OpenEdge Getting Started: Guide for New Developers

21 June 2023

Copyright

Table of Contents

1

Introducing OpenEdge Platform

Progress® OpenEdge® platform offers a complete, out-of-the-box solution for developing, integrating, and managing business applications offered as a service or on-premise. It is an integrated development environment that helps build high-performance multi-platform systems.

OpenEdge is designed to assist software developers and business analysts to become more responsive to market and customer needs now and in the future. With OpenEdge, service and application development is agile and cost-effective, and the resultant applications are reliable, easy-to-maintain, cost-effective, and service enabled.

Related Links

- Getting Help using the OpenEdge Knowledge Services Set
- What is OpenEdge
- A comprehensive view of the components in OpenEdge
- OpenEdge client-server process architecture

Getting Help using the OpenEdge Knowledge Services Set

This topic describes how to locate and use the different types of help that is available for Progress OpenEdge Platform.

- **OpenEdge Documentation Set in PDF**: Download the OpenEdge Release 11.4 documentation set in PDF format from the location provided below. Once you have downloaded and extracted the files to a desired location, open the start.pdf to access each document in the set.

  https://community.progress.com/technicalusers/w/openedgegeneral/1329.openedge-product-documentation-overview.aspx

- **OpenEdge Documentation Set in HTML**: Download the OpenEdge Release 11.4 documentation set in HTML format from the location provided below. Once you have downloaded and extracted the files to a desired location, open the index.html to access each document in the set.

  https://community.progress.com/technicalusers/w/openedgegeneral/1329.openedge-product-documentation-overview.aspx

  Note: You cannot view the downloaded HTML files using the Chrome browser.

- **Cheat sheets**: The Eclipse framework provides cheat sheets as guides for completing complex procedures. This release provides a number of cheat sheets that supplement the procedural help for Progress Developer Studio for OpenEdge. Choose **Help > Cheat Sheets** from the main menu bar to view the list of available cheat sheets.
- **Task Maps**: Task maps provide a visual representative method to locate information about performing key tasks related to OpenEdge Platform. The task maps can be accessed from the Getting Started portal, available in the Communities page.
- **Demos**: There are a variety of demos, hosted on the PSDN website, that illustrate the features of OpenEdge Platform.

  Note: The OpenEdge Release 10.2B demos have not been updated for OpenEdge Release 11.4. Since 10.2B, OpenEdge Architect has been renamed as Progress Developer Studio for OpenEdge. In addition, OpenEdge Release 11.4 includes some feature enhancements and other changes. Despite the differences between 10.2B and 11.4, the demos are still a useful introduction to Progress Developer Studio for OpenEdge.

For general information about both OpenEdge and Progress Developer Studio for OpenEdge, refer the OpenEdge Tour.

What is OpenEdge

Progress OpenEdge is a complete development platform for building dynamic multi-language applications for secure deployment across any platform, any mobile device, and any Cloud.

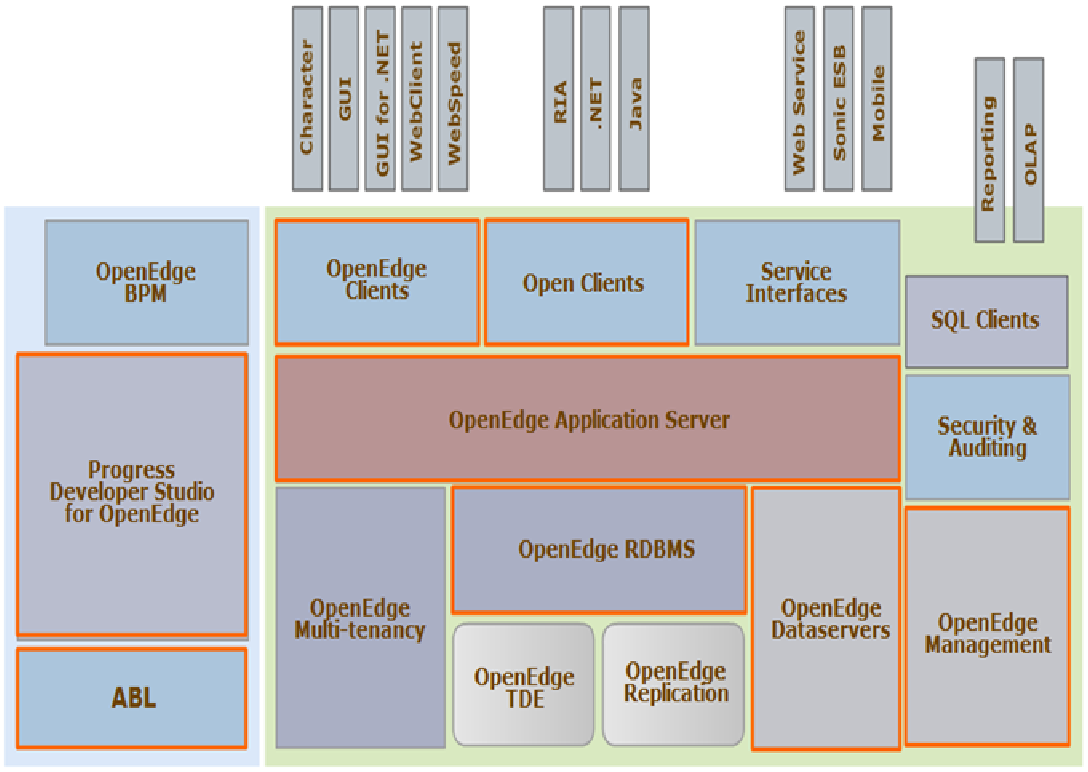
OpenEdge platform offers a complete, out-of-the-box solution for developing, integrating, and managing business applications offered as a service or on-premise.

With OpenEdge, software developers and partners can develop dynamic solutions that incorporate business process and integration capabilities securely across multiple platforms and devices. With OpenEdge, service and application development is agile and cost-effective, and the resultant applications are reliable, easy-to-maintain, cost-effective, and service enabled. This results in companies being able to capitalize on new opportunities by getting competitive applications to market faster.

A comprehensive view of the components in OpenEdge

The following topic provides a comprehensive overview of the OpenEdge product components and features that support application development and deployment in an integrated environment.

The following image illustrates a view of the components and features in OpenEdge:



These products include:

- Progress Developer Studio for OpenEdge
- OpenEdge RDBMS
  - OpenEdge SQL
- OpenEdge DataServers
- Clients
  - ABL clients
  - Non-ABL clients
  - REST and SOAP Web service clients

    - WebSpeed client
    - SQL client
- OpenEdge AppServer
- OpenEdge Management and Explorer
- OpenEdge Business Process Management

The features include:

- ABL Advanced Business Language (ABL)
- OpenEdge Multi-Tenancy
- OpenEdge Table-Partitioning
- OpenEdge REST Web Services
- OpenEdge SOAP Web Services

Related Links

- Advanced Business Language (ABL)
- The ABL Virtual Machine (AVM)
- Progress Developer Studio for OpenEdge
- OpenEdge RDBMS
- OpenEdge DataServers
- OpenEdge Management and OpenEdge Explorer
- Clients
- OpenEdge AppServer
- Progress Application Server
- OpenEdge Web Services
- OpenEdge Business Process Management

Advanced Business Language (ABL)

ABL (Advanced Business Language) is a high-level procedural and object-oriented programming language developed to enable you to build almost all aspects of an enterprise business application, from the user interface to the database access and business logic. ABL is a versatile and extraordinarily powerful tool. Not only can you use it to program applications, but you can build many of the tools that you use to help create and support those applications.

ABL includes powerful statements and keywords that are specialized for building business applications. Single programming statements in ABL can do the work of dozens or possibly hundreds of lines of code in a standard 3rd Generation Language, such as Visual Basic, Java, or C++. For example, a single ABL statement can access both existing data in an application database and apply changes to that data in the database. A statement separate from the data access and business logic can bind data to user interface controls. Other statements let you program with great precision, even down to the level of extracting individual bits from a data stream. This flexibility is what gives ABL its great power as a development language. Most of the development tools you use to develop OpenEdge applications are themselves written in ABL.

The ABL Virtual Machine (AVM)

The AVM is a platform for compiling and running ABL code. Often called the OpenEdge runtime or client, the AVM also provides many other services, such as performing syntax checks, running OpenEdge and user-designed tools, and running startup procedures. In Progress Developer Studio for OpenEdge, every OpenEdge project is associated with an AVM.

Note: The AVM process is prowin32.exe for Windows applications and _progres.exe for TTY applications.

Project-specific AVMs You might want to configure individual AVMs when projects in a workspace do not have the same requirements.You could, for example, create projects with different PROPATH settings by configuring an individual AVM for each project. Configure project-level AVMs on the properties pages for OpenEdge projects (**Project > Properties > Progress OpenEdge**).

Shared AVMs You can configure an AVM that is available to every project within a workspace. If, for example, you are developing an application that is divided into multiple project modules within a single workspace, you might want all the project modules to use an AVM that has the same startup parameters, the same PROPATH, the same database connections and so on. Configuring a shared AVM for all the project modules guarantees that all those settings are the same. In addition, the shared AVM conserves memory and CPU resources. You configure a shared AVM as a workspace preference (**Window > Preferences > Progress OpenEdge > Shared AVM**).

Runtime AVMs By default, when you run an ABL executable in a project, the AVM associated with the project (project-specific or shared) runs the ABL executable. However, you can also create launch configurations that start a new instance of the AVM when you run the file. Configure separate runtime AVMs when you create launch configurations in your workspace (**Run > Run Configurations...**).

Progress Developer Studio for OpenEdge

Progress Developer Studio for OpenEdge® is an integrated ABL development environment that helps you to:

- Quickly build user-interface and business logic
- Incorporate and manage application data sources
- Run, debug, and deploy code

Progress Developer Studio for OpenEdge consists of a set of editors, views, dialogs, and wizards that run in the Eclipse framework. Because Eclipse is an open-source, plug-in based framework, you can integrate a wide variety of software tools into the Eclipse installation that supports Progress Developer Studio for OpenEdge. See the Eclipse Plugin Central Web site for more information.

OpenEdge RDBMS

The Progress OpenEdge Relational Database Management System (RDBMS) is an established, reliable database management system for production systems world-wide, known for its scalability and ease of management.

The OpenEdge database can handle thousands of users and high-traffic loads with sub-second response times. It helps to ensure that users are productive, customers are satisfied, and, for SaaS/Cloud computing environments, service-level agreements are met.

Some of the key benefits of OpenEdge RDBMS include cost-effectiveness, raises productivity, easy to administer, scalable, and easy to manage.

OpenEdge DataServers

OpenEdge DataServers provide Advanced Business Language (ABL) applications with a flexible, transparent interface that enables simultaneous access to multiple data sources. This enables organizations to allow access to diverse data sources throughout the enterprise. These applications use ABL specifically designed to translate knowledge about business operations into software.

OpenEdge DataServers combine the rapid application development benefits of the OpenEdge component-based development environment with the high performance of today's powerful database engines, such as Oracle, Microsoft SQL Server, and ODBC.

