Progress®

OpenEdge DevOps Framework

OpenEdge®

Table of Contents

Copyright

April 2022

Product version: Progress OpenEdge DevOps Framework 2.1

Preface

Purpose

This manual provides information about OpenEdge DevOps framework for ABL applications. The OpenEdge DevOps framework is designed to implement an efficient Continous Integration (CI) pipeleine to handle compilation, repositry integration, testing, and packaging.

Audience

This document is intended for OpenEdge DevOps Engineers, and OpenEdge Developers.

Organization

- Learn About the OpenEdge DevOps Framework on page 9

  This section provides and introduction to OpenEdge DevOps Framework.

- Gradle on page 13

  This section covers the prerequisites for using the OpenEdge DevOps Framework and some basic concepts of Gradle.

Documentation conventions

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

I

Introduction

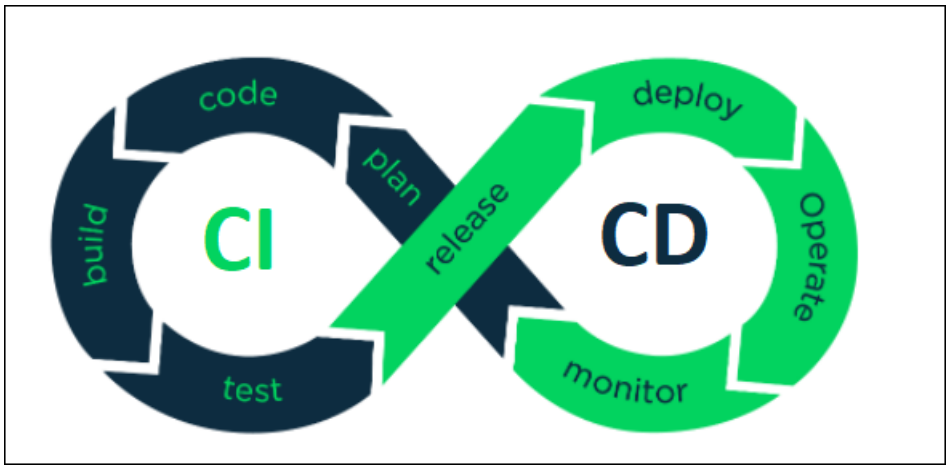
For details, see the following topics:

- Learn About the OpenEdge DevOps Framework

Learn About the OpenEdge DevOps Framework

Read the End User License Agreement.

Mission critical business applications go through many cycles of incremental changes over the course of their lifespan, complete with code changes, compilation, builds, and validation. Often these cycles are repeated multiple times a day, before a release is made available to users. Continuous integration (CI) is a practice that focuses on optimizing the process of generating these incremental builds and validating them. In the image below, CI represents DevOps and describes a typical software development and delivery process.



For ABL applications, the OpenEdge DevOps Framework is designed to help with implementing an efficient CI pipeline that handles compilation, repository integration, testing, and packaging. It also provides the convenience of sharing the CI pipeline configuration between the development and production build processes.

The OpenEdge DevOps Framework is comprised of a set of plugins designed to address the requirements of two types of users:

1. Users who are new to the CI process and want a simple way to set up their pipeline.
2. Advanced DevOps engineers with a complex CI process and additional flexibility needs.

The OpenEdge DevOps Framework also provides the convenience of sharing the CI pipeline configuration between the development and production build processes.

The following sections covers prerequisites for using the OpenEdge DevOps Framework, basic concepts of Gradle, understanding build process, managing dependencies, and details of the ABL Base plugin and the ABL plugin.

II

Gradle plugins

For details, see the following topics:

- Prerequisites
- Learn the basics
- Plugins
- Migrate from Apache Ant

Prerequisites

To use OEDF Gradle plugin, you must have:

- Certified OpenEdge 12.2 and later.
- ABL installed with OpenEdge 12.2 and later.
- Gradle 7.3.3.

  Note: You must use Gradle 5.6.x for OpenEdge DevOps Framework version 1.x.

- Java 11.
- PCT, which is used from DLC or pct by default.

  You can optionally specify a different PCT version by providing `PCT_VERSION` property either as system property, Gradle property, or s an environment variable.

- An ABL project.
- A basic understanding of Gradle.

OpenEdge DevOps Framework (OEDF) is supported on all the platforms that OpenEdge supports. For more information, see the OpenEdge Platform and Product Availability Guide for each OpenEdge release.

Setup the environment

OpenEdge provides the `progradle.bat/sh` script to simplify the OpenEdge DevOps Framework setup in your machine. This script sets up your machine to run the Gradle scripts using the OpenEdge DevOps Framework plugins.

The following activities happen in the background when you execute the `progradle.bat/sh` script:

- The specified version of gradle is automatically installed when you run the script for the first time.

  Note: For OpenEdge 12.2 and later, this script installs `Gradle 5.6.x`. If you are using OpenEdge DevOps Framework 2.0 or later, then update the default specified version to 7.3.3 at `$DLC\gradle\wrapper-scripts\gradle\wrapper\gradle-wrapper.properties`.

- The JDK used by OpenEdge is automatically configured to run the gradle build scripts.

You can alternatively install and configure the supported version of Gradle and Java separately. OpenEdge DevOps Framework (OEDF) is supported on all the platforms that OpenEdge supports. For more information, see the *Platform & Product Availability Guide* for the respective OpenEdge release.

How to configure a property

OEDF provides options to configure your project build by setting properties. There are numerous properties that are supported by OEDF depending on the project requirements. For example, one such property supported is `PCT_VERSION`, which, if set, instructs OEDF to use `PCT` from Maven instead of using the OpenEdge installation path.

You can configure a property in the following ways:

1. Set as a system property.

   For example, run the `-DPCT_VERSION=<PCT-version>` CLI command.

2. Set as a Gradle property.

   For example, set `PCT_VERSION=<PCT-version>` in the `gradle.properties` file.

3. Set as an environment variable.

The prefered order of precedence (highest to lowest) to configure a property is 1>2> 3.

Learn the basics

This topic describes basic understanding of Gradle and the typical build process followed for an ABL project.

Gradle

Gradle is an open-source project automation tool, which is an evolution of the concepts used in Apache Ant and Apache Maven. Instead of traditional XML, Gradle uses a domain-specific language based on Groovy for project configuration. Gradle intelligently determines which parts of a build tree are up to date, and executes only those parts that are needed. This process allows builds to be smarter, faster, and more efficient because it eliminates redundancies in the build phase.

The OpenEdge DevOps Framework provides Gradle plugins to evolve the way ABL applications are built. It is important to understand some of the Gradle terminologies and concepts that are used frequently across this guide. Some of these terminologies and concepts are covered in Basic Concepts below. For more information about Gradle, see the Gradle documentation.

Basic concepts

The following basic Gradle terminologies and concepts are helpful when using the OpenEdge DevOps Framework:

- Plugin—Gradle plugins add task objects and conventions to your Gradle environment.

  Plugins are the primary method by which you extend your capabilities using Gradle.

- Task—A Gradle task is the smallest piece of work for a build such as compiling classes or generating ABL doc.

  A task is the building block of Gradle functionality.

- Task type—Defines the blueprint of a task.

  Some tasks may require configurations like an input or output parameters to provide certain capability. The blueprint for all such parameters is defined by the Task type for a certain piece of work that can be set while creating a Task.

- Task dependencies—Gradle tasks can depend on other tasks.
- Extension—Plugins can add Extension that allows you to configure several settings and properties for the plugins.

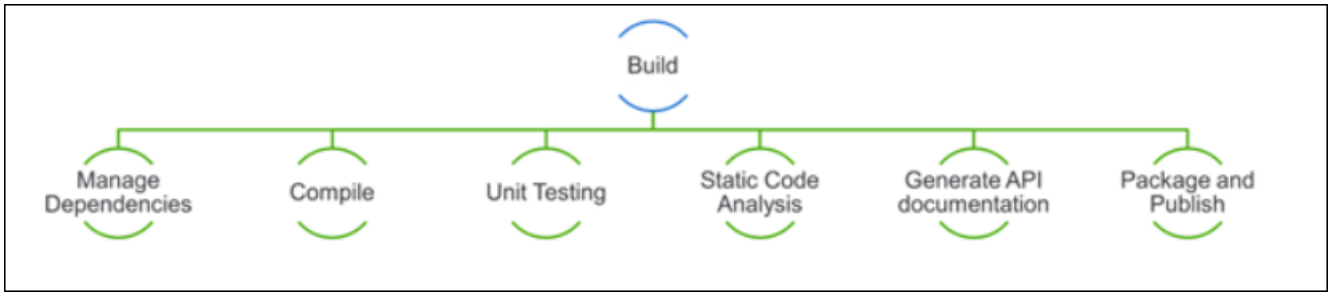
  Task types use these settings and properties as global configurations to set some of their properties.

