# Installer Support Gradle Plugin (ISP) version 0.6.10

This page will attempt to document the architecture of the "new" product installers. The product installers project was undertaken in order to:

1. Support automated creation of installers (from a Jenkins job, for example)
2. Provide a mechanism where consortium members can create their own installers, but have them use a standard "base" Raptor installer
3. Support better control mechanisms on the creation of the base Raptor installer
4. Simplify, yet again, the mechanisms used to configure an installer (even with the prior "generic" install4j script, it was getting cluttered)
5. Support the creation of product "variants" from a single installer project (e.g., Seraph and voltronAPI), minimizing duplication of resources
6. Reduce the repository bloat by storing large artifacts (like JRE bundles, WW Data, etc.) in Nexus and pulling them down during the build process (the old installers repo is ~5 GB and the new one is ~2 MB)

## Structural Overview

Each product's installers are managed in a separate repository with a top-level `build.gradle` script. That build script will apply this plugin, providing it a custom DSL with which to configure the installers(s) generated by the build. One significant improvement in the new structure is that all the "variants" for a given product (like Seraph vs. SeraphPluginAPI) are now managed from a single product-level project. Each variant is configured separately in the build script, but more on this below.

Previously, every product-level build created a complete Raptor `bin` directory and directly modified files such as `conf/DefaultProgramProperties.xml`. This is no longer the case. The `RaptorInstaller` project is the only project that constructs the full bin directory. The product-level builds provide overlays to augment that standard content. The `RaptorInstaller` project creates a complete set of stand-alone installers (one per platform) that will perform the equivalent of the prior "`pluginAPI`" installer project (in that it installs as "RaptorX" and contains a standard plugin load-out).

Each product-level project creates a set of installers, but they are "add-on" installers, and will be used to modify the content laid down by the Raptor Base installer (at install time, not at installer creation time). These add-on installers are generally NOT directly distributed. The product-level installer creation generates a set of "*Bundle*" installers that collect the Raptor Base, the add-on installer, and appropriate configuration files into a single installer that the end-user can execute. The Bundle installer extracts its content, and then executes the Raptor Base installer (which tailors its UI content according to the configuration files, such that it will say things like "`Installing KeyMaker x.y.z`" instead of "`Installing RaptorX a.b.c`"), and the Raptor Base installer invokes the add-on installer at the end of its execution. The entire process results in a single installation using the `ApplicationId` specified by the add-on installer configuration files.

## How to Apply This Plugin

As of Gradle 5, the public gradle plugin portal is automatically searched for plugins, so no 'buildscript' configuration is required. Simply specify the plugin in the 'plugins' section, like this:

```
plugins {
    id 'gov.raptor.gradle.plugins.installer-support' version '0.6.8'
}
```

## What is Built

So, what happens when you run "`gradlew build`" in an installer project? Currently, the build tasks just construct what will be used to create the installers, but the installers are not created by default. Additional tasks (described below) are used to create and publish the installers.

For RaptorInstaller (the base installer):

1. The scripts and artifacts published by the Raptor build (the bin creator artifacts) are used to construct the complete '`bin`' directory contents (within the '`build`' directory).
2. The configuration file processors are executed to transform files like `DefaultProgramProperties.xml`, etc.
3. The contents of the '`overlay`' directory are copied on top of that. Currently, all that does is lay down a custom help button configuration.
4. The deployed plugins (as defined in the build script) are copied into 'bin/plugins'.

For a product project (like Voltron):

1. The contents of the '`overlay`' directory are copied to '`build/overlay`'. This is the common, or shared, overlay that is applicable to all variants.
2. The deployed plugins (as defined in the build script) are copied into 'overlay/plugins'. These are the plugins common to all variants.
3. For each variant defined, the following operations are performed:
    a. The contents of the '`<variant>-overlay`' directory are copied to '`build/<variant>-overlay`'.
    b. The deployed plugins (as defined in the build script) are copied into '`<variant>-overlay/plugins`'

## Making Installers

Once the content has been built, installers can be compiled.