OpenEdge Management and OpenEdge Explorer

OpenEdge Management is a system management center that provides visibility, analysis, and proactive monitoring of critical information assets. OpenEdge Management optimizes the availability and performance of OpenEdge-based applications through system monitoring, alerting, and automatic handling of corrective actions. OpenEdge Management empowers Progress Software customers to become more efficient, decrease the cost of managing the OpenEdge environment, and ensure high availability and performance.

OpenEdge Explorer is a subset of OpenEdge Management. OpenEdge Explorer runs within the OpenEdge Management console, which runs in a Web browser. Using OpenEdge Explorer, you can set resource configuration properties, start or stop, and view the status of log files for various OpenEdge resources. You can use OpenEdge Explorer without OpenEdge Management, or with OpenEdge Management if you have OpenEdge Management installed.

Unlike Progress Explorer, which runs only in Windows, OpenEdge Explorer is supported on all the UNIX platforms that support OpenEdge.

Clients

ABL Clients

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

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

Open Clients

An Open Client runs in a client process that is not an AVM. Open Clients are written in C#, VB.NET, or Java, and they can present a variety of user interfaces to the end-user. They can also provide functionality to other applications that do not have a user interface. Basically, in OpenEdge, .NET or Java AppServer clients and OpenEdge Web services share common features that make them all Open Clients of the AppServer.

There are two possible models for Open Client development:

- Using proxies
- Using OpenAPI

When an Open Client is developed using proxies, .class files (Java) or assemblies (.NET) are created using OpenEdge tools. The proxies are used to communicate services from the AppServer to the Open Clients. The proxies help the developer quickly write code to access the AppServer because the details are implemented in the proxies. When you use proxies, you access the AppServer Internet Adapter using HTTP or HTTPS, or you access the AppServer directly, depending on how the proxies are generated.

REST and SOAP Web service clients

Clients written in a variety of languages (ABL, Java, C#, VB.NET, HTML) can access an ABL application's services if the application has been built to provide Web services. An ABL Web services client runs in its own AVM. Other Web services clients run in client processes that run in a DLL or JVM.

On the server side, the application developer must generate (using OpenEdge tools) the artifacts required for the AppServer to provide the services. The Web Server receives requests from Web services clients and forwards the requests to the Web Services Adapter, which is part of the OpenEdge run-time environment. The Web Services Adapter then forwards the request to the relevant AppServer.

WebSpeed client

A WebSpeed client runs in a Web browser and is written in HTML and/or JavaScript. It accesses services using a Web Server by using HTTP or HTTPS. The WebSpeed Messenger's role is to receive requests from WebSpeed clients and forward them to a WebSpeed Transaction Server. The WebSpeed Transaction Server then runs a variation of ABL called SpeedScript. SpeedScript is used to provide services to clients and to present HTML pages back to the Web Server for display by the clients. In addition, WebSpeed Messenger can be used as a simple agent that can receive requests and display static HTML pages from a Web Server. To learn more about WebSpeed Transaction Server or SpeedScript, refer to the OpenEdge documentation.

SQL client

An SQL client is a process that can run an SQL engine for interpreting SQL statements and using them to access a Database Server. SQL clients can run in different types of processes such as AVMs, DLLs, and JVMs. SQL clients access the Database Server directly without going through an AppServer. Access to the Database Server is accomplished using JDBC or ODBC drivers. SQL clients are typically used to provide database management and reporting functionality to administrators and specific business users. They are not used to provide any business logic for your application.

OpenEdge AppServer

The OpenEdge AppServer is the core of OpenEdge application and integration services and is the standards-based transaction engine for running ABL business logic that can be made available to application clients as application services. Essentially, the AppServer is an ABL runtime client with no user interface, but instead provides a means for client applications to call its ABL procedures, user-defined functions, and class methods remotely.

By partitioning applications and separating the business-processing logic from the user interface logic, the AppServer allows you to access applications through virtually any interface. Centralized business logic then improves productivity by providing you with a single point to manage access to data and processes. Since the business logic executes separately from the application user interface, a wide variety of clients can be used to scale up to support aggressive growth strategies.

The AppServer is often used together with the OpenEdge NameServer to provide connection and server-level fault tolerance and facilitate application service availability. With the help of additional server products and adapters, the AppServer can make its application services available to all types of OpenEdge clients in many different configurations.

Progress Application Server

The Progress Application Server (PAS) is a platform that provides Web server support for Progress applications. Progress applications are packaged as Web application archives (WAR files) and deployed to the Java Servlet Container of a running instance of PAS. Client access to a PAS server is through HTTP/HTTPS protocols. Clients include (but are not limited to) mobile apps and browser-based web apps.

The foundation of PAS is Apache Tomcat (see http://tomcat.apache.org/).

Apache Tomcat is a Web server that includes a Java servlet container for hosting Web applications. The Apache Tomcat that you can download from the Apache Software Foundation is tailored primarily as a development server for testing, validating and debugging Web applications. PAS is tailored primarily as a production server for deploying Progress web applications.

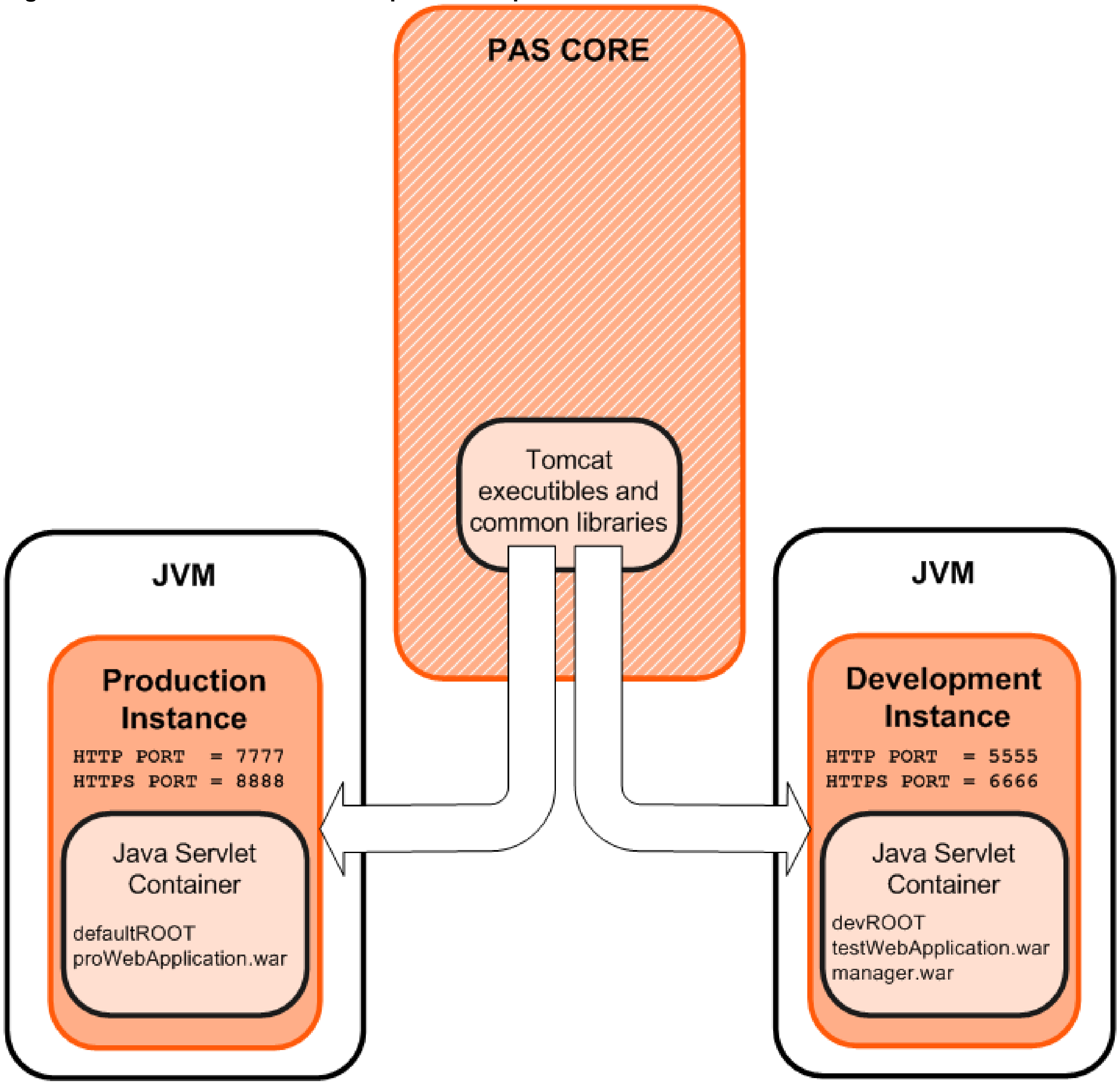
PAS tailors Apache Tomcat as a production server by replacing the default Tomcat ROOT Web application with a ROOT application that:

- Returns no content if a client accesses the root URL of the server, so that information about server configuration and deployed Web applications is not accessible
- Adds an implementation of the Spring Security framework to support user authentication and authorization
- Removes functionality, like remote administration, that could present a security risk

However, a key feature of PAS is that you can easily create and run more than one instance of the core server, and configure the instances separately to function either in a development or in a production environment. Each instance of the core PAS shares the executables and libraries of a common Tomcat server, but each instance is a separate process, running in a separate JVM, with its own configuration (ports, security framework, Web applications, etc.).

The following figure illustrates a common core PAS that supports a development server for testing, and a production server for publishing Progress Web applications.

Figure 1. The Core PAS with development and production instances



The figure shows that:

- Both the production instance and the development instance start up by invoking the same Tomcat runtime.
- Although it is possible to run the PAS core as a Web server, it usually does not run in a deployment with multiple instances. As shown here, the core is merely the source for the Tomcat executables and libraries that are common to all instances. (Among other advantages, this arrangement allows you to upgrade the core PAS without having to redeploy all of your Web applications.)
- Each instance is a unique process, running in its own JVM.
- Each instance has its own configuration, including uniquely defined ports.

- Each instance supports its own set of Web applications.

Note that each instance has a ROOT Web application tailored to its use as either a development or as a production server. Each ROOT application is also tailored to support whatever combination of Progress products that you intend to deploy. Also notice that the Development server has a manager.war application that is missing from the Production server. The manager.war application supports a variety of management and configuration utilities that you would not want to implement in a production sever.

Note: The ability to create multiple instances derived from a common core server is a powerful feature of the PAS architecture. Not only can you create and configure multiple server types, you can also create multiple servers to support load balancing. In addition, instances allow you to update the core Tomcat server without having to update or re-deploy Web applications.

OpenEdge Web Services

A Web service is an application that can be accessed and used over the Internet (or an intranet) using industry standard protocols.

In OpenEdge, a Web service is usually an AppServer application that is accessible to a client application through a Web server. Web services built with OpenEdge permit language and platform-independent access to OpenEdge business logic from non-OpenEdge clients, with the passing of data between the client and Progress ABL Web service application being largely transparent to the client language or platform.

In OpenEdge, you can:

- Create new Web services that you build as ABL (Advanced Business Language) applications and deploy on an AppServer.
- Expose existing AppServer applications as Web services.
- Create an interface to your Web services and deploy it on a Web server.
- Create the client-side applications that interact with your Web services.

