out, the server raises a STOP condition on the client. The client program flow never reaches the message following the RUN statement that executes lockrecords.p, However, the client can catch the STOP condition and handle character string returned in RETURN-VALUE.

On the server

On the server, another technique for managing lock conflicts is to use the NO-WAIT and NO-ERROR options on any FIND statement that also applies an EXCLUSIVE-LOCK to the record. You can then test the resulting record buffer using the 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 NO-LOCK

If you try to find a record with NO-LOCK and that record is already located in a buffer, then depending on the exact nature of the request, The ABL Virtual Machine (AVM) 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 server), 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 a server 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 to a request a server over a session-free or unbound session-managed connection. The server session that processes the request updates some record. The client then sends another request to the server. This request happens to go to a server session that is different then the one that processed the first request. This second server session 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 client's request executed by this server session, the AVM may not reread the record. When this occurs, the server session's 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 122
- Releasing the record on page 122
- Setting the -rereadnolock parameter on page 122

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 by setting the sessionStartupParam property in the PAS for OpenEdge openedge.properties file for the appropriate server 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 server.

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 PAS for OpenEdge to minimize the effects of schema changes by masking the changes from the client application.

The following list identifies specific techniques you can use to shield these schema changes from affecting client applications:

- Set up database access so that it is accomplished entirely through server sessions (the client is not connected to the database and all records are transferred from the server session through temp-table or ProDataSet parameters)
- Avoid using LIKE syntax for temp-tables and variables in client application code and in procedure parameters for remote procedures
- Server session 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

Note: You can run r-code that was compiled by OpenEdge running on a 32-bit platform in a PAS for OpenEdge instance, but PAS for OpenEdge only installs and runs on a 64-bit platform.

For more information, see the sections on portable r-code in *OpenEdge Deployment: Managing ABL Applications*.

Using portable r-code, you can perform PAS for OpenEdge development on a platform that is different from your PAS for OpenEdge deployment platform. For example, you can compile r-code that is destined to run on a UNIX 64-bit PAS for OpenEdge session 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 server session on a UNIX host. Therefore, if there can be multiple deployment platforms for server session code, you should use run-time resolution for platform-specific blocks as the second code example in this section shows.

Note: For for both `&OPSYS` and OPSYS, the value "`WIN32`" refers to both 32-bit and 64-bit Windows platforms. If you want to know if the current platform is 32-bit or 64-bit, test the value of `&PROCESS-ARCHITECTURE` or `PROCESS-ARCHITECTURE`.

Security considerations

The general security framework for your networking system is governed by security elements associated with your Spring Security configuration to connect to each PAS for OpenEdge ABL Web application using its supported transports, your operating system, and your file system

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 server security options

The following table summarizes the various application-specific security options available.

Table 8: Server security considerations

Security consideration	Design strategy
User authentication and authorization on page 125	Use the PAS for OpenEdge Connect procedure to validate a user
Database access on page 127	Limit or prevent client application access to a database
PAS for OpenEdge session access on page 127	Limit users' access to specific procedures through the `EXPORT( )` method
Audit trails on page 130	Use the PAS for OpenEdge Connect and PAS for OpenEdge Disconnect procedures to assist in management of audit trails
Digital certificate management on page 131	Support data privacy over HTTPS connections to PAS for OE.

The remaining sections on security provide additional information about each of these considerations.

For information about developing these features, see Programming for a PAS for OpenEdge application model on page 26. Also, for more information about standard ABL security features, see *OpenEdge Development: Programming Interfaces*.

User authentication and authorization

You typically handle authentication at the point by specifying user credentials in the URL to connect to the PAS for OpenEdge ABL Web application., which can accept parameters for user authentication and refuse a client connection according to the result. For a bound session-managed application, you can also add an additional level of authentication using the Connect procedure. From the Connect procedure, you can also authorize:

- What remote procedures (entry points) can be run by the connected client by setting an export list using the EXPORT( ) method.
- Connections to databases and other OpenEdge application servers from the connected server session. For more information on limiting database access through a connected server session see Database access on page 127.