***NOTE:*** You must have install4j installed and have the `INSTALL4J_HOME` environment variable properly configured in order to actually create installers. If you do not have INSTALL4J_HOME set, you will see an error message like "`No INSTALL4J_HOME found, Install4j use will not be configured!`". If you have it mis-configured, various installer creation tasks will fail.

### For RaptorInstaller

There is only a single command needed (since it has no variants and it is not bundled with anything else).

```
gradlew makeInstallers
```

Will perform the build tasks described above, and then it will invoke the install4j compiler using the `raptorinstaller.install4j` script as input. The compiled installers are placed into '`build/installers`'. The installers have names very similar to the old installers, with slight changes to ensure that they can be published into maven local. For example, the windows x64 installer is named `RaptorX-windows-x64-2.22.1-SNAPSHOT.exe`, assuming the current Raptor version is 2.22.1-SNAPSHOT. Once compiled, they are also published into the local maven repository.

The install4j compilation is controlled using compiler variables specified in the '`product.properties`' file (very similar to the old model).

### For product installer projects

Product installers are produced in 2 steps. First, the 'add-on' installer is created from the product's install4j script, and then *bundled* installers are created using the `BundleInstaller.install4j` script included with this plugin. The bundle installers contain the Raptor Base installer, a `product.properties` file, and the add-on installer for the product. Since a product project can have variants, a number of tasks are created (using Seraph as an example, which has 2 variants, named '`seraph`' and '`voltronApi`'):

| Task | Description |
|---|---|
| `makeVoltronApiInstallers` | Compile the installers for the '`voltronApi`' variant, installers are written to '`build/voltronApi-installers`', compiler variables are specified in '`voltronApi-product.properties`' |
| `makeSeraphInstallers` | Compile the installers for the '`seraph`' variant, installers are written to '`build/seraph-installers`', compiler variables are specified in '`seraph-` |

| | |
|---|---|
| | `product.properties`' |
| `makeAllInstallers` | Placeholder task that depends on all the variant '`make<`*`variant`*`>Installers`' tasks |
| `makeAllVoltronApiBundles` | Compile the bundle installers for the `voltronApi` variant, bundle installers are written to '`build/voltronApi-installers`' |
| `makeAllSeraphBundles` | Compile the bundle installers for the `seraph` variant, bundle installers are written to '`build/seraph-installers`' |
| `makeAllBundles` | Placeholder task that depends on all the variant '`makeAll<`*`variant`*`>Bundles`' tasks |

So, to build everything, the command would be:

```
gradlew makeAllInstallers makeAllBundles
```

To build just a single variant, it would be something like:

```
gradlew makeVoltronApiInstallers makeAllVoltronApiBundles
```

## Selecting a Target JDK

By default, the installer will be constructed with a JRE based on JDK 8. If another JDK is to be bundled with the installers, then the consuming build must set the `JDK` project property. This property *must* be set before this plugin is applied to the project. Currently, the `JDK` property can be set to '8' or '11'.

Setting the JDK to anything other than '8' will change the Nexus Group Id under which the artifacts are published to include a suffix of '-jdk[*JDK*]'. So, specifying `JDK = '11'` results in a group id of `gov.raptor.installers-jdk11`.

### Bundling a Custom JRE

Bundling a custom JRE with the generated installer requires a bit more work. Specifically...

1. Create JRE bundles for each platform. [More info can be found here.]()
2. Publish the JRE bundles to Nexus in the `gov.raptor.installer-support.jrebundles` group in the [Private_RAPTORX_Third_Party_External]() repository. When publishing, make sure that the artifact type is `tar.gz`, otherwise the resources will not be found.
    - `windows-amd64`
    - `macosx-amd64`
    - `linux-amd64`
3. Use the `-PJRE_BUNDLE="VALUE"` format to set the correct version and make sure that is the version used when the debug information is printed out at the beginning of the run.