OpenEdge includes support for Web services that are based either on SOAP, or on REST.

SOAP (Simple Object Access Protocol) uses XML messages to exchange information between client-side applications and Web services. WSDL (Web Service Definition Language) files publicize and describe the public interface to Web services. The XML Schema standard describes both simple and complex data types in WSDL files. SOAP uses HTTP only as a transport protocol. In OpenEdge, you deploy SOAP Web services on a Web Services Adapter (WSA) which runs in a Java Servlet Container on a Web server.

REST (Representation State Transfer) uses HTTP more extensively than SOAP. REST uses URIs to identify resources and employs HTTP verbs (GET, PUT, POST, DELETE) as methods to act upon resources. In OpenEdge, JSON is used as the data-interchange format. In OpenEdge, you deploy REST Web services in Web Application Archive files which run in a Java Servlet Container on a Web server.

OpenEdge Business Process Management

OpenEdge Business Process Management (BPM) provides an integrated authoring and runtime environment for developing, simulating and deploying process-oriented business applications. Using a drag-and-drop, BPM-compliant authoring tool, business analysts and OpenEdge application developers can create business process applications that span multiple worksteps, contain branching logic and support the integration of task-oriented, human interaction and integration requirements. It then organizes process tasks by individual or groups of users, with the ability to present the current list of tasks that require action.

Finally, flexible dashboard capabilities allow for monitoring execution of processes, to ensure that the business operations run smoothly but, just as importantly, to help identify areas for process optimization and improvement.

OpenEdge client-server process architecture

During runtime, applications deployed in an OpenEdge process architecture always include one or more client processes, and one or more server processes. This makes it very important for all developers to understand what OpenEdge processes are, and how the code uses these processes.

While developing applications, you must ensure that certain OpenEdge processes are running. For example, to test application code for the business logic, a server process must be running.

Related Links

- OpenEdge Servers
- OpenEdge Client processes

OpenEdge Servers

There are many types of server processes in an OpenEdge run-time environment. A typical OpenEdge installation comprises a number of components that represent several server processes at runtime. The following list below describes the default behavior of these processes.

- **OpenEdge AdminServer:** AdminServer is a process that is used to manage the startup of other server processes. It is also used to maintain the configuration of all other server processes. Use OpenEdge Explorer, a Web-based application of the AdminServer, to view, modify, and manage processes.
- **OpenEdge AppServer Process:** The AppServer process is a process that is used to execute ABL for the business logic or the business data part of an application. An AppServer consists of a broker process and one or more agent processes. The AppServer is used to service client requests from a number of different types of clients. The agent runs a specialized process that executes and compiles ABL called an ABL Virtual Machine (AVM).
- **WebSpeed Transaction Server:** The WebSpeed Transaction Server executes WebSpeed application code. It is used to service requests from HTML clients for dynamic Web pages. The WebSpeed Transaction Server's agent is typically a client of an AppServer that provides the data for the Web pages. A WebSpeed Transaction Server consists of a broker process and one or more agent processes. The agent runs a specialized process that executes and compiles ABL called an ABL Virtual Machine (AVM).
- **Database Server process:** The Database Server process services requests by the business data part of the application for data and transaction control for an OpenEdge database. An OpenEdge Server consists of a broker and one or more servers.
- **NameServer:** A NameServer is a process that is used to look up and register processes at run time. It is typically started before a client can communicate with an AppServer, WebSpeed Transaction Server, or Database Server.

- **DataServer process:** A DataServer process enables an ABL client to access a foreign database (non-OpenEdge database).
- **OpenEdge Adapter process:** An OpenEdge Adapter process is used to pass service requests to the AppServer from non- ABL clients that cannot communicate directly with the AppServer.

OpenEdge Client processes

OpenEdge client processes are used to implement the functionality required by a direct end-user (with a user interface) or an indirect end-user (embedded application). OpenEdge client processes communicate with server processes that provide business logic or business data to a client. ABL clients communicate directly with an OpenEdge Server process. Some non-ABL clients can communicate with the OpenEdge Server process directly, but some non-ABL clients must communicate through an OpenEdge Adapter process, which in turn communicates with an OpenEdge Server process. WebSpeed clients and other non-OpenEdge clients (Web and mobile apps) communicate through a Web server process.

2

Understanding OpenEdge Reference Architecture

Related Links

- An Overview of OpenEdge high-level application architecture
- OpenEdge Reference Architecture design principles and its usage

An Overview of OpenEdge high-level application architecture

Application architecture defines how your application code is structured and how the different parts of the application communicate. You should define your application architecture so that it is easy to understand, especially for developers who collaborate with you to develop or maintain the application.

When you design an application, the best practice is to separate your application into logical units for distribution across multiple physical computers. At a minimum, the units include the client side of the application, the business logic, and business data.

The client provides the primary interaction point for the user of an application and delivers information using any of a number of appropriate user interface technologies (graphical, web/browser, PDA, etc). This business logic provides for the processing of information for the application. This includes application logic for business rules and complex analytical processing.

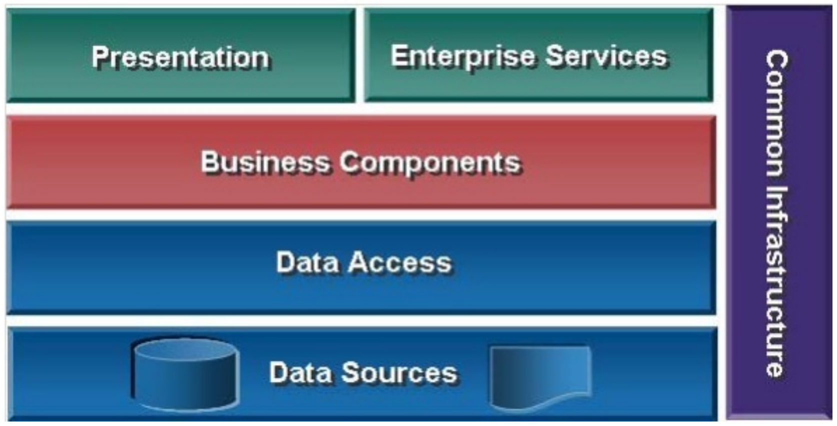
The separation into units enables you to reduce maximize code reuse. For example, if an algorithm change affects how your business logic executes, you do not have to modify code for the client or business data areas of your application. The logical partitioning of the user interface and application logic in a distributed architecture also allows for the physical separation of the application across multiple computers.

In the OpenEdge application architecture, you separate your application into functional components (user-client, business logic and business data). This recommended application architecture is called the OpenEdge Reference Architecture (OERA).

OpenEdge Reference Architecture design principles and its usage

The OpenEdge Reference Architecture (OERA) is intended to provide high-level design guidance focused on building modern competitive applications. OERA defines the general functional categories of components that comprise an application. OERA is generally used as a high-level blueprint for developing OpenEdge service-oriented business applications.

The following diagram shows the high-level elements of the Architecture:



Each layer of OERA consists of distinct components that have specific characteristics, roles, and responsibilities. In addition, OERA provides guidelines about how each of the architectural components interact. The Reference architecture makes it possible for application developers to solve business problems efficiently, effectively, and flexibly.

- The Data Access Layers manage the retrieval of data from either Managed Data Stores such as the OpenEdge RDBMS, or Unmanaged Data Stores such as flat files and XML documents, and the writing of data updates back to those data stores.
- The Business Components Layer holds the business logic that is central to the application’s behavior and value, along with application elements that provide general services to the application, such as security, auditing, and translation.
- The Presentation Layer manages the user interface and the communicating of data and logic between the application’s users and the business logic.
- The Enterprise Services Layer manages communication to other applications and services outside the bounds of the target application, to allow the application to be part of a wider enterprise.

Related Links

- Recommended order for an OERA design

Recommended order for an OERA design

As an architect or developer, you must determine how much of your application will follow the OERA design principles recommended by Progress Software. Start by first gathering the application requirements. If there are existing parts of the application, determine if you will retrofit them to conform to OERA. In order to partition your application into usable Business Components, document the use cases of the application.

Ideally, you would want the use cases to drive the Business Component design, and use ABL's strengths in supporting business logic and data access to help with the design and implementation.

Follow the recommended order provided below for an application design that uses OERA layers.

1. Determine the types of data each use case requires. These types of data can be used to begin the definition of your ABL DataSets.
2. Determine the sources of your data for each ABL DataSet to define the data source layer.
3. Define the business entities, business tasks, and business work flows to support the use cases. Each business entity has its own data access layer implementation. Define the functionality required for each business component. Define the key services or entry points each business component will provide.
4. Define the service interface(s) for every business component's service or entry point. Understand who the requesters will be as that will affect the types for the parameters used. Add discovery, authorization, and authentication to the logic of the service interface, as required.
5. Define service adapters based on the service interfaces. You have to understand which types of clients will access the business components. Ensure that the service adapters are written to locate the service they are interested in.
6. Define the client business entities used by the model for the user interface.
7. Define the model, view, and presenter components for the user interface.

3

An overview of ABL

Related Links

What is ABL?

Advanced Business Language (ABL) is a high-level procedural programming language that helps developers to develop applications optionally using its own integrated relational database and programming tool. These applications are portable across computing systems and allow access to various popular data sources without having to learn the underlying data access methods. This means that the end-user of these products can be unaware of the underlying architecture.

By combining a fourth generation language and relational database, ABL empowers application developers to manage relational data in a way that models how the business operates. A programmer and even end users can do rapid prototyping using the integrated and GUI tools of the development environment.

ABL is a versatile and extraordinarily powerful tool. Not only can you use it to program applications, but you can build many of the tools that you use to help create and support those applications. Most of the development tools you use to develop OpenEdge® applications are themselves written in ABL.

For more information on ABL, see the guide *OpenEdge Getting Started: ABL Essentials*.

Related Links

- Basic Characteristics of ABL

Basic Characteristics of ABL

- **ABL is a procedural programming language**: ABL provides a programmer a means to define precisely each step in the performance of a task. The programmer knows what is to be accomplished and provides through the language step-by-step instructions on how the task is to be done. Using ABL, a programmer specifies language statements to perform a sequence of algorithmic steps.
- **ABL is block-structured**: An ABL procedure is made up of blocks. The procedure itself is the main block of the procedure. There are multiple ways to define other blocks within the main procedure block. The FOR EACH statement and its matching END statement are one example of a nested block, in this case one that iterates through a set of database records and executes all the code in between for each record in the set. There are other block statements you can use for different purposes. Some of them are also iterating, and cause the block to be executed multiple times. Others simply define a set of statements to be executed together.
- **ABL procedures consist of statements**: ABL procedures are made up of a sequence of language statements. Each statement has one or more ABL keywords, along with other tokens such as database field names or variable names. A single statement can span multiple lines, and there can be multiple statements on a single line. ABL is case-insensitive.
- **ABL combines procedural, database, and user interface statements**: There are three basic kinds of statements in an ABL program: procedural statements, database access statements, and user interface statements. Sometimes individual statements contain elements of all three. Your first simple procedure contains all three types, and illustrates the power of the language.

For more information on characteristics of ABL, see the guide *OpenEdge Getting Started: ABL Essentials*.