However, for an unbound session-managed or a session-free application, you cannot directly authorize and implement these options at connect time.

Authorization over a bound session-managed connection

The following code shows a typical authentication and authorization example:

```
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 server session 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 tighter security model, establish an export list in conjunction with OpenEdge authentication of a client-principal that you initialize with pcPassword and pcUserId to restrict access from the client applications host to the remote procedure sources. For more information on authenticating with a client-principal and OpenEdge security, see sections on application security in *OpenEdge Development: Programming Interfaces*.

Authorization over an unbound session-managed or session-free connection

For an unbound session-managed application, you cannot easily set an export list and database connections at connect time, because the server session that runs the Connect procedure is not necessarily the one that executes subsequent remote procedure requests for the connected client. For a session-free application, you cannot perform these functions at connect time at all, because the connected server never runs a Connect procedure. For session-managed and/or unbound or bound PAS for OpenEdge instances, setting an export list and database connections works much more effectively as a global activity that you implement in the Session Startup procedure, which executes for every server session when it starts up. In this case, you also need to set the database connection identity in the Activation procedure for each request. For more information on setting database connection identity, see the sections on application security in *OpenEdge Development: Programming Interfaces*.

You can also pass all of the connection-based authentication and authorization information to each server session that handles a remote procedure request for a client connection. The server session can retrieve and re-save the authorization information for the next server session that handles the connection using the Activate and Deactivate procedures. For more information on using these procedures, see Programming for a PAS for OpenEdge application model on page 26. For more information on creating an export list using the EXPORT( ) method, see the Application model context management on page 13.

Database access

The traditional client/server model requires a connection from the ABL client to the database server. PAS for OpenEdge 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 the PAS for OpenEdge instance. Such an implementation provides tighter security since database access is limited to the parameters, and functionality contained in the procedures that are remotely executed on the server.

A less severe strategy is to allow the ABL client read-only access to the database, but provide full update access to the server session. This preserves full control of updates through the restricted set of procedures that can be run in the server session, 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 ABL browse widget behavior. See ABL browse design considerations on page 115 for detailed information about browse behavior.

PAS for OpenEdge session access

You can publish entry points (remote procedure pathnames) that control access to a PAS for OpenEdge session at run time using the EXPORT( ) method on the 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, especially over a bound session-managed connection, 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 PAS for OpenEdge PROPATH setting.

However, setting this list applies only to the server session in which you execute the EXPORT( ) method. This means that you might have to manage the list differently in your application depending on the application model and binding of the connection.

Note: As this is a context management problem, the only way to effectively manage this kind of information for a session-free application is to maintain your own context database for it.

For more information on setting and resetting entry points in a server session, see Programming for the application model on page 64 .

Application model interactions

Because of how the EXPORT( ) method and application models work together, you must use it differently, depending on your server. 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 the server session.

If the PAS for OpenEdge session servicing a session-managed request, 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, for a bound session-managed application, the number of entry points set for the server session can grow with each client connection, and for an unbound session-managed application, the number can grow with each client request.

How to set and reset an export list for each application model

Each combination of application model and connection binding requires a different arrangement for setting and resetting the published entry points for a given client:

- **Session-managed application with a bound connection on page 128** — You can set an export list any time before a client application uses it; if you set them in remote procedures, you probably want to reset it with each connection.
- **Session-managed application with an unbound connection on page 128** — You can set an export list any time before a client application uses it; you generally set it in the Session Startup procedure. You can also set it with a finer degree of session granularity, but you must exercise caution when doing so.
- **Session-free application on page 129** — You can most practically set an export list only in the Session Startup procedure. You can also set it with a finer degree of session granularity, but you must exercise caution and consider the connectionless nature of session-free remote requests when doing so.

Session-managed application with a bound connection

For a session-managed application with a bound connection, you can conveniently set a single export list for all server sessions, regardless of the connection, using the Session Startup procedure. Otherwise, you can set it, like on a session-managed server, in the Connect procedure or some remote procedure executed by the client.

When the connection terminates for a bound session-managed application, the disconnected server session 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 server session by resetting the list in the Connect or Disconnect procedure, as shown:

```
lReturn = SESSION:EXPORT(). /* Reset to empty */
```

Session-managed application with an unbound connection

For a session-managed application with an unbound connection, you can conveniently set a single export list for all server sessions, using the Session Startup procedure. Because export list settings apply only to the server session in which they occur, setting and resetting an export list at any other point in an unbound session-managed application requires careful attention.

Caution: Setting an export list during remote procedure requests from an unbound session-managed client 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 application using a bound connection, instead.

Once you set an export list, the list remains set in the server session until you reset it. For an unbound connection, because you cannot guarantee when an server session 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 sessions in the server sessions pool.

For an unbound connection, you can maintain consistent export list settings using a combination of the Connect, Activate, and Deactivate 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 context storage, such as the `SERVER-CONNECTION-CONTEXT` attribute or a context database.
3. In the Activate procedure, retrieve the export list from your 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 server session to handle remote procedures calls from all connected clients, 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 server session other than the one where the last export list setting occurred.

You can also use the Activate procedure to make other 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 context storage or from some other source available to ABL. For more information on using context storage, see Programming the Progress Application Server for OpenEdge on page 25.

Session-free application

For a session-free application, you cannot perform these functions at connect time at all, because the application never runs a Connect procedure. For session-free applications, setting an export list and database connections works much more effectively as a global activity that you implement in the Session Startup procedure, which executes for every server session 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 server session that handles a remote procedure request. The server session can retrieve and re-save the export list information for the next server session that handles the client using the Activate and Deactivate procedures.

Caution: Setting an export list during remote procedure requests from a session-free client 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 server using the session-managed model with a bound connection, instead.

To maintain consistent export list settings using the Activate and Deactivate procedures:

1. In the Session 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 context storage, such as a context database or operating system file.

   Note:

   This context storage **must** be accessible to all PAS for OpenEdge instances that handle requests distributed by a DNS load balancer for the same business application.

3. In the Activate procedure, retrieve the export list from your 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 re-save the reset export list to your context storage. This leaves the current or any other server session in any server that supports the same session-free application 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 Manage client context for session-free and unbound session-managed connections on page 46.

PAS for OpenEdge application audit trails

In a session-managed application, you can access the current connection ID using the SERVER-CONNECTION-ID attribute on the SESSION handle. In a session-free or session-managed application, 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 in a session-managed application include the 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 server. 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.

In a session-free or unbound session-managed application, you can also use the Activate and Deactivate procedures to generate audit trails for each request. However, this use can reduce performance, depending on the load on your server 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 Progress Application Server for OpenEdge on page 25 For more information on accessing the ClientContextId property, see Manage client context for session-free and unbound session-managed connections on page 46.

Note: Audit information created in a server session using the MESSAGE statement or other external output syntax such as the DISPLAY statement, is sent to the PAS for OpenEdge instance log file unless otherwise directed.

Client application audit trails

In 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 to the PAS for OpenEdge instance. 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 also 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 PAS for OpenEdge 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 77. For information on the common methods for accessing the connection ID from an open client, see *OpenEdge Development: Open Client Introduction and Programming*.

Digital certificate management