Here is an example that uses the 1.8.0_252 JRE...

```
./gradlew clean -PJRE_BUNDLE="1.8.0_252" makeVoltronApiInstallers makeAllVoltronApiBundles
```

As you can see, the JDK does not need to be explicitly stated as the parsing of the `JRE_BUNDLE` correctly identifies it as JDK8.

## Install4j Scripts and Compilation Control

As before, we use install4j to produce our installers. The previous `Generic.install4j` script helped to reduce the amount of copy/paste problems with having stand-alone installer scripts for each product. However, that approach was starting to show some wrinkles given the diverse needs of all the products (especially KeyMaker). The primary "common" sections of the installer were asking the user where to install, where to ProjectData, and the basic handling of the 'bin' directory (such as locating and extracting the proper Postgres archive). The new structure uses a layered approach. The Raptor Base installer now performs the majority of the work that was being done in the old Generic script. It configures its behavior (primarily in the content of the dialogs that interact with the user, as mentioned above).

The i4j scripts are converted into platform-specific executables via a compilation step controlled by the gradle build script. The compilation process is controlled by a set of variables, called *compiler variables* by i4j, and they control such things as the applicationId, the short and long names of the application, which in turn determine the name of the generated executables, among many other things. In addition to the predefined compiler variables, build-defined variables can be provided to the compiler, and referenced in the script. These variables are specified in a file called '`product.properties`'. Here is the product.properties file from the RaptorInstaller project:

**RaptorBase product.properties file**

```
# Set 'raptorBaseInstall' so the installer can react accordingly - no other installer should set this to true
raptorBaseInstall=true

# Set the default app Id to be the historical Plugin API app id
applicationId=3851-7797-4090-8717
applicationFullName=RaptorX
applicationShortName=RaptorX
executableName=RaptorX
releaseVersion=${project.programVersion}

retainAcmePlugins=true
```

The details for each entry are covered below, but take note that the file supports build variable expansion, so values declared in the build script can be referenced in the product properties. Note also, that the '`product.properties`' file is included with the product bundle installers, and the variables are available during installer execution (which allows them to override the predefined values set during compilation), but more on that later. The table below summarizes how each variable is used in the RaptorInstaller i4j script.

| **Variable** | **Use within the script** |
|---|---|
| `raptorBaseInstall` | Boolean value to directly indicate that this is a stand-alone (non-product) Raptor install. Controls whether the base script looks for and attempts to execute an add-on installer. |
| `applicationId` | The unique Id for this install. We retain the use of the UUID for PluginAPI so that the base installer will continue to locate previous PluginAPI installs.<br><br>Other projects have been using a combination of shortname and version as their Id. |
| `applicationFullName` | The full, or long, name for the application. Typically includes the version. Used in various dialogs presented to the user during installation. |
| `applicationShortName` | The short name for the application. Controls the naming of the generated installers as well as the maven artifact id (for publishing installers). |
| `executableName` | Specifies the name of the executable to create during installation. |
| `releaseVersion` | The version of the application. Controls the naming of the generated installers as well as the maven artifact version (for publishing installers). |
| `retainAcmePlugins` | Boolean value. If false, then the acme plugins are removed from the plugins directory during installation. |

The Raptor Base installer determines that it is acting as a product installer when it detects a file named '`product.properties`' in the same directory as the installer executable. It reads this file and internalizes the properties defined there. One of the key adaptations the base installer makes is that it will change its *applicationId*, which is the unique identifier that i4j uses to determine where a product was last installed and if an install is an upgrade. There are several additional variables that control runtime behavior that are specifically aimed at giving the add-on installer control of the Raptor Base installer when it is initiating a product-level install (i.e., not a standalone Raptor install). Here's a sample product.properties file for an add-on installer (from KeyMaker Web).