Understanding ABL Syntax

The following sections describe some general features of ABL syntax. ABL is a block-structured, but statement-oriented language. That is, much of the behavior of an ABL application depends on how statements are organized into blocks. However, the basic executable unit of an ABL application is the statement.

Related Links

- Statements
- Comments
- Blocks
- Programming Models
- Compile-time Versus Run-time Code
- Procedures
- REPEAT, FOR, and DO Blocks

- Looping

Statements

This is the basic syntax of an ABL application: An ABL application consists of one or more statements. Each statement consists of a number of keywords, user-defined placeholders for values, symbols, and constants.

```
statement { . | : } [ statement { . | : } ...
```

The following statement is a one-statement application that displays “Hello, World!” in a message alert box:

```
MESSAGE "Hello, World!" VIEW-AS ALERT-BOX.
```

Comments

An ABL application can also contain non-executable comments wherever you can put white space (except in quoted strings). Each comment begins with “/*” and terminates with “*/”, and you can nest comments within other comments:

```
/*/* Simple Application */*/ MESSAGE /* Statement Keyword */ "Hello, World!" /*
String */ VIEW-AS ALERT-BOX /* Options */ . /*/* Period Terminates */ -^--------
^- Statement */
```

Blocks

In ABL, a block is a sequence of one or more statements, including any nested blocks, that share a single context. A context consists of certain resources that a block of statements share. The content of this shared context depends on the type of block and its relationship to other blocks. The sample procedure below shows a typical layout of blocks in a procedure:

```
/* BEGIN EXTERNAL PROCEDURE BLOCK */ DEFINE BUTTON bSum LABEL "Sum Customer
Balances". DEFINE VARIABLE balsum AS DECIMAL. DEFINE FRAME A bSum. ON CHOOSE OF
bSum IN FRAME A DO: /* BEGIN Trigger Block */ RUN SumBalances(OUTPUT balsum).
MESSAGE "Corporate Receivables Owed:" STRING(balsum, "$>,>>>,>>>,>>9.99") VIEW-
AS ALERT-BOX. END. /* END Trigger Block */ ENABLE bSum WITH FRAME A. WAIT-FOR
WINDOW-CLOSE OF FRAME A. PROCEDURE SumBalances: /* BEGIN Internal Procedure
Block */ DEFINE OUTPUT PARAMETER balance-sum AS DECIMAL INITIAL 0. FOR EACH
customer FIELD (Balance): /* BEGIN Iterative Block */ balance-sum = balance-sum
+ Balance. END. /* END Iterative Block */ END. /* END Internal Procedure Block
*/ /* END EXTERNAL PROCEDURE BLOCK */
```

A block within a procedure or class file is bracketed by an ABL keyword that ends with the ":' character, any number of ABL statements, followed by the "end." statement.

Programming Models

OpenEdge supports two programming models. You can determine which model best suits your development effort, before you start programming. These models are as follows:

- Procedure-driven programming
- Event-driven programming

Related Links

- Procedure-driven and Event-driven Programming
- Flow of Execution

Procedure-driven and Event-driven Programming

Procedure-driven programming, which you can think of as traditional programming, defines the programming process as the development of procedures that explicitly direct the flow of data and control.

Event-driven programming defines the programming process as the development of procedures that respond to the flow of data and control as directed by the user, program, or operating system.

These programming models differ in flow of execution and structure. In addition, each model works best with a particular programming environment.

Flow of Execution

The difference between the procedure-driven and event-driven models is primarily one of program flow. In a typical procedure-driven program, the execution of program statements is predetermined and controlled by the programmer. It generally runs from the beginning of the source file to the end. The programmer codes any branches or loops.

In a typical event-driven program, program execution is largely determined by the system. First the programmer sets up a series of actions (triggers) to be executed in response to specific events. Then the system waits for user input to initiate these events. Finally, the system responds to these events, thereby controlling program execution.

Compile-time Versus Run-time Code

Like many languages, the OpenEdge ABL includes two basic types of code:

- Compile time, sometimes known as non-executable code
- Run-time, sometimes known as executable code

However, as an interpretive language, ABL syntax combines compile-time and run-time components in many more ways than a compiled language like C. The flexibility of this syntax helps implement the rich variety of defaults that can be overridden that characterizes the ABL.

Compile-time Code: Certain statements exist only to generate r-code when OpenEdge compiles them. These are compile-time statements. That is, they create static data and user interface resources (widgets) that the run-time statements can reference and modify, but not destroy, during execution.

Run-time Code: Run-time statements use the static resources created by compile-time statements, but can also create, use, and destroy dynamic resources at run time. That is, run-time statements include statements that interact with static resources, dynamic resources, or both. Many run-time statements also include compile-time options. These options generate resources at compile time that are later used by the same statements at run time.

Procedures

The procedure is the fundamental block in Progress. All types of procedures, external and internal, have certain common properties:

- They can contain all other types of blocks.
- You can execute them by name using the RUN statement.
- They can accept run-time parameters for input or output.
- They can define their own data and user interface environment (context) with restrictions depending on the type of procedure.
- They can share data and widgets defined in the context of another procedure, with restrictions depending on the type of procedure.
- They can execute recursively. That is, they can run themselves in a new context.

There are two types of procedures: external and internal. An external procedure block consists of a text file containing zero or more statements that you can compile into a unit and that you can execute by name using the RUN statement. An external procedure is the most basic block and can contain all other types of statements and blocks.

An internal procedure block consists of a PROCEDURE statement followed by one or more statements terminated by an END statement that you can execute by name using a RUN statement. An internal procedure is always compiled as part of an external procedure. An internal procedure cannot contain other internal procedure blocks, but it can contain any other type of statement or block that an external procedure can contain.

Related Links

- User-defined Functions

User-defined Functions

OpenEdge ABL user-defined functions let your application define a logical rule or transformation once, then apply the rule or transformation with absolute consistency an unlimited number of times. For example, if you are developing a weather application that converts temperatures between Celsius (C) and Fahrenheit (F), you can write two user-defined functions, one for converting from C to F and one for converting from F to C. Then, whenever your application must convert a temperature in either direction, you merely reference one of the user-defined functions.

The definition of a user-defined function can reside in the procedure that references it, in a procedure external to the procedure that references it, or in a procedure remote to the procedure that references it. In other words, a user-defined function can be defined locally, externally, or remotely.

User-defined functions follow the rules of scope and visibility that internal procedures follow.

REPEAT, FOR, and DO Blocks

REPEAT, FOR, and DO blocks consist of a block header statement (REPEAT, FOR, or DO) followed by zero or more statements terminated by an END statement. Each of these blocks can contain any other type of block, except a procedure and trigger block. REPEAT, FOR, and DO blocks execute wherever they appear in a procedure or trigger block.

Looping

Looping, or iteration, is one of the automatic processing services that OpenEdge provides for blocks. REPEAT blocks and FOR EACH blocks iterate automatically. The following code fragments show iterations using these block statements:

```
/* This REPEAT block loops through its statements until the user presses the
END-ERROR (F4) key. */ REPEAT: PROMPT-FOR customer.cust-num. FIND customer USING
cust-num. DISPLAY customer WITH 2 COLUMNS. END.
```

```
/* This FOR EACH block reads the records from the customer table one at a time,
processing the statements in the block for each of those records. */ FOR EACH
customer: DISPLAY customer WITH 2 COLUMNS. END.
```

```
/* This block loops through its statements for the first 5 customers. */ DEFINE
VARIABLE i AS INTEGER. DO i = 1 TO 5: FIND NEXT customer. DISPLAY customer WITH
2 COLUMNS. END.
```

Object-oriented programming with ABL

Related Links

- Understanding Object-oriented programming within ABL
- Advantages of Object-oriented programming within ABL

Understanding Object-oriented programming within ABL

Object-oriented programming is a programming model organized around objects rather than actions. Conventional procedural programming normally takes input data, processes it, and produces output data. The primary programming challenge is how to write the logic. Object-oriented programming focuses on the objects that you want to manipulate, their relationships, and the logic required to manipulate them. The fundamental advantage of object-oriented programming is that the data and the operations that manipulate the data are both encapsulated in the object with a well-defined interface.

OpenEdge provides language extensions to support these standard object-oriented programming concepts in a way that is commonly available in other object-oriented languages, such as Java. These object-oriented extensions complement the basic powers of ABL, and its procedural programming model, with an alternative programming model that can seamlessly coexist with applications written using the procedural programming model.

Object-oriented programming in OpenEdge has been designed to complement ABL, and is meant to be combined and integrated with 'traditional' procedures, when it makes sense to do so. The object-oriented enhancements extend the core values of ABL, not replace them.

For more information on object-oriented programming, see the guide *OpenEdge Development: Object-oriented Programming*.

Advantages of Object-oriented programming within ABL

The following lists advantages of using object-oriented programming within ABL:

- Helps manage complexity while increasing productivity
- Improves code re-use by making it simple. For example, take an object that implements a stack. By inheriting the object into another object (like an error minder), the stack object immediately becomes part of the error minder with the error minder “adding value” to the stack object with error code translation etc.
- Code re-use can be instantiated in parallel. You can simply create an instance of the same object as many times as needed, thereby avoiding copy-paste option.
- Can “mix and match” object-oriented programming and ABL strengths with Procedural strengths
- Makes it easier for programmers to create abstraction layers. Programmers can easily implement modularization right from the conceptualize to the implementation stage.
- Increases application robustness
- Increases development productivity with encapsulation, inheritance, and polymorphism
- Helps implement OpenEdge Reference Architecture (OERA).
- Closely supports Service-oriented architecture (SOA)
- Ease of application maintenance
- Natural integration with:
  - Modeling tools
  - Other Object-oriented platforms

Object-Oriented Concepts Overview

A few concepts that must be understood before working with object-oriented programming and ABL are classes, types, data members, methods and objects. All these concepts are closely related. It is important to understand the difference between these terms. The following list explains all the above important concepts.

- **Type:** A type is a name that identifies specific members of a class, which can include methods, properties, data members, and events.
- **Class:** A class defines the implementation of a type and its class members. An abstract class is essentially a type with an incomplete implementation. Classes in ABL support a basic set of object-oriented concepts.
- **Interface:** An interface also defines a type that identifies certain class members (properties, methods, or events) that a class must implement.
- **Data Members:** Data of a class is stored in data members. Each data member is defined by a name and type.
- **Object:** An object is an instance of a class whose type can be represented as any class or interface that contributes members defined in the object’s class hierarchy. Objects have a life cycle in which they can be repeatedly created, used, and destroyed during an ABL session.

For more information on Object-oriented programming, see the guide *OpenEdge Development: Object-oriented Programming*.

Related Links

- Encapsulation
- Inheritance

- Delegation
- Polymorphism
- Method Overloading
- Strong Typing

Encapsulation

Encapsulation is a way of organizing data and methods into a structure by concealing the way the object is implemented, thereby preventing access to data by unspecified means. Because of the goal to maintain the privacy of a class's implementation, encapsulation is also known as information hiding.

For more information on Object-oriented programming, see the guide *OpenEdge Development: Object-oriented Programming*.