- Groovy DSL—You can write Gradle build scripts using a Groovy or Kotlin DSL.

For more information about these and other Gradle components, see the Gradle documentation.

Understand build process of an ABL project

Building an ABL project may involve various steps from managing project dependencies to compiling ABL files, performing unit tests, generating ABL API documentation, packaging the artifacts and then publishing them to an artifact repository.



It is important to understand these steps in depth before you embark on writing a build script.

This topic is intended to familiarize you with the steps and requirements needed while building an ABL project.

Manage dependencies

An ABL application can have various dependencies like library files containing ABL code, databases, application packages (for example, a `.war files`), resource files (images, icons, configurations), and many more. Thus, managing project's dependencies is an important part of building an ABL project.

OpenEdge client or ABL Virtual Machine (AVM), which is the platform for compiling and running ABL code, uses **PROPATH** to search and locate ABL files and resource files. The below section describes all the dependencies that a **PROPATH** can manage.

PROPATH specific dependencies

AVM can search and locate files based on following types of values in PROPATH:

- Path to a directory containing ABL code and resources.
- Path to a `Procedure Library` file containing only r-code.

Procedure Library (PL) dependencies in PROPATH

Procedure Library (PL files) is the most common way to bundle ABL code as library. PL files can bundle both `r-codes` and potentially any other source files (ABL files such as procedure files, class files and include files) or resource files.

Since, AVM cannot recognize PL files with content other than `r-code` in **PROPATH**, you must handle PL file dependencies in the following ways:

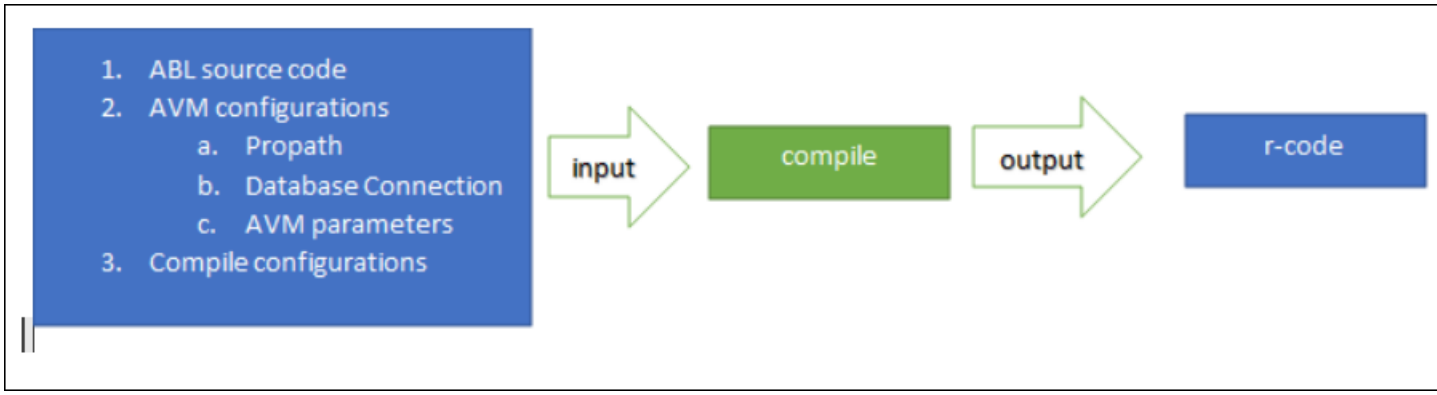
- PL file that contain only `r-code`— Directly add to the **PROPATH**.
- PL file containing other source files (such as ABL sources, image files, config files, and other files)— First extracted, and then add the path of the extracted directory to the **PROPATH**.

Note: The dependencies pertaining to database artifacts such as schema file, structure file, backup file, etc. and application packages such as WAR files, OEAR files, PAAR files, etc. are not yet supported.

Compile source code

Compiling source code refers to converting ABL source codes into the binary `r-code` that is then executed inside the ABL Virtual Machine (AVM).

To compile an ABL code, you require an ABL source code and an AVM with compiler. The illustration below typically represents the input and output parameters for a compile operation:



ABL source code

ABL files such as procedure files and class files that need to be compiled.

AVM configurations

AVM configuration consists of:

1. Propath—The paths where the AVM needs to look for resolving various project dependencies. See Managing Dependencies for more details.
2. Database connection—For projects that require data from a database, you need to provide the database connection details.

   In a CI environment, it is useful to use a test database. Use single user mode (`-1`) for connection details to overcome the overhead of starting a database server.
3. AVM parameters—The startup parameters as supported by AVM.

Compile configurations

There are various options provided by the ABL's COMPILE statement that can be configured while compiling, such as OPTIONS (strict options), LISTING, etc. For more information, see COMPILE statement in the *OpenEdge Development: ABL Syntax Reference* guide.

Run unit test

Running unit tests is important to eliminate bugs at the code level or unit level. A unit is the smallest entity which can independently exist, for example: a program module.

Unit testing verifies whether the smallest entity functions correctly when isolated from the rest of the code to ensure application features work at the component level.

Progress OpenEdge provides the **ABL Unit testing framework** to unit test ABL programs. To run unit tests, you will require sources (test files) and AVM. See AVM configurations to configure AVM for propath, database connection and AVM parameters.

Run static code analysis

Code Analyzer for ABL helps in analyzing and measuring the quality of your code and highlights any problems found.

Code Analyzer for ABL also provides recommendations on how to fix the problem. This helps to achieve coding best practices and to improve product performance. For more information, see Introduction to Code Analyzer for ABL.

Note: Code Analyzer for ABL is currently not supported with OpenEdge DevOps Framework gradle plugins.

Generate API documentation

You can generate API documentation (ABLDoc) in HTML format (default) from ABL source code to refer to your codes outside OpenEdge.

You can generate the ABLDoc documentation for the following source codes:

- Class (`.cls`) files.
- Procedure (`.p`) files.
- Include (`.i`) files.

Note: Generating ABLDoc is not natively supported by OpenEdge DevOps Framework gradle plugins. You can alternatively use the ABLDoc ant task in your gradle script.

Package and publish

Packaging refers to creating and bundling the build output of an ABL application. How you package and potentially publish your ABL project depends on what type of project it is. For example, libraries, applications, and web applications (PASOE applications) will have different requirements and will produce different set of artifacts.

A common package step could be creating a Procedure Library (PL) artifact by packaging the compiled `r-code` files.

Note: You can write a custom task using the `PL` task type to create a PL artifact.

Server project (PASOE application) require creating `OEAR`, `OEDS`, `WAR` or `PAAR` files.

Note: Tasks to create a server project package artifact is not yet supported in OEDF.

The packaged artifacts can then be published to an artifact repository using one of the Gradle's publishing plugins:

- Maven publish plugin .
- Ivy publish plugin.

Plugins

Plugins are the primary method by which all the useful features are provided in Gradle. Plugins add various capabilities (such as task types to compile code) and conventions (such as creation of default tasks to build a project, depending on the project structure).

There is a vast number of plugins available in the Gradle community that can be used depending on the build needs such as the base plugin that provides some tasks and conventions that are common to most builds, maven publish plugin that provides the ability to publish build artifacts to an Apache Maven repository, and many more.

The OpenEdge DevOps Framework provides Gradle plugins for ABL projects. When creating Gradle plugins, it is important to separate the capabilities from convention. This is recommended because conventions can be opinionated (by having predefined tasks and defaults) and might expect a particular project structure (such as a PDSOE project) whereas in real time scenario projects might differ for some customers. Hence, the plugins provided by the OpenEdge DevOps Framework are designed to be flexible enough to cater to all the customer needs.

Currently, there are two ABL gradle plugins available:

- ABL base plugin (progress.openedge.abl.base), and
- ABL plugin (progress.openedge.abl)

ABL base plugin (progress.openedge.abl-base)

The ABL base plugin defines various capabilities, under task types and extensions. Task types define individual task capabilities like compile ABL code, run ABL unit tests, and other tasks. Extensions are global configurations available to all the task types.

The ABL base plugin is intended for users who have ABL source files that are not structured as Progress Developer Studio projects. Users can create and define their own tasks for various steps required for building an ABL project by using the task types; for example, using `ABLCompile` task type to create a task for compiling ABL code. For more information, see ABL Base Plugin .