**KeymakerWeb-product.properties**

```
applicationId=KeyMaker
releaseVersion=$project.programVersion
applicationShortName=KeyMakerWeb
applicationFullName=KeyMaker-$project.programVersion
executableName=KeyMaker

selectGraphicsPower=false
createDesktopLinkAction=false

backup.projectdata.1 = ProjectData/Globals/Files/gov.raptor.webserver/security/store.xml
backup.projectdata.2 = ProjectData/Globals/Files/gov.raptor.provisioning/ProvisionedDevices.csv
```

The table below summarizes how each variable is used in the RaptorBase i4j script.

| **Variable** | **Use within the script** |
|---|---|
| `executableName` | Specifies that instead of creating `RaptorX.exe` and `RaptorXConsole.exe` (on windows), the executables would be `KeyMaker.exe` and `KeyMakerConsole.exe`. |
| `selectGraphicsPower` | Boolean value, if true (the default) the user will be presented a screen asking them to choose the "graphics power" of their machine (low, medium, or high).<br>If false, the user will not be given this choice and the rendering limits will be set to "medium". |
| `createDesktopLinkAction` | Boolean value, if true (the default) the installer will attempt to create a desktop link (except on Mac where it creates a dock entry).<br>If false, then no link will be made. False would be typical for headless environments. |
| `backup.projectdata.N` | This family of variables allows the add-on installer to request that a collection of files and/or directories be saved prior to deleting the previous installation contents.<br>The specified files/directories will be placed in a temporary location, and that location will be provided to the add-on installer in the runtime variable `backupTempDir`.<br>It is the responsibility of the add-on installer to process them as needed (such as restoring them to their original location).<br>The specified file paths are interpreted relative to the previously selected ProjectData dir (the directory above ProjectData). |
| `backup.installer.N` | Similar to `backup.projectdata.N`, but the files/directory paths are expected to be relative to the installation directory. |

## Product-level Build Scripts

The product-level build scripts are very similar to the previous model, in that they configure an overlay that will be laid down on top of the Raptor Base installation. The new feature is that all variants of the installer are managed by a single build script instead of requiring multiple sub-projects. Here's an example (taken from the Voltron build.gradle script):

**Voltron build.gradle script**

```
    buildOverlay {
        // Configure the documentation files to include
        includeDocumentation 'GRG_Manual'
        includeDocumentation 'High_Low_Manual'
        includeDocumentation 'RaptorX_Supported_Formats'
        includeDocumentation 'RaptorX_Technical_Appendix'
        includeDocumentation 'Route_Planning_Manual'

        includeDocumentation 'Seraph_User_Manual_2.2'
    }

    installer('voltronApi') {
    }

    installer('seraph') {
        // Configure the documentation files to include
        includeDocumentation 'Sandbar_Help_File'
        includeDocumentation 'Sandbar_Technical_Appendix'
        includeDocumentation 'DPACS_Plugin_Instructions'
        includeDocumentation 'LAREDO-X_Operators_Manual'
        includeDocumentation 'LAREDO-X_TRAINING_PPT'

        doLast {
            // We need to extract the Sandbar data folder into our target directory
            extractFromPluginJar 'seraphDeployedPlugins', 'plugin.sandbar', 'data/**'
        }
    }
```

The primary '`buildOverlay`' task is configured to produce the overlay content that is common to ALL variants. Anything copied/generated there will be the basis from which the variant-specific content is generated. In the case above, several documents are copied into the overlay. Then, each installer variant is defined using an '`installer`' configuration (which has access to all the same supporting methods as the `buildOverlay` task). In the example above, nothing additional happens for the '`voltronApi`' variant, but the '`seraph`' variant includes several additional documents and it extracts some data from the sandbar jar. Note that that extraction references the configuration named '`seraphDeployedPlugins`'. See below for details on how to specify the plugins to include in the installer and the configurations created. Each overlay (the base plus one for each variant) is constructed independently in the '`build`' directory, named '`overlay`' and '`<`*`variant`*`>-overlay`'.