Inheritance

Inheritance allows a given class to inherit data members, properties, methods, and events from another class —that is, it allows a given class to adopt members of another class in such a way that the given class appears to have defined these members within itself. It is used by programmers as a mechanism for code reuse. The relationships of classes through inheritance give rise to a class hierarchy.

If one class explicitly inherits from another, the inherited class is a super class of the class that inherits from it. Conversely, any class that inherits from a super class is a subclass (also known as a derived class) of the specified super class.

For more information on Object-oriented programming, see the guide *OpenEdge Development: Object-oriented Programming*.

Delegation

Delegation is the simple yet powerful concept of letting contained objects do their own work. It is used to describe the situation where one object assigns a task to another object, known as the delegate.

In ABL, Delegation uses composition to build a class from one or more other classes without relating the classes in a hierarchy. You can create your own container and delegate classes in ABL. You can also use interfaces to help enforce consistency between the method stubs implemented in a given container class and the effective method implementations defined in a corresponding delegate class.

For more information on Object-oriented programming, see the guide *OpenEdge Development: Object-oriented Programming*.

Polymorphism

Polymorphism is one of the most powerful advantages of object-oriented programming, and it relies fundamentally on inheritance and overriding. When a message is sent to an object, the object must have a method defined to respond to that message. In an inheritance hierarchy, all subclasses inherit the methods from their super class; each subclass however may require a separate response to the same message, as each subclass is a separate entity.

Polymorphism allows the message in the superclass to be dispatched to the method in the subclass at runtime. In other words, Polymorphism means that objects in a class hierarchy can respond to the same message in different manners. For example, when you have a system with many shapes, each shape—for example, a circle, a square, or a star—is drawn differently. By using polymorphism, you can send each of these shapes the same message (for example, Draw), and each shape would be responsible for drawing itself.

For more information on Object-oriented programming, see the guide *OpenEdge Development: Object-oriented Programming*.

Method Overloading

Methods with the same name in a class are called overloaded methods. Method overloading allows a class to define multiple methods with the same name, but different signatures. That is, it allows you to define different methods that have the same name, but that respond to correspondingly different messages sent to an instance of the class.

For more information on Object-oriented programming, see the guide *OpenEdge Development: Object-oriented Programming*.

Strong Typing

Strong Typing is the enforcement of type conformance. The object types defined by classes and interfaces are analogous to the built-in primitive types of the language. Object types are identified using an object type name that includes the name of the class or interface. For a user-defined class or interface, this name matches the file name of the file that stores the definition of the class or interface type.

Strong types defined by a programmer are enforced by the compiler for conformance to a specific Class definition. Among other benefits, this allows checking many possible errors at compile time instead of discovering them at run time.

For more information on Object-oriented programming, see the guide *OpenEdge Development: Object-oriented Programming*.

Understanding the ABL Class-based model and its members

Related Links

- Constructor
- Destructor
- Data Members
- Properties
- Events
- FINALLY End block Statement
- Traditional Error-handling

- Structured Error-handling

Constructor

Constructors are special methods that define initial behavior for a class, similar to internal procedures and user-defined functions in procedures.

A constructor allows you to execute behavior and initialize data members and properties during the instantiation of a class. ABL allows you to define a constructor as a named block that always begins with the `CONSTRUCTOR` statement and always ends with the `END CONSTRUCTOR` statement. A constructor must have the same name as the name of the class in which it is defined and it can only execute when the class is instantiated. A constructor can be defined with an access mode of `PRIVATE`, `PROTECTED`, or `PUBLIC`, where `PUBLIC` is the default.

A constructor can also have parameters, but no defined return type. You can define more than one constructor (overload constructors) in a class as long as the calling signature of each constructor is unique.

For more information on Object-oriented programming, see the guide *OpenEdge Development: Object-oriented Programming*.

Destructor

A destructor, which you can define as needed and which is automatically invoked during garbage collection or when you delete it by garbage collection, by a `DELETE OBJECT` statement, or halting a class instantiation by a `RETURN ERROR` or `UNDO`, `THROW`. Thus, you never call a destructor directly. You can provide a destructor when the class needs to free resources or do other cleanup work when a class instance is destroyed. If a destructor is not provided, ABL provides a default destructor in order to delete the object.

Unlike ordinary methods, a destructor definition is identified by the `DESTRUCTOR` statement. Destructors have no return type, are always PUBLIC, and cannot have any parameters. Destructors also are instance members. Note that there is no static destructor.

A class is not required to have a destructor. If a class has not defined a destructor for the class, ABL provides a default destructor for the class.

For each class instance that you delete, its destructor is responsible for deleting any resources allocated during the execution of that class instance. Deleting the class instance automatically deletes all dynamic handle-based objects that are created in any default unnamed widget pool that is specified for the class.

For more information on Object-oriented programming, see the guide *OpenEdge Development: Object-oriented Programming*.

Data Members

Data members define instance data of a class. Data Members must be defined in the main block of the class. Data members of a class are defined using the standard `ABL DEFINE` statements, with the addition of an optional access mode for each data member. The access mode is valid only for data member definitions in the main block of a class definition file (not for local method data, for example).

Data members in a class can specify an access mode, which tells ABL whether the data member should be visible only within the class (PRIVATE), visible to itself and other classes that inherit from it (PROTECTED), or visible to any other procedure or class in the same OpenEdge session (PUBLIC).

For more information on Object-oriented programming, see the guide *OpenEdge Development: Object-oriented Programming*.

Properties

Properties are class members similar to variable data members, but they can have behavior defined and associated with them. This behavior specifies if a property can be read or written, and includes any statements to be executed when the property is read or written. ABL allows you to define a property using a `DEFINE PROPERTY` statement, which includes the definition of any one or two special methods, each of which is referred to as an accessor.

A `GET` accessor indicates that the property is readable and includes optional statements to be executed when the property is read. A `SET` accessor indicates that the property is writable and includes optional statements to be executed when the property is written.

For more information on Object-oriented programming, see the guide *OpenEdge Development: Object-oriented Programming*.

Events

You can define events as class members that allow you to send (publish) notification of any run-time conditions that you detect. Your application can then respond to this notification by executing one or more methods and internal procedures (event handlers) that you specify (subscribe) for the event. ABL allows you to define a class event with a name and a signature using the DEFINE EVENT statement. The signature defines the parameters (if any) that any subscribed event handler must have.

For more information on Object-oriented programming, see the guide *OpenEdge Development: Object-oriented Programming*.

FINALLY End block Statement

The FINALLY End block statement defines a final end block for an ABL block. An end block is an ABL block that can occur only within another block. The block that contains the end block is known as an associated block. End blocks must occur between the last line of executable code in the associated block and the `END` statement. For any ABL block statement, the `FINALLY` statement defines an optional end block that contains ABL code to execute at the conclusion of all other processing in the associated block or once at the conclusion of each iteration of an associated iterating block.

For more information, see the guide *OpenEdge Development: ABL Reference*.

Traditional Error-handling

ABL has always been a procedural, block-structured, database language with the ability to mix procedural, database, and user-interface logic in a single procedure file of statements, blocks, and nested blocks. The semantics of working with a database means ABL error handling protects persistent data above all with tight integration to transaction management. Error handling draws on ABL procedural strengths to provide flow-of-control options that increase the chance of restoring application flow after an error.

ABL has expanded to provide object-oriented programming, distributed programming, and extensive integration features with other features of the OpenEdge platform. Consequently, there are many specialized error-handling topics. The traditional error-handling model provides an easy-to-use set of features for general ABL programming. It also provides a standard way of communicating errors to ABL from integrated OpenEdge features. For example, a Web service returns errors as industry standard SOAP faults. The traditional error-handling model lets your application handle a returned SOAP fault as an ABL system error.

For more information, see the guide *OpenEdge Development: Error Handling*.

Structured Error-handling

Structured error-handling is an expansion of the original ABL error handling features, which are now referred to as traditional error handling features. While the new language features do provide you with a more robust error handling model, they coexist and inter-operate with the traditional error handling features. Existing code will continue to work as is, and code using the new model can work with this existing code.

Structured error-handling is a model used in many languages, but is typically associated with object-oriented languages. It is characterized by the following features:

- Represents all errors as objects (instances of a class)
- Allows you to define your own error types (classes)
- Allows you to explicitly raise (throw) an error
- Allows you to handle (catch) particular error types in a particular context (ABL block)
- Allows you to propagate (re-throw) errors from the current context (inner block) to the immediate outer context (outer block). In other words, when an error occurs in a called context, you can direct the calling context to handle the error
- Allows you to specify code that executes at the conclusion of some associated code, whether or not the associated code executed successfully (`FINALLY` block)

For more information, see the guide *OpenEdge Development: Error Handling*.

Accessing Data

Related Links

- Data Types
- Database Connections
- Transactions
- Temporary Tables (TEMP-TABLES)
- ProDataSets

Data Types

The data type of a database field, temp-table field, or program variable defines what kind of data the field or variable can store. To allow better interoperability between OpenEdge applications and external products, OpenEdge provides additional data types and new features for some existing data types in ABL (Advanced Business Language).

Note: For a list of all the ABL data types, see OpenEdge Development: ABL Reference.

ABL supports the following basic kinds of data types:

- ABL built-in primitive types, including mappings to corresponding .NET primitive types
- Object types, which include both ABL and supported .NET object types, including both built-in and user-defined class and interface types
- ABL handle-based objects
- ABL arrays, including one-dimensional arrays of ABL primitive types,
- ABL object types, or
- .NET object types

Database Connections

An OpenEdge application can access one or more OpenEdge or non-OpenEdge databases simultaneously. The databases can be located on different operating systems using different networking protocols. To access non-OpenEdge databases, you must use the appropriate DataServer, such as DataServer for Oracle. You must connect to a database before you can access it. There are four ways to connect to a database:

- As a command-line argument (connection parameter) when starting OpenEdge
- With the CONNECT statement (in the Procedure Editor or in an ABL—Advanced Business Language—procedure or class)
- With the OpenEdge Data Dictionary or other database administration tools
- Using the auto-connect feature

Transactions

A transaction is a unit of work that is either completed as a unit or undone as a unit. Proper transaction processing is critical to maintaining the integrity of your databases.

A transaction is a set of changes to the database, which the system either completes or discards, leaving no modification to the database. The terms physical transaction and commit unit refer to the same concept as the Progress transaction.

Example: While entering new customer records into your database, you have completed entering 98 records and are working on entering the 99th customer record, when your machine goes down. OpenEdge in such a scenario keeps the first 98 records in the database and discards the partial 99th record.

```
REPEAT: INSERT customer WITH 2 COLUMNS. END.
```

Each iteration of the REPEAT block above is a transaction. The transaction is undone (or backed out) if:

- The system goes down (or crashes).
- The user presses STOP (CTRL–BREAK on Windows; usually CTRL–C on UNIX).

In either of these cases, OpenEdge undoes all work it performed since the start of the transaction.

Temporary Tables (TEMP-TABLES)

An important ABL construct is the temporary table (temp-table). A temp-table gives you nearly all the features of a database table, and you can use a temp-table in your procedures almost anywhere you could reference a database table. However, temp-tables are not persistent. They are not stored anywhere permanently. They are also private to your own OpenEdge session, the data you define in them cannot be seen by any other users.