ABL plugin (progress.openedge.abl)

The ABL plugin depends on the Progress Developer Studio project structure and is recommended for Progress Developer Studio users. This plugin, by default, provides various predefined tasks required to build an ABL project (such as task to compile ABL sources), based on the project's propath and properties. This enables you to build your project with minimum effort and enforce best build practices.

These predefined tasks and the necessary parameters required for these tasks (for example, compiling ABL files) can be configured in the `build.config` file that is part of each Developer Studio project. This ensures that you do not have to maintain separate configuration for development environments and build environments.

Users that do not have a PDSOE project but still prefer using predefined tasks can use the ABL plugin, provided the users add and configure the required `build.config` and `.propath` files for their source artifacts. For more information about how to use and configure this plugin, see ABL Plugin .

ABL base plugin

The ABL Base plugin provides various capabilities and features required to run or build ABL applications such as running ABL procedure, compiling ABL sources, running ABL unit tests, creating Procedure Library (PL) files, and other such capabilities.

The ABL base plugin provides *Task Types* using which you can create tasks to perform various actions described above. The name of task types gives an idea about what they are supposed to do. For more information, see Task Types .

When writing tasks to build an ABL project there can be a requirement to pull dependencies such as `PL` files from remote repositories. The base plugin provides configurations which can be used to manage dependencies. For more information , see Dependency Management.

Usage

To use the ABL Base plugin, include the following in your build script (`build.gradle` file):

```
plugins {

  id "progress.openedge.abl-base" version <version>

}
```

Task types

Create tasks using these task types:

ABLCompile

The ABLCompile task compiles ABL code.

Methods

Method	Description	Example
`exclude(String... excludes)`	Adds an ANT-style exclude pattern to this task.	`exclude('**/*.txt', '**/*.conf')`
`include(String... includes)`	Adds an ANT-style include pattern to this task.	`include('**/*.p', '**/*.cls')`
`source(String... sources)`	Adds a source to this task after the include and exclude patterns are applied. Sources can have an absolute or relative path to the root directory.	`source('src1', 'src2')`
`propath(String... propaths)`	Sets the AVM's PROPATH. This method will be appended to the PROPATHs provided in ABLExtension	*`propath`*`('path1', "${dlcHome}/path2")`
`avmOptions(Closure avmOptions)`	AVM options. The argument should be provided as a closure. See `avmOptions` in the ABLCompile section for more information.	`avmOptions{ tmpDir="path" tty.enabled=true}`
`dbConnection(Closure connection)`	Adds database configurations to this task.	`dbConnection{ dbName = 'sports2000' connectionParameters='-S 8000'}`
`dbConnectionReferenceId(String... refIds)`	Adds database task references. See the *id* property of DBConnection for more information. You must add the database tasks' dependency to this task.	`dbConnectionReferenceId("id1", "id2")`
`compileOptions(Closure compileOptions)`	Compile options. The argument should be provided as a closure. See the `compileOptions` section for more information.	`compileOptions{ multiCompile.enabled=true outputType="json"}`
`arguments(Map args)`	Additional arguments. These arguments are directly passed to PCT.	`arguments([k1:"v1", k2:'v2'])`

Properties

Property	Required?	Description	Example	Default value
`openedgeVersion`	No (read-only property)	Prints the OpenEdge version. This can be used for logging. This is a read-only property.	None	None
`rcodeDir`	No	Sets the directory where the r-code will be generated.	`rcodeDir="path"`	`rcodeDir` set in ABLExtension.
`propath`	No	Sets the PROPATH.	`propath = files('path1', "${dlcHome}/path2")`	`propath` set in ABLExtension
`wrkDir`	No	Sets the working directory.	`wrkDir = 'path'`	`wrkDir` set in ABLExtension
`avmOptions{}`	No	AVM options. See the `avmOptions` section for more information.	None	`avmOptions` set in ABLExtension.
`compileOptions{}`	No	Compile options. See the `compileOptions` section for more information.	None	`compileOptions` set in ABLExtension.
`arguments`	No	Additional arguments. These arguments are directly passed to PCT.	`arguments = [k1:"v1", k2:'v2']`	None

avmOptions{}

Property	Description	Example	Default value
`tmpDir`	The temporary directory for the AVM run time. `-T` is the startup parameter option.	`tmpDir = 'path1'`	None

Property	Description	Example	Default value
tty.enabled	The flag for using the _progres or prowin executables. Set to true for _progres, set to false for prowin (or prowin32 for a 32-bit AVM).	tty { enabled = 'true' }	true
xcodeSessionKey	The XCODE session key for the secuity policy.	xcodeSessionKey='myKey'	None
parameterFile	The parameter file. -pf is the startup parameter option.	parameterFile='pathToFile'	None
startupParameters	The startup parameters as a string. This will be ignored if a parameterFile is provided	startupParameters='-tok 4000 -s 200'	None
assembliesDir	The assemblies directory. -assemblies is the startup parameter option.	assembliesDir='path1'	None

compileOptions{}

Property	Description	Example	Default value
xrefDir	The path where xref will be generated.	xrefDir='xref2'	'${buildDir}xref'
debugListingDir	The path where the debug listing will be generated.	debugListingDir='debugList2'	'${buildDir}/debugList'
preprocessDir	The path where the preprocess output will be generated.	preprocessDir='preprocess2'	'${buildDir}/preprocess'
multiCompile.enabled	The COMPILER:MULTI-COMPILE attribute	multiCompile.enabled=true	false
strictOptions. requireFullNames	Specifies that the options require full names in the COMPILE statement. Value can be ignore or error.	strictOptions.requireFullNames='error'	ignore

Property	Description	Example	Default value
`strictOptions.requireFieldQualifiers`	Specifies that the options require field qualifiers in the `COMPILE` statement. Value can be `ignore` or `error`.	`strictOptions.requireFieldQualifiers='error'`	`ignore`
`strictOptions.requireFullKeywords`	Specifies that the options require full keywords in the `COMPILE` statement. Value can be `ignore` or `error`.	`strictOptions.requireFullKeywords='error'`	`ignore`
`strictOptions.requireReturnValues`	Specifies that the options require return values in the `COMPILE` statement. Value can be `ignore` or `error`.	`strictOptions.requireReturnValues='error'`	`ignore`
`listing.enabled`	The `LISTING` option in the `COMPILE` statement	`listing.enabled=true`	`false`
`xcodeKey`	The `XCODE` option in the `COMPILE` statement. Specify the expression as a string.	`xcodeKey='myKey'`	None
`xref.enabled`	The `XREF` option in `COMPILE` statement	`xref.enabled=true`	`false`
`xrefXml.enabled`	The `XREF-XML` option in the `COMPILE` statement. To enable this option, set the value to `true`.	`xrefXml.enabled=true`	`false`
`xrefString.enabled`	The `STRING-XREF` option in the `COMPILE` statement	`xrefString.enabled=true`	`false`
`xrefString.append`	The `APPEND` option in `STRING-XREF`	`xrefString.append=true`	`false`
`streamIO.enabled`	The `STREAM-IO` option in the `COMPILE` statement. To enable this option, set the value to `true`.	`streamIO.enabled=true`	`false`
`languages.list`	This is a comma-separated list of language segments to include in the compiled r-code.	`languages.list='lang1,lang2'`	None

Property	Description	Example	Default value
languages.textSegGrow	The TEXT-SEG-GROW option. Set the growth-factor as an integer. Supported only when a language list is provided.	languages.textSegGrow=10	None
debugList.enabled	The DEBUG-LIST option in the COMPILE statement	debugList.enabled=true	false
preprocess.enabled	The PREPROCESS option in the COMPILE statement	preprocess.enabled=true	false
v6Frame.enabled	The V6FRAME option in the COMPILE statement	v6Frame.enabled=true	false
v6Frame.useRevVideo	The USE-REVVIDEO option in V6FRAME	v6Frame.useRevVideo=true	false
v6Frame.useUnderline	The USE-UNDERLINE option in V6FRAME	v6Frame.useUnderline=true	false
minSize.enabled	The MIN-SIZE option in the COMPILE statement. To enable this option, set the value to true.	minSize.enabled=true	false
outputType	Use this option to change the output format. Available options are: json	outputType='json'	None

Sample code snippet

The following code snippet is an example using ABLCompile:

```
abl{
    propath("${dlcHome}/tty/")
    avmOptions {}
    compileOptions{}
    rcodeDir='rcode1'
    wrkDir = "myWrkDir"

    dbConnection{
        dbName = 'sports2000'
        connectionParameters='-S 8000'
    }
}

task myDbConTask(type: DBConnection){
    id='id1'
    dbName='sports2020'
    parameterFile='pathToFile'
    aliases = ['alias1', 'alias2', 'aalias3']
}

task myCompileTask(type: ABLCompile){
    println "OE version: ${openedgeVersion}"

    source('src1', 'src2')
    include('**/*.p', '**/*.cls')
    exclude('**/*.txt', '**/*.conf')
    rcodeDir='rcode1'
    propath('path1', "${dlcHome}/path2")
    wrkDir = 'path1'
    avmOptions{
        tty.enabled='true'
        startupParameters='-tok 4000 -s 200'
    }
    compileOptions{
        xrefDir='xref2'
        preprocessDir=""
        preprocess.enabled=""
        strictOptions{
            requireFullNames='warning'
        }
        outputType='json'
    }

    dbConnection{
        dbName = 'sports2020'
        parameterFile='pathToParameterFile'
        connectionParameters='-S 8000'
        port = '8000'
        host = 'localhost'
        username = 'admin'
        password = 'admin2'
        aliases = ['a1', 'a2']
    }
    dbConnectionReferenceId("id0")
    dbConnectionReferenceId("id1", "id2")
}

myCompileTask.dependsOn myDbConTask
```

ABLRun

Use this task to run ABL procedures.