## Helper Methods Available

A number of helper methods are added to the `buildOverlay` tasks and `installer` method as part of the DSL provided by this plugin. All of these methods (with the exception of `includeDocumentation`) should be used in a task *action*, not configuration. Meaning, they would be used in a `doLast` block. The table below describes each of these methods.

| **Method** | **Description** |
|---|---|
| `includeDocumentation(String name, String version = '0.0.1', String ext = 'pdf')` | Configures a documentation file to be pulled from nexus and copied into the overlayBuildDir dir.<br><br>Note that unlike all the rest of the methods here, this one is used in the configuration of the task, not an action on the task. The buildOverlay task automatically processes any configured documentation files and copies them into the overlayBuildDir. All documentation artifacts are assumed to reside in the group '`gov.raptor.installer-support.documentation`', resulting in an artifact dependency of the form `"gov.raptor.installer-support.documentation:$name:$version@$ext"`.<br><br>`@param name The base name of the documentation artifact`<br>`@param version The version of the artifact to retrieve, defaults to '0.0.1'` |

| | |
|---|---|
| | `@param ext The file extension, defaults to 'pdf'` |
| `expandZip(zipFile, String destDir = '.')` | Causes the referenced ZIP file to be expanded into the overlay directory being constructed. `destDir` defaults to '.', which is the target overlay directory.<br><br>`@param zipFile The ZIP file to expand`<br>`@param destDir The destination directory within overlayBuildDir into which to expand the zip` |
| `nexusArtifact(String configurationName, String artifact, includeTransitives = false)` | Define a Nexus hosted artifact that this installer needs to pull down, typically so it can be extracted into the build overlay.<br><br>`@param configurationName The name of the configuration to create`<br>`@param artifact The artifact coordinates, e.g., 'gov.raptor.installer-support:PRU_Utils:1.0.0@zip'`<br>`@param includeTransitives Pass true to include transitive dependencies in the configuration, default is false` |
| `expandNexusArchive(String configurationName, String destDir = '.')` | Causes the Nexus artifact from the named configuration to be downloaded and expanded into the overlay directory being constructed. `destDir` defaults to '.', which is the target overlay directory.<br><br>`@param configurationName The name of the configuration (as specified in nexusArtifact)`<br>`@param destDir The destination directory within overlayBuildDir into which to extract files` |
| `copyNexusArchive(String configurationName, String destDir = '.')` | Causes the Nexus artifact from the named configuration to be downloaded and copied (not expanded) into the overlay directory being constructed. `destDir` defaults to '.', which is the target overlay directory.<br><br>`@param configurationName The name of the configuration (as specified in nexusArtifact)`<br>`@param destDir The destination directory within overlayBuildDir into which to copy the artifact` |
| `extractFromPluginJar(String configurationName, String pluginName, includes)` | Extract files from the specified plugin's jar file into the overlay directory being constructed. This method just invokes the `extractFromPluginJarToDir` method below with a `destDir` of '.'. |
| `extractFromPluginJarToDir(String configurationName, String pluginName, String destDir, includes)` | Extract files from the specified plugin's jar file into the overlay directory being constructed.<br><br>`@param configurationName The name of the configuration containing the plugin dependency`<br>`@param pluginName The name of the plugin (like 'plugin.sandbar')`<br>`@param destDir The destination directory within overlayBuildDir into which to extract matching files`<br>`@param includes The include patterns to match files within the jar, e.g., 'data/**'` |
| `extractCustomHelpButtonIcon(configuration, pluginName, includes)` | Extract files from a plugin's jar file into the `'overlayBuildDir/Custom Help Buttons/icons'` directory. See the `extractFromPluginJarToDir` method above, which is called with a `destDir` of `'Custom Help Buttons/icons'`.<br><br>This is a simple shortcut method that invokes `extractFromPluginJarToDir configuration, pluginName, 'Custom Help Buttons/icons', includes` |