You can define sets of data using temp-tables that do not correspond to tables and fields stored in your application database, which gives you great flexibility in how you use them. You can also use a temp-table to pass a whole set of data from one procedure to another, or even from one OpenEdge session to another. Together, queries and temp-tables provide much of the basis for how data is passed from one application module to another. They are essential to creating distributed applications, with data and business logic on a server machine and many independent client sessions running the user interface of the application.

Temporary tables are database tables that OpenEdge stores in a temporary database. Unlike regular database tables, which are permanent and which multiple users can access simultaneously, temporary tables last only for the duration of the procedure that defines them (or for the duration of the OpenEdge session, if you make them GLOBAL), and allow only one user at a time to access them. Finally, temporary tables are private, visible only to the user (process) that creates them. In short, if you want to sort, search, and process data for a duration not longer than the OpenEdge session, use temporary tables.

OpenEdge stores temporary tables and temporary files in the same directory—by default, your current working directory. You can change this directory by using the Temporary Directory (–T) startup parameter. Temporary tables require less memory than large work tables.

Note: You cannot access temporary tables using SQL statements.

You define a temporary table by using the DEFINE TEMP–TABLE statement. For example, the following statement defines temporary table Temp–Cust that inherits the field definitions of table Customer, which must belong to a database that is connected.

```
DEFINE TEMP-TABLE Temp-Cust LIKE Customer.
```

ProDataSets

An ABL ProDataSet is an in-memory relational data object that can encapsulate one or more ABL temp-tables (sets of data rows) and any parent-child relationships among them. It also includes mechanisms to track changes, which facilitates synchronization with any data sources used to initialize it. In effect, a ProDataSet models a relational database, its entity relations, and its change state in memory. Because of the power and efficiency with which it can organize and communicate complex relational data, the ProDataSet is a primary mechanism for bundling and passing complex data within applications that conform to the OpenEdge Reference Architecture (OERA).

4

Configuring OpenEdge

In the following topics, you will learn about Progress OpenEdge Relational Database Management System (RDBMS) architecture components and how they work together to support a running database. You will also learn about the different environments in which your databases can operate, configure OpenEdge by setting up your ABL development environment, configuring the distributed environment using AppServer technology, and managing the traditional AppServer in OpenEdge Management or Explorer.

Related Links

- Understanding the RDBMS architecture
- Reviewing environment variables
- Using the Proenv utility
- Creating and configuring an OpenEdge database server
- Working with AppServers

Understanding the RDBMS architecture

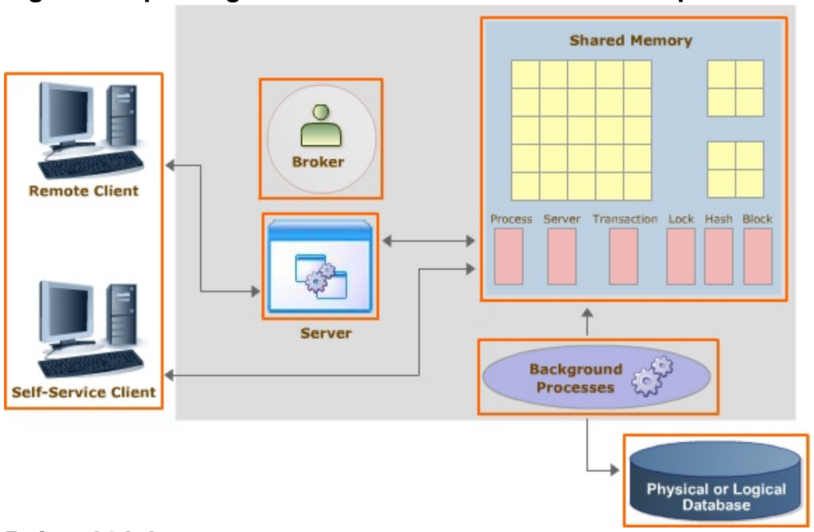
A relational database management system (RDBMS) is a database management system (DBMS) that is based on a relational model in which data is stored in the form of tables and the relationship among the data is also stored in the form of tables.

To effectively administer your databases, you must understand the OpenEdge RDBMS architecture. The OpenEdge RDBMS architecture consists of the following components:

- Logical and Physical Database
- Shared memory

- Broker
- Server
- Client
- Background processes

Figure 1. OpenEdge RDBMS architecture and its components



Related Links

- Logical Database
- Physical Database
- Broker
- Server
- Client
- Background processes

Logical Database

A Logical database helps you define and communicate your business' information requirements. Creating a logical data design is an information-gathering, iterative process. It includes the following steps:

- Define the tables you need based on the information your business requires.
- Determine the relationships between the tables.
- Determine the contents (or columns) of each table.
- Normalize the tables to at least the third normal form. To understand more about normal forms, refer the topics on **Normalization** in the *OpenEdge Getting Started: Database Essentials* guide.
- Determine the primary keys and the column domain. A domain is the set of valid values for each column. For example, the domain for the customer number can include all positive numbers.

Physical Database

A physical database is a refinement of the logical database design. In this phase, you examine how the user will access the database, and refine the database design by answering questions like:

- What data will I commonly use?
- Which columns in the table should I index based on data access?
- Where should I build in flexibility and allow for growth?
- Should I denormalize the database to improve performance? A database is typically denormalized at this stage to meet performance requirements.

Broker

A broker is the master database process that instantiates and manages the shared memory area of a multi-user database. It performs various tasks in a running database. A broker coordinates all the database connection requests, and servers retrieve and store data on behalf of the clients. The broker process locks the database to prevent any other broker or single-user process from opening it.

Server

Servers are database processes that access the database through shared memory on behalf of one or more remote clients. The broker launches them, as needed, to handle incoming remote clients. ABL servers are the servers, which support remote OpenEdge Advanced Business Language (ABL) clients only.

Client

A client process could be an ABL client or another type of OpenEdge client such as an Open Client. Clients access the business logic of an application by requesting a service from a Business Entity (or Business Task or Business Workflow). Clients can query, add, delete, or modify data in a database. There are two types of clients:

- **Remote (Network):** Remote clients can either be local or remote, but it cannot connect to a database directly, so it must use a server. Network clients access the database through a server process that the broker starts over a network connection. The network or remote client does not have access to shared memory, and it must communicate with a server process.

  Note: OpenEdge RDBMS treats SQL clients as remote clients even if they reside on the same machine as the database.

- **Self-service:** Self-service clients reside on the same machine as the broker, accesses the database directly through shared memory, perform both client and server functions in one process and execute application logic directly.

Background processes

Background processes are database processes running in the background. The processes enhance database performance and maintain data integrity in the event of a system crash or media failure.

There are four background processes:

- Before-image Writer (BIW): Writes full BI buffers to the BI files on disk.
- After-image Writer (AIW): Writes full AI buffers to the AI files on disk.
- Asynchronous Page Writer (APW): Writes modified database blocks to disk.
- Watchdog (PROWDOG): Cleans up after improperly terminated self-service clients as well as the remote clients of lost servers.

Reviewing environment variables

By default, the OpenEdge installation program tailors all the necessary OpenEdge and Java environment variables to the directories where they are installed. For example, the installation automatically sets the %DLC% environment variable to your OpenEdge installation path.

This section briefly reviews some system and Java environment variable details of which you should be aware.

Related Links

- System environment variables
- Java environment variables

System environment variables

The %DLC% environment variable is not set at the system level and should not be changed. After installing OpenEdge, however, you can set environment variables to suit your own preferences. You can use Proenv to set the %DLC% environment variable to the directory where OpenEdge is installed.

CAUTION: Although editing environment variables is an option, this procedure is not recommended if more than one version of an OpenEdge product exists on the same system.

For more information on environment variables, see the information on maintaining user environments in *OpenEdge Deployment: Managing ABL Applications*, or see your specific product documentation.

Related Links

- Latest information updates

Latest information updates

Before you continue, consult *OpenEdge Release Notes*. These notes contain the latest information about the current release that the OpenEdge documentation set might not include. Progress Software Corporation ships release notes in Microsoft Write (readme.wri) format. Click the **Release Notes** icon in your OpenEdge program group, or access Readme.pro with any text editor.

Java environment variables

OpenEdge bundles the Java Runtime Environment (JRE) component and the Java Development Kit (JDK) component with certain products that you install.

Note: OpenEdge supports Java version 7.0. For specific information about these components, see the *OpenEdge 10 Platform & ProductAvailability Guide* on the Progress Software Corporation Web site http://www.progress.com/products/lifecycle/index.ssp.

Related Links

- JDKHOME

JDKHOME

Java is used by some products, such as WebSpeed, the AppServer, and SQL, for product functionality. After you install any of these products, you should verify that the JDKHOME value is set correctly in the registry. The value must be set to the directory where the JDK included in the OpenEdge installation resides.

You can verify the JDKHOME value in the following location in the registry:

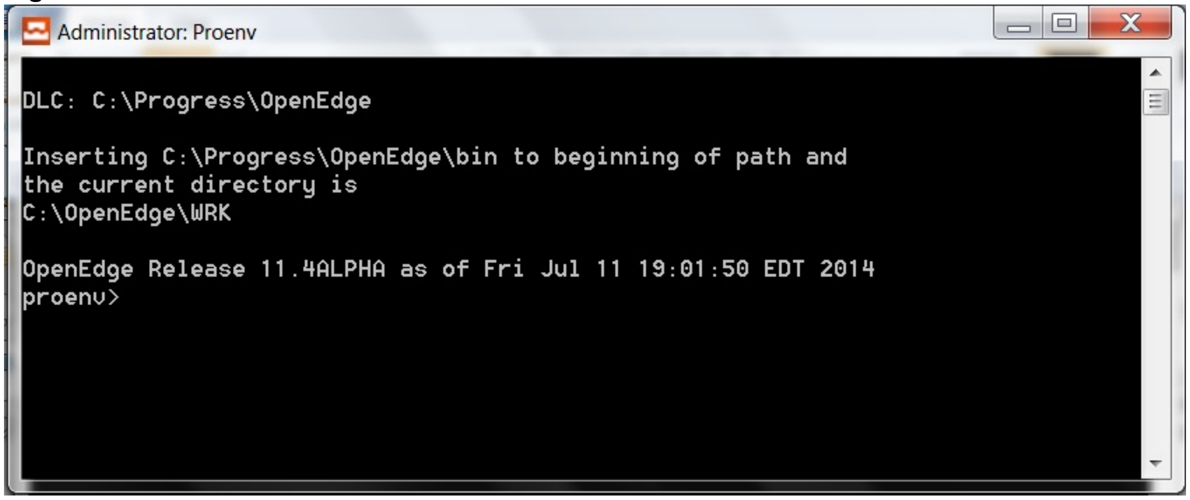
```
HKEY_LOCAL_MACHINE\SOFTWARE\PSC\PROGRESS\version11.7\JAVA
```

Using the Proenv utility

You can use the OpenEdge Proenv utility to perform certain OpenEdge Management actions from the command line.

You access the Proenv utility by choosing **Start > Programs (or All Programs) > Progress > OpenEdge > Proenv**. The environment window sets the shell environment variables needed for executing both OpenEdge Management and OpenEdge commands, as shown in the following figure.