Methods

Method	Description	Example
`propath(String... propath)`	Sets the AVM's PROPATH. Will be appended to the PROPATHs provided in `ABLExtension`.	`propath('path1', "${dlcHome}/path2")`
`profiler(Closure options)`	Enables profiling. See the `profiler` section for options	`profiler { enabled = true outputDir = "${buildDir}/profiler"}`
`procedure(String procedure)`	Provides the procedure file to be run.	`procedure('src/main.p')`
`environment(String name, String value)`	Adds environment variables to pass to the system command	`environment('k1', 'v1')`
`environment(Map environment)`	Adds environment variables to pass to the system command	`environment([k1:"v2", k2:"v2"])`
`dbConnectionReferenceId(String... refIds)`	Adds database task references. Reference the `id` property of the DBConnection tasks for more information. Note that you must add the database task dependency to this (compile task).	`dbConnectionReferenceId("id1", "id2")`
`dbConnection(Closure connection)`	Adds database configurations. The argument should be provided as a closure with its object following the structure supported by the DBConnection task type. Note that braces can be removed following the Java/Groovy syntax.	`dbConnection{ dbName = 'sports2000' connectionParameters='-S 8000'}`

`avmOptions(Closure avmOptions)`	AVM options. The argument should be provided as a closure.
	See the `avmOptions` section for more information.
		`avmOptions{`
		`tmpDir="path"`
		`tty.enabled=true`
		`}`

`arguments(Map args)`	Additional arguments for ABL procedures. (These arguments are directly passed to PCT).	`arguments([k1:"v1", k2:'v2'])`

Properties

Property	Required?	Description	Example	Default value
`dlcHome`	Yes Should be set as described in the `ABLExtension` section.	OpenEdge DLC path	None	None
`openedgeVersion`	No (read-only property)	Specifies the OpenEdge version. This can be used for logging. This is a read-only property.	None	None
`procedure`	Yes	Path of the procedure file that will be run. Can be provided using the method `procedure(String procedure)`.	`procedure = 'src/main.p'`	None
`propath`	No	Sets the AVM's PROPATH. Will be appended to the PROPATHs provided in `ABLExtension`.	`propath = files('path1', "${dlcHome}/path2")`	`propath` set in `ABLExtension`
`wrkDir`	No	Working directory from where the AVM is started	`wrkDir = 'path'`	`wrkDir` set in ABLExtension
`avmOptions`	No	AVM options. Use the method `avmOptions(Closure avmOptions)` to set the properties. The properties defined in this task take priority if the same property is defined in `ABLExtension`	None	`avmOptions` set in `ABLExtension`

Property	Required?	Description	Example	Default value
dbConnection	No	DB connection details. Use methods dbConnection(Closure connection) or dbConnectionReference(String... refIds) to add	None	dbConnections set in ABLExtension
environment	No	Environment variables to pass to the system command	environment = [k1:"v2", k2:"v2"]	None
arguments	No	Additional arguments for ABL procedures. (These arguments are directly passed to PCT.)	arguments = [k1:"v1", k2:'v2']	None

Profiler

Property	Description	Example	Default value
enabled	Enables the profiler	enabled=true	false
description	Description of the profiler session	description="description"	None
outputDir	Generates a profiler output in this directory, with a unique name. Will be ignored if outputFile is provided.	outputDir = 'path'	"${buildDir}/profiler"
outputFile	Profiler output file name	outputFile='path'	None
coverage	Enables code coverage	coverage=true	false
statistics	Enables statistics	statistics=true	false
debugListDir	Generates debug listing files in this directory	debugListDir="path"	None

Sample code snippet

The following code snippet is an example using ABLExtension:

```
task rumMain(type: ABLRun){
    procedure = "src/main.p"
    propath('src')

    wrkDir = "${buildDir}/rcode"
    avmOptions{
        tty.enabled = false
    }

    dbConnection{
        dbName="db/sports2000/sports2000"
        connectionParameters = "-1"
    }

    arguments = [ failOnError : true]
    profiler {
        enabled = true
        coverage = true
        outputDir = "${buildDir}/profiler"
        description = "Coverage for main.p"
    }
}
```

ABLUnit

Use this task to run ABLUnit tests. This task type has all the methods and properties that are in ABLRun (except `procedure`). Additional properties and methods are listed in the following table:

Methods

Method	Description	Example
`exclude(String... excludes)`	Adds an ANT-style exclude pattern	`exclude('**/*.txt', '**/*.conf')`
`include(String... includes)`	Adds an ANT-style include pattern	`include('**/*.p', '**/*.cls')`
`source(String... sources)`	Adds sources to this task, after the include and exclude patterns are applied. The task is skipped if no sources are available.	`source('src1', 'src2')`

Properties

Property	Required?	Description	Example	Default value
`outputDir`	No	Directory where the result file is placed. Don't use this property under Linux, because a bug prevents `results.xml` from being generated	`outputDir="path"`	`"${projectDir}"`

BackupDB

Use this task to create a backup of an OpenEdge database.

Properties

Property	Required?	Description	Example	Default value
`source`	Yes	Path to the database where the `.db` file is located. Can be relative or absolute. May or may not include the `.db` file extension.	`source="path"`	None
`target`	Yes	Path of the backup file for the database. Can be absolute or relative.	`target="path"`	None
`overwrite`	No	Whether to replace if the backup file exists	`overwrite=true`	`false`

Property	Required?	Description	Example	Default value
online	No	Indicates the database must be online or not when doing a backup. If not set, the proutil -C holder is used to detect if the database is up. Leave this out if you want the task to auto detect if this option should be used.	online=true	Auto
incremental	No	Performs an incremental backup.	incremental=true	false

Sample code snippet

The following code snippet is an example using BackupDB:

```
task createDBBackup(type: BackupDB){
    source = "db/sports2020/sports2020.db"
    target = "${buildDir}/dist/sports2020.bck"
    overwrite = true
}
```

CreateDB

Use this task to create an OpenEdge database.

Methods

Method	Description	Example
arguments(Map args)	Additional arguments. These arguments are directly passed to PCT.	arguments([k1:"v1", k2:'v2'])

Properties

Property	Required?	Description	Example	Default value
dbName	Yes	Database name	dbName="dbName"	None
outputDir	No	The directory where the database will be created	outputDir="path"	"${projectDir}"

Property	Required?	Description	Example	Default value
sourceDb	No	Copy the specified database to the target database.	sourceDb="path"	If attribute is not provided, the empty DB is used
schemaFile	No	Initial dump files to load into database. Separate dump filenames with commas. Files are resolved first as an absolute path, then relative to base directory. Wildcards are not expanded.	schemaFile="path"	None
structFile	No	Structure description file	structFile="path"	None
blockSize	No	Block size in kilobytes (1, 2, 4, or 8). Cannot be used with sourceDb attribute.	blockSize=2	8
tmpDir	No	The -T parameter when loading a schema	tmpDir="path"	None
cpInternal	No	The -cpinternal parameter when loading a schema	cpInternal="value"	None
newInstance	No	Appends -newInstance in the PROCOPY command line.	newInstance=true	false
largeFiles	No	Enable large files for this database.	largeFiles=true	false
arguments	No	Additional arguments for databases. (These arguments are directly passed to PCT.)	arguments = [k1:"v1", k2:'v2']	None

Sample code snippet

The following code snippet is an example using CreateDB:

```
task createSports2000(type: CreateDB){
    dbName = 'sports2000'
    sourceDb = "${dlcHome}/sports2000"
    outputDir = "db/sports2000"
}
```

DBConnection

The DBConnection task defines the OpenEdge database connection, which can be referred to by a compile task.