## Specifying The Plugins to Install

The specification of which plugins to install is made within the build script's using `ext` variables, and allows for a "common" configuration (shared by all variants) as well as variant-specific configurations. The build script fragment below shows an example of both common and variant-specific configurations):

```
    // Now configure our deployed plugins
```

```
ext {
    voltronPluginVersion = '2.4.8-SNAPSHOT'
    sandbarPluginVersion = '2.23.1'

    // A map from artifact coordinates to the version to include
    pluginVersions = [
            // Raptor Plugins
            'gov.raptor.plugins:plugin.freedraw'                  : '1.0.1',
            'gov.raptor.plugins:plugin.grg'                       : '1.2.6-RC2',
            'gov.raptor.plugins:plugin.highlow'                   : '1.1.1',

            // Voltron Core Plugins
            'gov.raptor.seraph.core:plugin.seraph.api'            : voltronPluginVersion,
            'gov.raptor.seraph.core:plugin.seraph.devices'        : voltronPluginVersion,
            'gov.raptor.seraph.core:plugin.voltron.client'        : voltronPluginVersion,

            'gov.raptor.plugins:plugin.sandbar'                   : sandbarPluginVersion,
    ]

    voltronApiPluginVersions = [
            'gov.raptor.seraph.core:plugin.voltron.vdt'            : voltronPluginVersion,
            'gov.raptor.seraph.plugins:plugin.voltron.icarus'      : voltronPluginVersion,
            'gov.raptor.seraph.plugins:plugin.voltron.voltrondemo': voltronPluginVersion,
    ]
}
```

Each plugin configuration is specified as a map using the group and artifact Id as the key and the version as the value. With these 3 pieces, the GAV coordinates are used to create dependencies used to download the plugins from Nexus and include them in the installers. The common set of plugins (to be included in all variants) is defined in the map named `pluginVersions`, and the variant-specific configuration are defined in maps named `<variant>PluginVersions`.

Since each variant can reference different plugins, multiple configurations are created, one for the common overlay, named '`deployedPlugins`' (built with dependencies generated from the `pluginVersions` map) and one for each variant (as defined by the '`installer`' configurations) named '`<variant>DeployedPlugins`' (built with dependencies generated from the `<variant>pluginVersions` map).

For each installer configured, an artifact named `<variant>-plugins.bom` containing the plugin dependencies will be published under the coordinates group='`gov.raptor.installers`', artifact='`<variant>-plugins`', and version=`<projectVersion>`. This file can then be read by other projects (initially RaptorT) to configure dependencies allowing the complete set of plugins to be resolved and pulled in for processing (such as installing into the Raptor/bin/plugins directory).

## New Requirements for add-on Installers

Previously, since each installer was a "monolithic" installer, and included the complete content of the bin directory, the installer build scripts handled things like modifying configuration files (`DefaultProgramProperties.xml`, for example) before compiling the installer. With the new model, the base content is controlled by the RaptorBase installer and can't be modified during compilation of the add-on installer. These types of changes must now be performed during the execution of the add-on installer. The list below identifies the tasks that were previously handled prior to compilation, but which now must be performed by the installer at runtime.

1. There were a set of '*config processor*' scripts in the 'config' directory of each product installer sub-project. This processing has been moved into the i4j script for each product installer. I4j has several actions for modifying text and XML files, which makes this process fairly simple.
2. Moving the desired RibbonBarConfig file into ProjectData. Each installer must include the config files in their media and add an action to copy it to the required location in ProjectData.

## Changes to the 'bin' Directory

Previously, two launchers were created by the i4j installers: a GUI launcher and a Console launcher. The GUI launcher was an executable generated by i4j during compilation. However, the Console launcher was configured as an *external* launcher, and we just pointed it at the `raptor.bat` or `raptor.sh` file. This was problematic for a number of reasons, the main one being that we had to ensure the scripts work on a per-platform basis. In the new model, the Console launcher is also generated by i4j, but it is configured to run as a 'console' application, which provides the expected behavior (a console window and no redirection of the output, so it can be seen by the user). The `.bat` and `.sh` files are no longer distributed with the installer.