Figure 1. Proenv window



```
DLC: C:\Progress\OpenEdge

Inserting C:\Progress\OpenEdge\bin to beginning of path and
the current directory is
C:\OpenEdge\WRK

OpenEdge Release 11.4ALPHA as of Fri Jul 11 19:01:50 EDT 2014
proenv>
```

Creating and configuring an OpenEdge database server

When you create an OpenEdge database, you can either create a new database or convert an existing database:

- Use the PRODB command, the Data Administration Tool, or the Data Dictionary to create a new OpenEdge database.
- Convert an existing OpenEdge or Progress database to OpenEdge.

For more information on creating databases, see *OpenEdge Database Management: Database Administration*.

You can also create and configure an OpenEdge database server in Windows.

To use OpenEdge features with a server:

1. From the desktop, choose **Start > All Programs > Control Panel > Administrative Tools > Services**. Verify that the status for the **AdminService for OpenEdge Release** is **Started**.

   Note: If **Administrative Tools** is not available, right click from the Task Bar. Choose **Properties**, then select the **Advanced** tab. Select the **Display Administrative Tools** check box, then choose **OK**.

2. Use OpenEdge Management or OpenEdge Explorer Tool to add a database configuration. (You cannot create the physical database with OpenEdge Explorer Tool.)

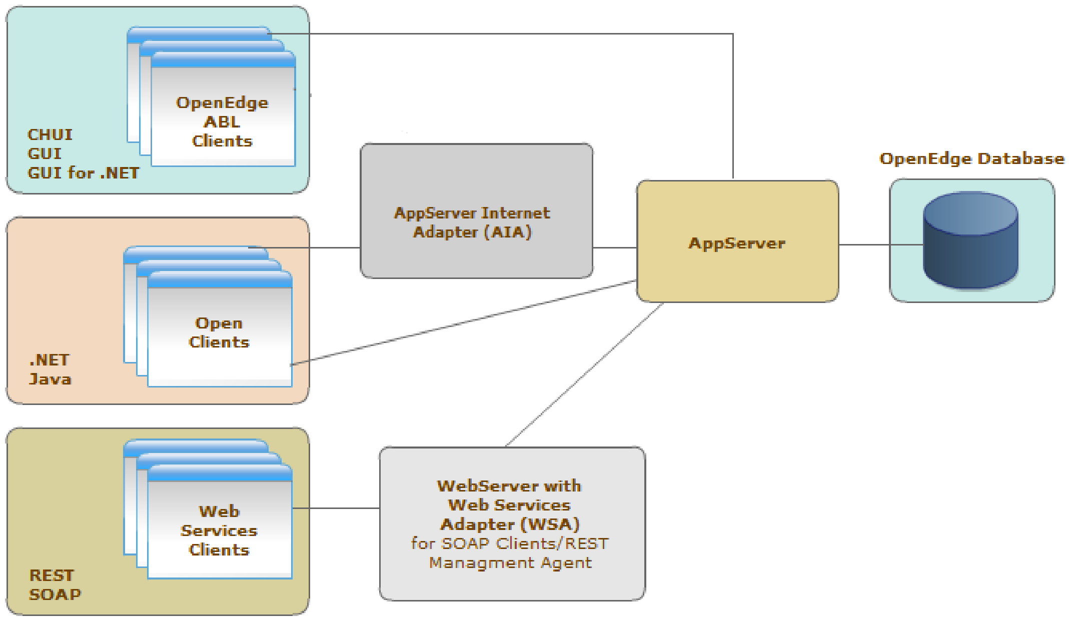
Note: For more information on the AdminServer and the OpenEdge Management or OpenEdge Explorer, see the *OpenEdge Management or OpenEdge Explorer online help*.

Working with AppServers

The OpenEdge AppServer is a standards-based transaction engine that provides a reliable foundation for high-volume secure transactions, and user interface independence.

By partitioning applications and separating the business-processing logic from the user interface logic, you can access applications through virtually any interface. Centralized business logic then improves productivity by providing you with a single point to manage access to data and processes. Since the business logic executes separately from the application user interface, a wide variety of clients can be used to scale up to support aggressive growth strategies.

Figure 1. OpenEdge AppServer Technology



A few key benefits of the OpenEdge AppServer include:

- Provides direct access to the same business logic from a wide variety of OpenEdge and common industry technology, giving you many options for how your application interacts with your end users.
- Decouples the physical hardware location from your application, providing flexibility in setting up and maintaining your operating environment.
- Supports asynchronous processing for faster application response time.
- Ensures reliability by offering load balancing and failover Reduces hardware requirements by providing state management.

5

How to get started with Progress Developer Studio for OpenEdge

Getting started with Progress Developer Studio for OpenEdge requires basic knowledge of the Eclipse framework. At the minimum, you should understand the following concepts that are explained in the *Eclipse Workbench User Guide*:

- Projects
- Views
- Perspectives

Also see the Basic Tutorial in the *Eclipse Workbench User Guide* for more information about working in the Eclipse framework.

Related Links

- Starting Progress Developer Studio for OpenEdge
- Working with OpenEdge perspectives
- Set workspace preferences

Starting Progress Developer Studio for OpenEdge

🎥	**Demos** Get started with Developer Studio - Part 1

	Get started with Developer Studio - Part 2 **Note:** The OpenEdge Release 10.2B demos have not been updated for OpenEdge Release 11.0. Since 10.2B, OpenEdge Architect has been renamed as Progress Developer Studio for OpenEdge. In addition, OpenEdge Release 11.0 includes some feature enhancements and other changes. Despite the differences between 10.2B and 11.0, the demos are still a useful introduction to Progress Developer Studio for OpenEdge.

Start Progress Developer Studio for OpenEdge by running the Eclipse executable (`OpenEdge-install-dir/DeveloperStudio3.6/eclipse/eclipse.exe`) with a startup parameter that points to the appropriate Java Runtime Environment (JRE). The default is `-vm OpenEdge-install-dir/DeveloperStudio3.6/jre/bin/javaw.exe`).

In Windows, you can start Progress Developer Studio for OpenEdge from the Start menu by selecting one of the Developer Studio items under the **Progress** node.

Note: Some of the Developer Studio items include the Clean (-clean) option. The -clean option in the startup command clears any cached data before starting the Eclipse executable. This is useful when restarting Eclipse after modifying any plugins. It ensures that you see the latest version of all files.

In general, the Progress Developer Studio for OpenEdge startup sequence is:

1. Eclipse starts, modified by any startup parameters set on a command line or in icon properties. (See Setting Eclipse startup preferences.)
2. The **Workspace Launcher** dialog appears and prompts you to specify the path of a workspace. (See Selecting a workspace.)
3. The Eclipse Workbench appears in one of the following modes:
4. If you specified a new workspace, Eclipse opens displaying the Welcome to Progress Developer Studio page.
5. If you specified an existing workspace, Eclipse attempts to start the AVM and any required database connections.The Workbench displays an existing workspace in the state in which it was left the last time it was opened.

Related Links

- Setting Eclipse startup preferences
- Selecting a workspace

Setting Eclipse startup preferences

- When you start Progress Developer Studio for OpenEdge from a desktop icon or from a Start menu selection, the default Eclipse startup command has the following format:
  `OpenEdge-install-dir/DeveloperStudio3.6/eclipse/eclipse.exe -vm OpenEdge_install_dir/DeveloperStudio3.6/jre/bin/javaw.exe`
- The default startup command points to the Eclipse executable (eclipse.exe) and the Java Runtime Environment (javaw.exe) that support Progress Developer Studio for OpenEdge.

- You might need to add startup parameters to the default so that Progress Developer Studio for OpenEdge runs on your system. For example, the default memory allocation (256MB) for Java might be inadequate in your operating environment. You can add the following command line parameters to increase the available memory for Java:
  `-vmargs -Xmx<memory size>`
- In Windows, you can open the properties of a desktop icon or a Start menu selection and add startup parameters in the Target field of the **Properties** dialog .

  Note: Running Eclipse (in the Eclipse Workbench User Guide), scroll to Advanced Topics in Running Eclipse for a complete list of startup parameters.

Selecting a workspace

- When you start Progress Developer Studio for OpenEdge, Eclipse prompts you to specify a workspace in the **Workspace Launcher** dialog. A workspace is a folder that contains the content of your projects along with any project metadata.
- You can specify an existing workspace, or you can specify a folder in your file system that will function as a workspace. By default, the **Workspace Launcher** dialog maintains a list of the last five workspaces that you used.
- You can change workspaces when Progress Developer Studio for OpenEdge is running by selecting **File > Switch Workspace** from the main menu bar. Progress Developer Studio restarts when you change workspaces. If you want to work in multiple workspaces, you must start a separate instance of Progress Developer Studio for each workspace.

Working with OpenEdge perspectives

	Demo Use and Customize Perspectives **Note:** The OpenEdge Release 10.2B demos have not been updated for OpenEdge Release 11.0. Since 10.2B, OpenEdge Architect has been renamed as Progress Developer Studio for OpenEdge. In addition, OpenEdge Release 11.0 includes some feature enhancements and other changes. Despite the differences between 10.2B and 11.0, the demos are still a useful introduction to Progress Developer Studio for OpenEdge.

The perspectives included with Progress Developer Studio for OpenEdge are OpenEdge AppBuilder, OpenEdge Server, Debugger, DB Navigator, OpenEdge Editor, OpenEdge Tools for Business Logic, and OpenEdge Visual Designer.

Related Links

- Opening a perspective
- Modifying a perspective
- Setting a perspective to its default state

Opening a perspective

Opening a perspective

Demo
Use and Customize Perspectives

Note: The OpenEdge Release 10.2B demos have not been updated for OpenEdge Release 11.0. Since 10.2B, OpenEdge Architect has been renamed as Progress Developer Studio for OpenEdge. In addition, OpenEdge Release 11.0 includes some feature enhancements and other changes. Despite the differences between 10.2B and 11.0, the demos are still a useful introduction to Progress Developer Studio for OpenEdge.

To open an OpenEdge perspective in the Eclipse Workbench:

1. Select **Window > Open Perspective**.
2. Select an OpenEdge perspective from the list.

Modifying a perspective

Demo
Use and Customize Perspectives

Note: The OpenEdge Release 10.2B demos have not been updated for OpenEdge Release 11.0. Since 10.2B, OpenEdge Architect has been renamed as Progress Developer Studio for OpenEdge. In addition, OpenEdge Release 11.0 includes some feature enhancements and other changes. Despite the differences between 10.2B and 11.0, the demos are still a useful introduction to Progress Developer Studio for OpenEdge.

You can add views to a perspective, and you can close currently open views, thereby removing them from the perspective. You can save a given configuration of views as a custom perspective.

To modify the current perspective:

1. Select **Window > Show View** from the main menu bar.
2. Select Other to display a full set of views, organized in categories shown as folders, from which you can choose. You can select any view. Progress Developer Studio views are in folders whose names begin with "Progress."

   Note: The views in the Progress OpenEdge Support Views folder are not intended to be selected. OpenEdge automatically displays these views when they are required. If selected from the Show View list, they may appear empty and non-functional.