Properties

Property	Required?	Description	Example	Default value
`dbName`	Yes, if `parameterFile` is not set	The physical database name (`-db` parameter). This can also be a full path.	`dbName = 'sports2020'`	None
`parameterFile`	Yes, if `dbName` is not set	The parameter file	`parameterFile='pathToFile'`	None
`connectionParameters`	Yes, if `dbName` and `parameterFile` are not set	The connection parameters as string. A temporary pf file is created and used as the `parameterFile` property. This property will be ignored if *connection-name*`.parameterFile` is provided	`connectionParameters='-S 8000'`	None
`port`	No	The database port (`-S` parameter)	`port = '8000'`	None
`host`	No	The database host (`-H` parameter)	`host = 'localhost'`	None
`username`	No	The database user name (`-U` parameter)	`username = 'admin'`	None
`password`	No	The database user password (`-P` parameter)	`password = 'admin'`	None

Property	Required?	Description	Example	Default value
`aliases`	No	The database aliases	`aliases = ['alias1', 'alias2', 'aalias3']`	None
`id`	Yes, if using as a task	The identifier for this task, which can be referred to from a compile task.	`id = 'id1'`	None

Sample code snippet

The following code snippet is an example using DBConnection:

```
task con1(type: DBConnection){
    id = "idCon1"
    dbName = 'sports2020'
    parameterFile='pathToFile'
    aliases = ['a1', 'a2']
}
```

Note:

These properties are passed in the following order:

`-db <dbName> -pf parameterFile> <other - attributes like -S 8655>`

The properties that are defined later take priority, if the same parameter is repeated inside the `parameterFile`:

- If `-db` is defined inside the `pf` file and `dbName` is also set, then the property defined in the pf file takes priority.
- If any other parameter, such as `-S 8655`, is defined in the `pf` file and the corresponding attribute, such as *`port`*`='8665'`, is also set, then the attribute, *`port`*, takes priority.

Progress reccommends that you do not define multiple database connection details, because this can lead to unexpected behavior.

ExtractPL

Use this custom task to extract a PL file.

Method

Method	Description	Example
*from(String plFilePath	Specify the *pl* path.	from("./corelib.pl")
into(String destinationDir)	Specify the destination directory for the extracted files.	into("$buildDir/corelib")
exclude(String... excludes)	Adds an ANT style exclude pattern.	exclude('**/*.p', '**/*.cls')
include(String... includes)	Adds an ANT style include pattern.	include('**/*.r', '**/*.conf')
*arguments(Map args)	Additional arguments. • https://ant.apache.org/manual/Types/fileset.html • https://ant.apache.org/manual/Tasks/zip.html#ZipFileSet These arguments are directly passed to PCT.	arguments([dirmode:"0777", k2:'v2'])

Properties

Property	Required	Description	Example	Deafult Value
source	No	The file path to the PL which will be extracted.	source="./corelib.pl"	None
destinationDir	No	The destination directory for the extracted files.	destinationDir="$buildDir/corelib"	None
*overrideDuplicatesStrategy	No	-	-	None
arguments	No	Additional arguments. These arguments are directly passed to PCT.	arguments = [k1:"v1", k2:'v2']	None

Sample Code Snippet

```
task extractCorelibPl(type: ExtractPL){
    from("./corelib.pl")
    into("$buildDir/corelib")
    include('**/*.r')
    arguments = [ dirmode:"0777", encoding  : 'undefined' ]
}
```

LoadDBSchema

Use this task to load an OpenEdge database schema file to a database.

Methods

Method	Description	Example
`exclude(String... excludes)`	Adds an ANT-style exclude pattern.	`exclude('**/*.txt', '**/*.conf')`
`include(String... includes)`	Adds an ANT-style include pattern.	`include('**/*.df')`
`source(String... sources)`	The schema (`.df`) files to be loaded. Adds sources to this task after the include and exclude patterns are applied. The task is skipped if no sources are present.	`source('myDb.df', 'src/schema')`
`dbConnection(Closure connection)`	Adds database configurations. The argument should be provided as a closure with its object following the structure supported by DBConnection task type. Note that braces can be removed following Java/Groovy syntax.	`dbConnection{ dbName = 'sports2000' connectionParameters='-S 8000'}`
`dbConnectionReferenceId(String... refIds)`	Adds database tasks' references. Reference the `id` property of the DBConnection tasks for more information. Note that you mustadd the Database task dependency to this (compile task).	`dbConnectionReferenceId("id1", "id2")`
`arguments(Map args)`	Additional arguments for dtaabases. (These arguments are directly passed to PCT.)	`arguments([k1:"v1", k2:'v2'])`

Properties

Property	Required?	Description	Example	Default value
onlineChanges	No	Relaxes requirements in order to apply online changes (use SESSION:SCHEMA-CHANGE = 'NEW OBJECTS').	onlineChanges=true	false

Sample code snippet

The following code snippet is an example using LoadDBSchema:

```
task loadSports2000Schema(type: LoadDBSchema){
    source("myDb.df")
    onlineChanges = true
    dbConnection{
        dbName="db/sports2020/sports2020"
        connectionParameters = "-1"
    }
}
```

PL

Use this task to create a Procedure Library (PL) file.

Methods

Method	Description	Example
from(String... from)	Adds sources and r-codes to this task, after the include and exclude patterns are applied.	from("${buildDir}/rcode", 'src2')
exclude(String... excludes)	Adds an ANT-style exclude pattern	exclude('**/*.txt', '**/*.conf')
include(String... includes)	Adds an ANT-style include pattern	include('**/*.r', '**/*.p')
arguments(Map args)	Additional arguments for libraries. (These arguments are directly passed to PCT.)	arguments([k1:"v1", k2:'v2'])

Properties

Property	Required?	Description	Example	Default value
plFile	Yes, if sharedFile is not provided	The file path where the PL file is created	plFile="path"	None

Property	Required?	Description	Example	Default value
`sharedFile`	Yes, if `plFile` is not provided	Memory mapped library to create the PL file	`sharedFile="path"`	None
`cpInternal`	No	Internal code page (`-cpinternal` parameter)	`cpInternal="value"`	None
`arguments`	No	Additional arguments for libraries. (These arguments are directly passed to PCT.)	`arguments = [k1:"v1", k2:'v2']`	None

Sample code snippet

The following code snippet is an example using PL:

```
task createPL2(type: PL){
    plFile = "${buildDir}/dist/test.pl"
    from("${buildDir}/rcode")
    include('**/*.r')
    arguments = [encoding  : 'undefined' ]
}
```

Sample custom task

The following sample code demonstrates how you can use the ABLCompile and DBConnection task types in a custom task:

```
import com.progress.gradle.abl.tasks.*
plugins {
  id "progress.openedge.abl-base" version "1.0.0"
}

// This section defines the global configuration available to all task types.

abl{
    propath("${dlcHome}/tty/")
    avmOptions {}
    compileOptions{}
    rcodeDir='rcode1'
    wrkDir = "myWrkDir"

    dbConnection{
        dbName = 'sports2000'
        connectionParameters='-S 8000'
    }
}

// This section defines the DBConnection task.

task myDbConTask(type: DBConnection){
    id='id1'
    dbName='sports2020'
    parameterFile='pathToFile'
    aliases = ['alias1', 'alias2', 'aalias3']
}

// This section defines the ABLCompile task.

task myCompileTask(type: ABLCompile){
    println "OE version: ${openedgeVersion}"

    source('src1', 'src2')
    include('**/*.p', '**/*.cls')
    exclude('**/*.txt', '**/*.conf')
    rcodeDir='rcode1'
    propath('path1', "${dlcHome}/path2")
    wrkDir = 'path1'
    avmOptions{
        tty.enabled='true'
        startupParameters='-tok 4000 -s 200'
    }
    compileOptions{
        xrefDir='xref2'
        preprocessDir=""
        preprocess.enabled=""
        strictOptions{
            requireFullNames='warning'
        }
        outputType='json'
    }

    dbConnection{
        dbName = 'sports2020'
        parameterFile='pathToParameterFile'
        connectionParameters='-S 8000'
        port = '8000'
        host = 'localhost'
        username = 'admin'
        password = 'admin2'
        aliases = ['alias1', 'alias2', 'aalias3']
    }
    dbConnectionReferenceId("id0")
    dbConnectionReferenceId("id1", "id2")
}

myCompileTask.dependsOn myDbConTask
```

You can find more sample projects on the Progress community.