You can use HTTPS (implemented with TLS) to securely connect a client to and communicate with PAS for OpenEdge. HTTPS 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 digital certificates, and both 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 (TLS client) and the responder (TLS server).

The server is identified by the private key that it stores and the client is identified as a valid TLS client for that server by the public key that it stores and provides to the server. TLS 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 HTTPS server. You can use Tomcat tools to manage these certificates and private keys for a PAS for OpenEdge instance. For more information, see the sections on HTTPS support in the *Progress Application Server for OpenEdge: Administration Guide* and the Apache Tomcat documentation on managing the certificate and key store using the `keytool` command-line utility.

OpenEdge uses the `certutil` command-line tool to manage public key certificates for ABL clients. The client can then connect to a corresponding PAS for OpenEdge instance using an HTTPS URL. For more information on managing the public key certificates using `certutil`, see the sections on managing OpenEdge key and certificate stores in *OpenEdge Getting Started: Installation and Configuration*. For information on connecting to PAS for OpenEdge from an ABL client using HTTPS, see the sections on using the `-URL` connection parameter in Establishing a connection with the CONNECT( ) method on page 72.

You use Microsoft tools to manage public key certificates for .NET Open Clients and use the OpenEdge `procertm` command-line tool to manage public key certificates for Java Open Clients. The Open Client connects to a PAS for OpenEdge instance using an HTTPS URL very similar to an ABL client. 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 *OpenEdge Development: Java Open Clients*. For information on connecting to a PAS for OpenEdge instance from an Open Client, see the sections on connecting to a server using HTTPS in *OpenEdge Development: Open Client Introduction and Programming*, and Connection URL syntax for the APSV transport on page 74 in this manual.

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 server procedures to coordinate different languages between the client and PAS for OpenEdge sessions. In an unbound session-managed or session-free application, 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 server session 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 in the Session Startup procedure (using the -cpcase startup parameter) that the server session uses to handle case conversions.
- **Year offset (-yy startup parameter)** — Using the Century (-yy) startup parameter, you can ensure that the server session 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 PAS for OpenEdge, and to migrate your application code from your present configuration to a distributed PAS for OpenEdge session environment.

Understanding the scope of persistent, single-run, and

A remote procedure on PAS for OpenEdge 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 server session where the procedure runs. This is consistent with the existence of the processing boundary that separates a server 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 server session in which they were created. These objects cannot be shared between an ABL client application and the server session.

Cleanup of remote procedure instances

Persistent, single-run, and singleton procedures differ significantly with respect to the removal of instances from the server. 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 server session 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 a singleton instance of the same procedure is already running. In such a scenario, the server session deletes the singleton instance and runs the procedure in single-run mode. The server session 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

Every active ABL procedure, whether running on an ABL client or in a server session, 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 a server 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 server 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 server session 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 9: PROXY, REMOTE, and PERSISTENT procedure handle attribute settings

Procedure call type	Caller handle	Callee handle (THIS-PROCEDURE)
Local procedure	Not applicable	PROXY = FALSE REMOTE = FALSE PERSISTENT = FALSE SINGLE-RUN = FALSE SINGLETON = FALSE
Remote procedure	Not applicable	PROXY = FALSE REMOTE = TRUE PERSISTENT = FALSE SINGLE-RUN = FALSE SINGLETON = FALSE
Local persistent procedure	PROXY = FALSE REMOTE = FALSE PERSISTENT = TRUE SINGLE-RUN = FALSE SINGLETON = FALSE	PROXY = FALSE REMOTE = FALSE PERSISTENT = TRUE SINGLE-RUN = FALSE SINGLETON = FALSE
Remote persistent procedure	PROXY = TRUE REMOTE = FALSE PERSISTENT = TRUE SINGLE-RUN = FALSE SINGLETON = FALSE	PROXY = FALSE REMOTE = TRUE PERSISTENT = TRUE SINGLE-RUN = FALSE SINGLETON = FALSE
Local single-run procedure	PROXY = FALSE REMOTE = FALSE PERSISTENT = FALSE SINGLE-RUN = TRUE SINGLETON = FALSE	PROXY = FALSE REMOTE = FALSE PERSISTENT = FALSE SINGLE-RUN = TRUE SINGLETON = FALSE
Remote single-run procedure	PROXY = TRUE REMOTE = FALSE PERSISTENT = FALSE SINGLE-RUN = TRUE SINGLETON = FALSE	PROXY = FALSE REMOTE = TRUE PERSISTENT = FALSE SINGLE-RUN = TRUE SINGLETON = FALSE