3. To remove a view from the current perspective, close it by clicking the X on the view's tab.

4. To save your current configuration of views as a perspective, choose **Save Perspective As** from the **Window** menu and enter a name. You can overwrite an existing perspective by entering its name (a confirmation prompt appears), but it is not recommended. It is advisable to leave the standard Progress Developer Studio perspectives unchanged so that you retain the ability to return to a default state.

Setting a perspective to its default state

Demo
Use and Customize Perspectives

Note: The OpenEdge Release 10.2B demos have not been updated for OpenEdge Release 11.0. Since 10.2B, OpenEdge Architect has been renamed as Progress Developer Studio for OpenEdge. In addition, OpenEdge Release 11.0 includes some feature enhancements and other changes. Despite the differences between 10.2B and 11.0, the demos are still a useful introduction to Progress Developer Studio for OpenEdge.

1. After you change a standard perspective by adding or removing views, you might want to reset it to its unmodified state. To do so, choose **Window > Reset Perspective**, and click OK at the confirmation prompt.
2. The perspective returns to its last-saved state. To ensure that you can always return to the original, as-installed state, refrain from saving changes to a standard perspective. Instead, save your custom perspective with a new name.

Set workspace preferences

Workspace preferences are settings that apply to all the projects in a workspace. To set workspace preferences for OpenEdge projects:

1. Select **Window > Preferences**.
   Preference pages contributed by plugins are listed in the tree view in the left pane.
2. Expand the OpenEdge node.
3. Select a preference page and specify the desired options.

Related Links

- Set the default AVM startup parameters
- Setting up a shared AVM

Set the default AVM startup parameters

To set the default AVM startup parameters:

1. Select **Window > Preferences**.
   Preference pages contributed by plugins are listed in the tree view in the left pane.
2. Expand the Progress OpenEdge node.

3. Select the Startup node.
4. Specify the desired port ranges and startup parameters.
   Do not specify database connections as startup parameters. Use the **Database Connections** tab in the **Project Properties** dialog instead.

The startup parameters in the Default Development Startup field can be included, appended to, or overridden on a project's OpenEdge properties page. The default startup parameters can also be included, appended to, or overridden on the Shared OpenEdge AVM preference page.

Setting up a shared AVM

To configure a shared AVM:

The shared AVM starts if there is a project associated with it.

1. Select **Window > Preferences** from the main menu bar of Progress Developer Studio for OpenEdge.
2. Open **Progress OpenEdge > Shared OpenEdge AVM** in the tree view of the **Preferences** dialog.
3. Set the AVM startup parameters on the Shared OpenEdge AVM preference page.
4. In the child nodes of the Shared OpenEdge AVM page, you can set the other properties (AppBuilder, Assemblies, Database connection, and PROPATH) that affect all the projects that use the shared AVM. In addition, you can choose which projects use the shared AVM.

6

Creating your first ABL application

In this section, you will be guided through the creation of a simple ABL application using the Client-Server technology. In the following topics, you will create an OpenEdge project, and procedure files for the Server and Client side of the application. The simple ABL application that you create will have access to an OpenEdge database.

You will also assign the Server procedure file to an AppServer, and accesses and display the information received from the database using the Client file.

Related Links

- Creating a new AppServer project
- Connecting the OpenEdge AppServer to the OpenEdge Database
- Creating an ABL Procedure file for the Server
- Creating an ABL Procedure file for the Client
- Configuring the AppServer and associating the AppServer module
- Running the client which accesses the ABL code running in the AppServer

Creating a new AppServer project

You will use the **New OpenEdge Project wizard** to create a faceted OpenEdge project in a simple and configurable manner.

To create an AppServer project, follow the given steps:

1. From the workbench menu, select **File > New > OpenEdge Project**. The New OpenEdge Project dialog box appears.
2. Specify **CustomerReport** in the **Project name** field.
3. In the drop-down list of **Project Type configuration**, OpenEdge is selected by default. Change this to **AppServer**.
4. Click **Next**.
5. The **Select AVM and layout options** dialog box appears. You can use this page to specify the project runtime, the project source folder, and the project r-code folder. Keep all default values and click **Next**.
6. The **Define AppServer Content Module** page appears. Select the server that should publish the AppServer module as **asbroker1 AppServer**.
7. The **Define PROPATH** page appears. Click **Next**.
8. In the **Select database connections** page that appears, click **Configure database connections**. The Database Connections preference page that appears is used to maintain connections available to the project. The page provides a list of available database connections as defined for the workspace.
9. For this app that you are creating, you must add a new connection to the `Sports2000` database provided with OpenEdge.
   a. Click **New** to add a new connection profile where you can define a new workspace database connection.
   b. In the **Add Connection Profile page** that appears, enter `Sports2000` as the connection name.
   c. Browse for the complete path name of an OpenEdge database. The `sports2000.db` is available in the **<instalaltion_directory>/OpenEdge** directory.
   d. Enter `localhost` as the Host name.
   e. Enter port as `5555`.
   f. Click **Next** until you reach the **Define Data Server Configuration** page.
   g. Click **Finish**.
   h. Click **OK**.
10. In the **Select Database Connections page**, select the database connection you just created and click **Finish**.
11. Click **Yes** when prompted if you want to open the OpenEdge Server perspective.

By default, the AppServer project you just created will be launched in the Project Explorer.

Connecting the OpenEdge AppServer to the OpenEdge Database

To enable your app to access OpenEdge database data, the OpenEdge AppServer must be connected to an OpenEdge database.

1. To connect the OpenEdge AppServer to the OpenEdge Database:
   a. Double-click **asbroker1** in the **Server** view.
   b. In the **asbroker1** tab that opens in the editor, click on the link, **Open launch configuration**. This opens the **Edit launch configuration properties** page.
   c. Go to the **Database** tab, and select the **Show all** radio button.

   d. Select the **Sports2000** database connection.
   e. Click **OK**.
   f. Close the **asbroker1** editor.
2. To start the AppServer:
   a. In the Servers view, right-click **asbroker1**, and click **Start** in the pop-up menu.
      This starts the AppServer. In the Servers view, the status of the server changes to Started. If the server fails to start, check for the reason that it failed in the Console.

Creating an ABL Procedure file for the Server

This topic shows you how to create an ABL Procedure file using the New ABL Procedure wizard. This procedure file will contain the business logic for the application.

1. Select **File > New > ABL Procedure** from the main menu.
2. The Container field specifies the currently open project or a folder within the project where the procedure file is to be created. Browse to the **\CustomerReport\AppServer** project, if not already selected.
3. Enter the name of the procedure file as `Server`. The .p extension is appended automatically.
4. Click **Finish**.
5. In the Procedure file that opens in the Editor, copy and paste the following code in the file.
   ```
   DEF INPUT PARAM customerNumber AS INT64. DEF OUTPUT PARAM customerName AS
   CHARACTER. IF CONNECTED("sports2000") THEN DO: FIND FIRST customer WHERE
   custNum = customerNumber NO-LOCK NO-ERROR. IF AVAILABLE customer THEN
   customerName = Name. ELSE customerName = "No record". END. OUTPUT CLOSE.
   ```
6. Save the changes. Right-click anywhere in the edit pane and select **Check Syntax**. You see a message that says "**Syntax Checked: OK**".
7. Click **OK** to close message.

Creating an ABL Procedure file for the Client

This topic shows you how to create an ABL Procedure file using the New ABL Procedure wizard. This file will contain

1. Select **File > New > ABL Procedure** from the main menu.
2. The Container field specifies the currently open project or a folder within the project where the procedure file is to be created. Browse to the **CustomerReport\AppServer** project, if not already selected.
3. Enter the name of the procedure file as `Client`. The .p extension is appended automatically.
4. Click **Finish**.
5. In the Procedure file that opens in the Editor, copy and paste the following code in the file.
   ```
   DEFINE VAR custNum AS INTEGER INIT 1. DEFINE VAR custName AS CHARACTER.
   DEFINE VARIABLE UserName AS CHARACTER. DEFINE VARIABLE Pwd AS CHARACTER.
   DEFINE VARIABLE happsrv AS HANDLE NO-UNDO. CREATE SERVER happsrv. IF
   happsrv:CONNECT("-H localhost -S 3090 -AppService asbroker1 -DirectConnect")
   THEN DO : RUN Server.p ON happsrv (INPUT custNum, OUTPUT custName). MESSAGE
   custName VIEW-AS ALERT-BOX. happsrv:DISCONNECT(). END. DELETE OBJECT happsrv.
   ```

6. Save the changes. Right-click the editor and select **Check Syntax**. You see a message that says "**Syntax Checked: OK**".
7. Click **OK** to close message.

Configuring the AppServer and associating the AppServer module

This section shows you how to work with the AppServer, associate the module to the AppServer, edit the AppServer properties, restart the AppServer, and publish the module.

1. Select the **Servers** view.
2. Double-click the **asbroker1 AppServer**. The **Server Configuration** dialog box opens in the Editor. Ensure that all your configurations are correctly set.
   You use the Server Editor to view or modify the server properties that define the connection for OpenEdge Explorer and also the Broker. The Server Editor provides information on the following:
   - General Information - Provides the host name and other common settings.
   - Connection - Specifies the information on connection to OpenEdge Explorer.
   - Publishing - Specifies when to publish.
   - Timeouts - Specifies the time limit to complete server operations (Start and Stop).
   - Publish Location - Specifies the server publish directory.
3. In the **Servers** view, right-click on **asbroker1 AppServer** and select **Add and Remove**. The **Add and Remove** dialog box appears.
4. The project, **CustomerReport** should be already selected in the **Configured** section of the dialog box. If it is not, select CustomerReport from the **Available** section and click **Add and Remove** to add the CustomerReport project to the configured section.
5. Click **Finish**.
   The **CustomerReport** project is added and listed under **asbroker1 server** in the **Servers view**.
6. From the workbench menu, select **Run > Run Configurations**.
   The Run Configurations dialog box appears where all the run configurations grouped by category are listed together.
7. Expand **Progress OpenEdge AppServer** and select **asbroker1 AppServer**.
   A launch configuration defines the characteristics of the AVM instance under which the selected program runs. These characteristics include configurations such as startup parameters, PROPATH settings, and environment settings for the AVM session; database connections; and whether the program uses a dedicated instance of the AVM or the instance under which your OpenEdge project is currently running.

   You can use the Configurations wizard to define all the launch configuration's characteristics. Although this wizard contains a large number of fields on multiple tabs, defining a launch configuration need not be a complicated task. In fact, with a single click, you can create and run a launch configuration that uses default settings, and then edit any of these settings, if necessary.

8. Select the **Databases** tab. Select the **Show All** option button to show all database connections available in the workspace. **Sports2000** connection is listed.
9. Select the check box next to **Sports2000** if not selected. Click **Apply** and then click **Run** to start the server.

This starts the AppServer and the status changes to **Started** in the Servers view.

10. In the Servers view, right-click on **asbroker1 AppServer** and select **Publish**.
    The server status changes to **Synchronized**.

Running the client which accesses the ABL code running in the AppServer

This section shows how to run the Client file on the AppServer.

1. Select the **Client.p** file in the Explorer.
2. Right-click on the file and select **Run As > Run as OpenEdge Application**.
3. The Client file runs on **asbroker1 AppServer** and the result appears in an alert-box.

The name of the first customer is displayed in an Alert-box.