Dependency management

The ABL base plugin provides configurations which can be used to manage PL file dependencies. Refer to the sections below for the list of configutations provided to handle PL dependencies.

pl

Use this configuration to specify PL files dependencies containing only `r-code`. The PL files automatically downloads from a repository. These dependencies can then be used to configure the AVM's PROPATH in different tasks (for example, while creating compile tasks using ABLCompile).

Example

```
//declare repositories

repositories {

 maven {
  url ='<maven repo url>'
  metadataSources {
   artifact()//needed to downloaded artifacts (like .pl file) without a pom file in the
 repository
  }
 }
}

//define the dependencies
dependencies {
 pl("progress.openedge.oedf.test.apps:avengers:1.0.0@pl")
}
task myCompileTask1(type: ABLCompile){
 source("src/")
 propath(files(configurations.pl.files))
}
```

The `configurations.pl.files` returns a set of PL files defined using the `pl` configuration. The PL files are directly added to the **PROPATH**. AVM cannot recognize non `r-code` files inside the PL and you should use `plSource` to manage such dependencies.

pl source

Use the `plSource` configuration to specify PL files containing source files (such as ABL sources, image files, config files, etc.). The PL files are automatically downloaded from a repository and then extracted using the `ExtractPLTransform` transform. Use the extracted directory of these dependencies to configure the AVM's PROPATH in different tasks (for example, while creating compile tasks using `ABLCompile`).

Example

```
//declare repositories
repositories {
 maven {
  url ='<maven repo url>'
  metadataSources {
   artifact()//needed to downloaded artifacts (like .pl file) without a pom file in the
 repository
  }
 }
}

//define the dependencies
dependencies {
 pl("progress.openedge.oedf.test.apps:avengers:1.0.0@pl")
 plSource("progress.openedge.oedf.test.apps:avengers:1.0.0:sources@pl")
}

task myCompileTask2(type: ABLCompile){
 source("src/")
 propath(files(configurations.pl.files))
 propath(files(configurations.plSource.files))
}
```

The `configurations.plSource.files` returns a set of directories where `PL` files defined using the `plSource` configuration are extracted. Thus, the paths to extracted directories are added to the **PROPATH**.

Global configurations using the contributed extension

The ABL base plugin adds the `abl` extension, which allows you to configure numerous ABL language-specific configurations inside a DSL block. These configurations are available to all task types.

The task types can use these `abl` extention configurations by adding it to their existing configuration during runtime.

ABLExtension

This extension is a set of ABL language-specific configurations that are available to all task types.

Methods

Method	Description	Example
`propath`	Sets the AVM's PROPATH.	`propath('path1', "${dlcHome}/path2")`
`avmOptions(Closure avmOptions)`	AVM options. The argument should be provided as a closure. See `avmOptions` in the ABLCompile section for more information.	`avmOptions{ tmpDir="path" tty.enabled=true }`
`dbConnection({})`	Adds database configurations. See the ABLCompile section for more information.	`dbConnection{ dbName = 'sports2000' connectionParameters='-S 8000'}`

Method	Description	Example
`dbConnectionReferenceId(String... refIds)`	Adds database task references. See the *id* property of DBConnection for more information.	`dbConnectionReferenceId("id1", "id2")`
`compileOptions(Closure compileOptions)`	Compile options. The argument should be provided as a closure. See `compileOptions` in the ABLCompile section for more information.	`compileOptions{ multiCompile.enabled=true outputType="json"}`

Properties

Property	Required?	Description	Example	Default value
`dlcHome`	Yes	The OpenEdge DLC path. This property can be set in the following ways: **1.** Set *DLC* as a system property (`-DDLC=<dlcPath>` set on the command line) **2.** Set *DLC* as a Gradle property (`DLC=<dlcPath>` set in `gradle.properties` file) **3.** Set *DLC* as an environment variable The order of priority is 1 > 2 > 3.	None	None
`openedgeVersion`	No (read-only property)	Specifies the OpenEdge version. This can be used for logging. This is a read-only property.	None	None
`rcodeDir`	No	Sets the directory where the r-code will be generated.	*rcodeDir="rcode2" or rcodeDir="path"*	*${buildDir}/rcode*

Property	Required?	Description	Example	Default value
`propath`	No	Sets the AVM's PROPATH.	*propath* = files('path1', "${dlcHome}/path2")	None
`wrkDir`	No	Sets the working directory.	*wrkDir* = 'path1'	"${*projectDir*}".
`avmOptions{}`	No	AVM options. Use the method `avmOptions(Closure closure)` to set the properties.	None	None
`dbConnections`	No	Database connection details. To add a database connection, use methods, `dbConnection(Closure connection)` or `dbConnectionReferenceId(String... refIds)`.	None	None
`compileOptions{}`	No	Compile options. See the ABLCompile section for more information.	None	None

Sample code snippet

The following code snippet is an example using ABLExtension:

```
abl{
    rcodeDir='${buildDir}/myRcode'
    wrkDir = "."
    propath("${dlcHome}/tty/")
    avmOptions{
        tmpDir="${buildDir}"
        tty.enabled=true
    }
    compileOptions{
        multiCompile.enabled=true
        outputType="json"
    }
    dbConnection{
        dbName = 'sports2000'
        connectionParameters='-S 8000'
    }
    dbConnectionReferenceId("id1", "id2")
}
```

Transform

Transform is a Gradle concept that is used to transform or modify dependencies. OpenEdge DevOps Framework (OEDF) has added a transform for extracting PL file dependencies. For more information about the OEDF transform, see ExtractPL.

ABL plugin

The ABL plugin expands the capabilities of the ABL Base plugin to provide various default tasks and conventions. The ABL plugin extends the Gradle's Base plugin, and uses its lifecycle tasks to group the default tasks.

Project requirement

The ABL plugin depends on the Progress Developer Studio project structure and is recommended for Progress Developer Studio users. The mandatory files are:

- .propath file—Described in the Propath file.
- build.config file—Described in the the Build config file.

The ABL plugin provides various tasks required to build an ABL project (such as task to compile ABL sources) by default, based on the project's propath and properties. This enables you to build a project with minimum effort and enforces the best practices for your build.

These predefined tasks and the necessary parameters required for these tasks (for example, compiling ABL files) can be configured in the build.config file that is part of each Developer Studio project. This ensures that you do not have to maintain a separate configuration for development environments and build environments.

Users who do not have a PDSOE project but prefer to use predefined tasks can use the ABL plugin, provided the users add and configure the required build.config and .propath files for their source artifacts. For more information, see the Build config file and Propath file.

Usage

To use the ABL plugin, include the following in your build script build.gradle file.

```
plugins {

  id "progress.openedge.abl " version <version>

}
```

Propath file

This file is part of a Developer Studio project and contains details of the project's source paths and propaths.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<propath version="12.3" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="schema/propath.xsd">
    <propathentry env="all" kind="src" path="@{ROOT}\src" platform="All"/>
    <propathentry env="gui" kind="con" path="com.openedge.pdt.text.GUI_LIBRARIES"
platform="All"/>

    <propathentry env="tty" kind="con" path="com.openedge.pdt.text.TTY_LIBRARIES"
platform="All"/>
    <propathentry env="all" kind="con" path="com.openedge.pdt.text.DLC_PATHS"
platform="All"/>

   <propathentry env="all" kind="lib" path="@{ROOT}\lib\sports2000.pl" platform="All"/>