Local singleton procedure	PROXY = FALSE REMOTE = FALSE PERSISTENT = FALSE SINGLE-RUN = FALSE SINGLETON = TRUE	PROXY = FALSE REMOTE = FALSE PERSISTENT = FALSE SINGLE-RUN = FALSE SINGLETON = TRUE
Remote singleton procedure	PROXY = TRUE REMOTE = FALSE PERSISTENT = FALSE SINGLE-RUN = FALSE	PROXY = FALSE REMOTE = TRUE PERSISTENT = FALSE SINGLE-RUN = FALSE

Procedure call type	Caller handle	Callee handle (THIS-PROCEDURE)
	SINGLETON = TRUE	SINGLETON = TRUE

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 a PAS for OpenEdge 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 10: ABL conditions

Condition	Description
ERROR	An unhandled ERROR condition raised in a client application has no effect on any server sessions to which it is connected. Conversely, an unhandled ERROR condition raised in a server session has no effect on the client application to which it is connected. Handling this condition conforms to standard ABL rules. For more information about these rules, see *OpenEdge Getting Started: ABL Essentials*. The only way a server session can raise ERROR in the client application is for a remote procedure or an Activate procedure to execute the RETURN ERROR statement. Either one raises ERROR on the RUN statement in the client application, which can also be returned in a CATCH block on the client..
ENDKEY	An unhandled ENDKEY condition raised in a client application has no effect on any server sessions to which it is connected. Conversely, an unhandled ENDKEY condition raised in a server session 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 *OpenEdge Getting Started: ABL Essentials*. A server session raises the ENDKEY condition if an attempt to find a database record fails or an INPUT FROM statement reaches the end of

Condition	Description
	an input stream. Like a batch client, there are no user-interface events that can raise the ENDKEY condition.
STOP	A STOP condition raised in a client application while it is executing a remote procedure causes a STOP condition to be raised in the context of the executing procedure in the server session. An unhandled STOP condition in a server session results in the request being returned with the STOP condition. **On a root client application** — An unhandled STOP 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 server sessions it is connected to and deletes active remote persistent procedures. **In a server session** — An unhandled STOP condition causes the STOP 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 server session. All active remote persistent procedures and server session connections remain intact.
QUIT	An unhandled QUIT condition raised in a client application disconnects the client application from all server sessions it is connected to and deletes the proxy procedure handles for that client application. An unhandled QUIT condition raised in a server session results in the immediate, successful completion of the current remote procedure request. The client application is then automatically disconnected from the server session. **Note:** Executing QUIT as the final statement in the Disconnect procedure for a bound session-managed connection causes the AVM to clear all context, such as globally shared variables, from the server session before it disconnects from the client.

Schema triggers

Schema trigger code defined in the database is executed by database clients. Any database access by a PAS for OpenEdge session that invokes database trigger code causes the server session to run such code. This is because the server session acts as a client to the database.

There is no functional difference between how schema triggers run in a server session and how they run in an ABL client session. However, trigger code executed by a server session can:

- Fail if it requires user input or output to a user interface other than a character user interface
- Send any output to the PAS for OpenEdge log file, unless specifically directed otherwise

Invoking schema triggers on a client application raises a basic security issue. Code in the server session is protected because of *encapsulation* (the session barrier that isolates server code from the client); code on your client application is not protected.

Conditions that make migrating your application model

Certain conditions related to your current application framework allow you to more easily take advantage of PAS for OpenEdge functionality. These conditions include any of the following:

- Your present server machine has adequate resources such as memory, speed, and disk space to be easily scaled up for use with PAS for OpenEdge.
- Your current model has a predominance of small, well-structured, identifiable transactions. The server favors a small transaction model. For more information, see Transaction and record management considerations on page 116.
- You are already using temp-tables to manipulate data before committing the data to the database. Or, if you are not using temp-tables and (especially) ProDataSets, it is easy to get your present code to work with temp-tables and ProDataSets 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 data cannot be shared between a client application and a server session.
- 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 PAS for OpenEdge.