</propath>
```

The propath entries with `kind=src` denotes the source path for compilation. A compile task is created for each of these source entries. For more information, see Tasks.

The default location of this file is `<project-root-location>/.propath`. Though this can be configured, if required, to a custom propath file path by setting the property CUSTOM_PROPATH_FILE. To configure this property, check the How to configure a property section in the Prerequisites topic.

Build config file

The `build.config` file contains all the project configurations required for you to build a project. The `build.config` file allows you to move your project's property settings along with your project so that your development and integration environment uses the same configuration. You do not need to set up the configurations again.

Note: You can configure the `build.config` to provide a custom `build.config` file by using the `CUSTOM_BUILD_CONFIG_FILE`. For more information about configuring this property, see How to configure a property in the Prerequisites topic.

Note: The `build.config` file has a different name in a previous release, see FAQ about changes to PDSOE for more information.

This topic describes the parameters in the `build.config` file.

Note: Progress recommends that you do not modify the `build.config` file manually within Progress Developer Studio. All the parameters can be updated from the Progress Developer Studio user interface (**Project Properties**) with the exception of the following:

- `streamIO.enabled`
- `languages.list`
- `textSegGrow`
- `minSize.enabled`

You can update the `build.config` file manually to build the project outside of Progress Developer Studio.

The `database` and `compilableFileExtensions` parameters are not used with Progress Developer Studio.

For file portability, Progress recommends that you use variables instead of an absolute path for the following attributes:

- `'buildDir'`
- `'oeide.xrefXmlDir'`
- `'oeide.preCompileCallbackRoutine'`
- `'avm.wrkDir'`
- `'avm.avmOptions.tmpDir'`
- `'avm.avmOptions.assembliesDir'`

The following default variables are supported:

- **`ROOT`**—Project's root location
- **`WRKDIR`**—Working directory; set during installation

Note: The **system properties**, **system environment variables**, and **Eclipse variables** are also supported.

The variable should be used as `"${`*`variable-name`*`}"`, for example, `buildDir="${ROOT}/myBuildDir"`

Global parameters

Parameters	Default value	Description
`buildDir=''`	—	Specifies the project build directory. This directory contains the saved r-code files. If this field is blank, then the r-code files are saved in the same directory as the source files.

oeide parameters

The following table contains the IDE-specific configurations that are only used by Progress Developer Studio for OpenEdge.

Parameter	Default value	Description
`useSharedAVM=''`	`false`	This parameter specifies whether to use a shared ABL Virtual Machine (AVM) when building projects in Progress Developer Studio. This parameter uses a Boolean value to specify whether the project uses a shared or a dedicated AVM.
`oeideEvents=''`	`true`	This parameter uses a Boolean value to enable or disable oeide events. This parameter enables you to write procedures that use the ABL SUBSCRIBE statement to capture and respond to those events.
`useGlobalToolboxVD=''`	`false`	This parameter uses a Boolean value to enable or disable the global Visual Designer toolbox.
`addDefaultParams=''`	`true`	This paramter specifies whether to use the default startup parameters when starting the AVM. This parameter uses a Boolean value to enable or disable the passing of default AVM parameters.
`hideTTYConsole=''`	`true`	This parameter uses a Boolean value to enable or disable the project-specific run-time console.
`useProjectCompilerSettings=''`	`false`	This parameter uses a Boolean value to enable or disable the use of project-specific `strictOptions` under `compile`.
`saveRCode=''`	`true`	This parameter uses a Boolean value to enable or disable persistant r-code on compiling sources.
`xrefXmlDir=''`	—	This parameter specifies the output value of the `XREF-XML` option when `xrefXml.enabled='true'`. This parameter uses a string value to specify the path where cross-reference information is saved in an XML file.

Parameter	Default value	Description
`preCompileCallbackRoutine=''`	—	This parameter uses a string value to specify the path to an ABL file that is run before compilation. For example, this option can used to set `SECURITY-POLICY:XCODE-SESSION-KEY`.

AVM parameters

Parameter	Default value	Description
`wrkDir=''`	The project's root directory	This parameter uses a string value to specify the working directory in which the AVM starts.

AVM option parameters

The following table lists the AVM option parameters (defined in the `build.config` file as `avmOptions`).

Parameter	Default value	Description
`tmpDir=''`	The DLC working directory	This parameter specifies the temporary directory for the AVM run time (equivalent to the `-T` startup parameter option). This parameter uses a string value to specify the path.
`tty.enabled=''`	false	Flag to use `_progres` or `prowin` executables. For `_progres`, the value is `true`; otherwise, it is `false`. This parameter uses a Boolean value.
`startupParameters=''`	—	This parameter uses a string value to specify the startup parameters as a string.
`assembliesDir=''`	—	Specifies the location of the `Assemblies` directory (equivalent to the `-assemblies` startup parameter option). This parameter uses a string value.

Compile parameters

Parameter	Default value	Description
`compilableFileExtensions=''`	`p`, `w`, `cls`, `pgen`, `html`, `htm`	This parameter allows you to set the compilable file extensions.

Compile option parameters

The following table lists the Compile option parameters (defined in the `build.config` file as `compileOptions`).

Parameter	Default value	Description
`multiCompile.enabled=''`	`false`	To enable the `COMPILER:MULTI-COMPILE` attribute, set the value to `true`.
`xcodeKey=''`	—	`XCODE` option in the `COMPILE` statement. Specify the expression as a string.
`xrefXml.enabled=''`	`false`	`XREF-XML` option in the `COMPILE` statement. To enable this option, set the value to `true`. Specify the expression as a Boolean value.
`streamIO.enabled=''`	`false`	`STREAM-IO` option in the `COMPILE` statement. To enable this option, set the value to `true`. Specify the expression as a Boolean value.
`minSize.enabled=''`	`false`	`MIN-SIZE` option in the `COMPILE` statement. To enable this option, set the value to `true`. Specify the expression as a Boolean value.
`attrSpace.enabled=''`	`false`	To enable the `ATTR-SPACE` option in the `COMPILE` statement, set the value to `true`.

Strict option parameters

The following table lists the Strict option parameters (defined in the `build.config` file as `strictOptions`).

Parameter	Default value	Description
`requireFullNames=''`	`ignore`	This parameter specifies the `OPTIONS require-full-names` in the `COMPILE` statement. Value can be `ignore`, `warning`, or `error`.
`requireFieldQualifiers=''`	`ignore`	This parameter specifies the `OPTIONS require-field-qualifiers` in the `COMPILE` statement. Value can be `ignore`, `warning`, or `error`.
`requireFullKeywords=''`	`ignore`	This parameter specifies the `OPTIONS require-full-keywords` in the `COMPILE` statement. Value can be `ignore`, `warning`, or `error`.
`requireReturnValues=''`	`ignore`	This parameter specifies the `OPTIONS require-return-values` in the `COMPILE` statement. Value can be `ignore`, `warning`, or `error`.

Languages parameters

Parameter	Default value	Description
`list=''`	—	Comma-separated list of language segments to include in the compiled r-code.
`textSegGrow=''`	—	This parameter specifies the `TEXT-SEG-GROW` option. The value must be an integer. Supported only when a language list is provided.

Tasks

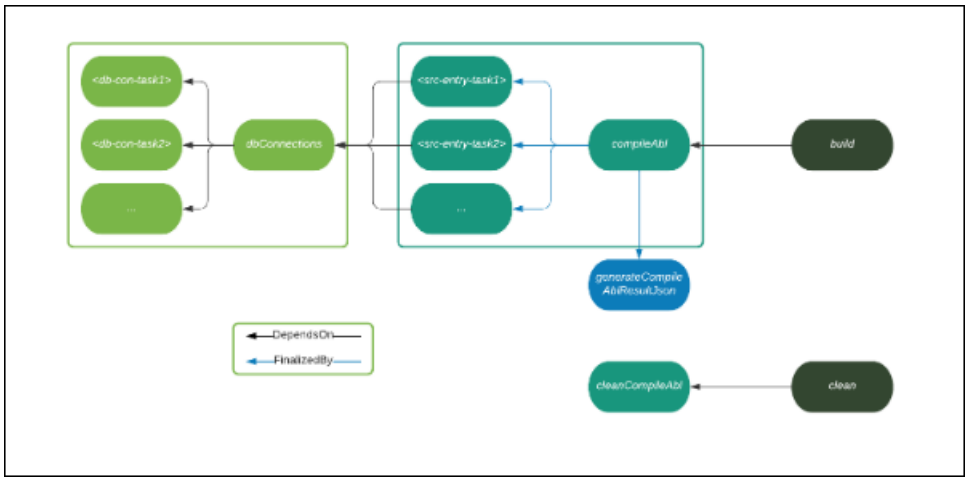
The `ABL Plugin` provides various tasks required to build an ABL project such as task to compile ABL sources by default, based on the project's propath (`.propath` file) and properties (`build.config` file).

You can refer to Understand the build process of an ABL project topic to understand in greater details about various stages of a project build.

By default, a project consists of the following tasks:

- Build
- compileAbl
- dbConnections
- clean
- cleanTaskName

This image shows the relationship between the tasks:



build (lifecycle task)

This task is used to build a project. Currently, the build task only compiles the project and generates the `r-code`. This task relies on the `compileAbl` task. In future, this should do all other necessary things needed for building the project.

compileAbl

The `compileAbl` task compiles an ABL project. It compiles the ABL sources for all the sourcepath entries defined in the `propath` file. This task has `finalizedBy` dependency on the *<src-entry-tasks>*.

<src-entry-tasks>

A task (of type `ABLCompile`) is created to compile ABL sources for each sourcepath entry (`kind=src`) defined in the `propath` file.

The name of the task follows the convention `compileAbl-root-<subpath-path-from-root>`. For example for a sourcepath, `@{ROOT}/src`, the task name would look like `compileAbl-root-src`.

When compiling the sources, you may prefer to configure these tasks for various options as described in Compile source code section.

Follow these steps to configure the task:

- AVM configurations
  - Use Propath file or refer to Dependency Management for providing PROPATH specific dependencies.
  - Use Build Config file to configure database connection details. A `dbConnections` task is created to hold these connection details. All these *<src-entry-tasks>* have `dependsOn` dependency on the `dbConnections` task.
  - Use Build Config file to configure AVM parameters.
- Compile configurations
  - Use Build Config file to configure various compile configurations.

Note: You can also use the global configurations, described in the ABL base plugin section, to configure various properties of the tasks. Though, you might not always have a requirement to do so.

---

generateCompileAblResultJson

Generates a unified result of the JSON output of different tasks. This is used when the property `outputType` is set in `compileOptions`.

dbConnections

Holds ABL database connection details that are defined in the config file. A task (of type `DBConnection`) is created for each connection entry provided in the Build Config file.

clean

Cleans the project by removing the build output. Run this task first if you want to re-build the project again in a clean environment.

cleanTaskName

Deletes files created by the specified task. For example, `cleanCompileAbl` deletes the `r-code` files and any other output files generated by the `compileAbl` task.

Dependency Management

This section describes how to manage dependencies when building an ABL project using the ABL plugin. Refer to Manage dependencies section to understand about various kinds of dependencies that you might want to handle.

You can handle PROPATH specific dependencies in the following ways:

1. Using Propath file (this approach can only handle dependencies that are present locally). Refer to the Propath file section for more details.
2. Using dependency configurations—The ABL plugin has contributed to Gradle's configurations using which you can manage your PL file dependencies both from local and from remote repositories.

Managing dependencies using dependency configurations is a two-step process—Declaring repositories, and Declaring dependencies.

Declaring Repositories

OEDF supports all types of repositories as supported by Gradle such as Maven, Ivy, and local (flat) directories. For more informations on how to configure repositories, see Gradle doc.

Example

```
//declare repositories

repositories {

 maven {

  url ='<url-to-maven-repository>'

  metadataSources {

   artifact()//needed to downloaded artifacts (like .pl file) without a pom file in the
 repository

  }

 }

}
```

Declaring Dependencies

Every dependency declared for a project applies to a specific scope. For example, some dependencies should be used for compiling source code whereas others may only need to be available for testing. Gradle represents the scope of a dependency with the help of a Configuration.

OEDF introduces different configurations for various types of dependencies for ABL projects. (Currently, OEDF has added only default tasks for compilation, so naturally, the scope of these configurations is limited to compilation).

These configurations support both Module dependencies and File dependencies. For more information, see Gradle doc. You can also configure the dependencies further as supported by Gradle. For example, to enable whether Gradle should always check for a change in the remote repository.

See Dependency configurations to handle PL dependencies for the list of configurations that should be supported in OEDF for ABL.

Dependency configurations to handle PL dependencies

This sections describes the list of configurations that should be supported in OEDF for ABL.

implementationAbl

The implementation only dependencies. Used at both compile-time and runtime.

Example

```
//define the dependencies

 dependencies {

  implementationAbl(<group>:<module-name>:<version>:<classifier>@pl')

}
```

compileOnlyAbl

Dependencies used only for compilation (not used at runtime).

Example

```
//define the dependencies
 dependencies {
  compileOnlyAbl(<group>:<module-name>:<version>:<classifier>@pl')
}
```

compilePropath

Extends `compileOnlyAbl` and `implementationAbl`. Compile propath that is used when compiling sources. Used by the `compileAbl` (*<src-entry-tasks>*)task.

Manage dependencies and propaths from `.propath`

The propaths from `.propath` file is added first, followed by the dependencies coming from the dependency configurations. Priority is given to the dependencies coming from the Propath file.

configurationNameSource

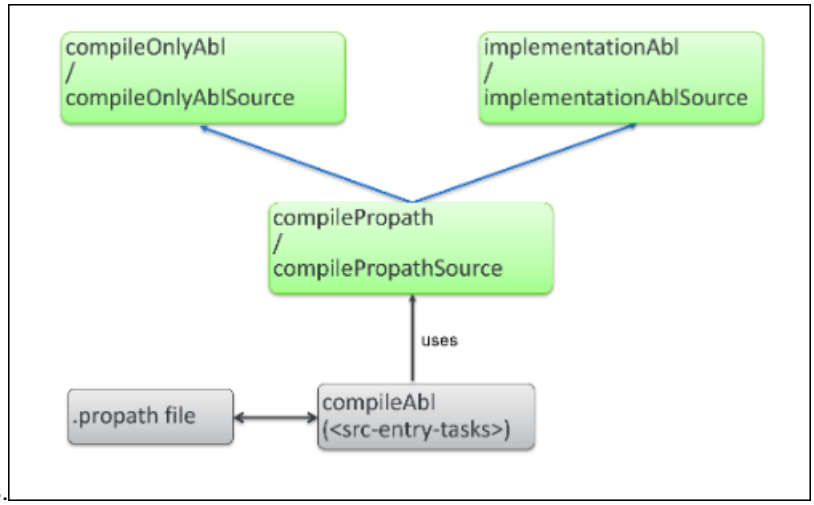
For each of the above dependency configurations there is one dependency configuration added to handle `PL` file dependencies with sources (see Procedure Library (PL) dependencies in PROPATH for more information.

Such `PL` file dependencies are treated similar to their counterpart, except that the `PL` file dependencies defined by this configuration is extracted using `ExtractPLTransform` transform and then added to the Propath.

Example

- `implementationAblSource` for `implementationAbl`.
- `compileOnlyAblSource` for `compileOnlyAbl`.
- `compilePropathSource` for `compilePropath`.

This illustration shows the relations between dependency configuration and tasks.



Sample Example

```
plugins {
id "progress.openedge.abl"version "<version>"
}

//declare repositories
repositories {
 maven {
  url ='<maven-repo-url>'
  metadataSources {
   artifact() //needed to downloaded artifacts without pom file
  }
 }
}

 //define the dependencies
 dependencies {
  implementationAbl(<group>:<module-name>:<version>:<classifier>@pl')
   implementationAblSource(<group>:<module-name>:<version>:<classifier>@pl')
   compileOnlyAbl(<group>:<module-name>:<version>:<classifier>@pl')
   compileOnlyAblSource(<group>:<module-name>:<version>:<classifier>@pl')
```

In this example, the dependencies defined by `implementationAbl` and `compileOnlyAbl` are added to the Propath directly as `PL` files whereas dependencies defined by `implementationAblSource` and `compileOnlyAblSource` will be extracted first and the location of the extracted directory will be added to the Propath.

Migrate from Apache Ant

Many ABL applications make use of Apache Ant and PCT. Apache Ant is a build automation tool that uses XML and Java, and it was designed to make build processes simpler and more portable. Ant uses targets and tasks to compile, test, and run applications. It is primarily used for Java applications, but can be used for other types of applications as well. PCT is an open source set of Apache Ant tasks and plugins created by Riverside Software that automate work in OpenEdge. PCT is a large ecosystem of plugins that provides common build steps for automation and CI/CD.

Why migrate from Ant to Gradle?

Migrating from Ant to Gradle has the following benefits:

- Gradle uses a domain-specific language, which allows users to add logic to the build pipeline.
- Gradle has reusable tasks and can reuse other parts of a build pipeline, which makes the build script less verbose, more readable, and easier to maintain.
- Gradle provides smarter dependency management than Ant.
- Gradle make builds modular, which allows larger projects to be built from interdependent parts. You can even build multiple projects from the same pool of parts.

Use ant.importBuild

The Gradle task `ant.ImportBuild` is a straightforward and quick way to migrate your build processes from PCT or Ant to Gradle. This task converts Ant targets in your Ant scripts into Gradle tasks. To use `ant.ImportBuild`:

1. Add the following to your `build.gradle` file in your project:

   ```
   ant.ImportBuild 'build.xml'
   ```

   This assumes `build.xml` contains your previous Ant tasks.

2. After saving your `build.gradle` file, run your previous Ant tasks using the Gradle wrapper:

   ```
   gradlew ant target
   ```

Using `ant.ImportBuild` works quite well as an initial phase of migrating to Gradle, but it is not a full migration because you still need to maintain your existing Ant scripts.