The definitive guide to install4j

Building professional installers on the JVM

Index

Introduction To Install4j

What is install4j?

install4j is a professional tool for building installers for multiple platforms, especially for applications that run on the Java Virtual Machine.

Main features that distinguish install4j are:

- **Flexible configuration of screens and actions**

  In your installers you can define your own flow of installer screens and installer actions [p. 24] to gather user input and initialize your installation with it. Configurable form screens [p. 46] allow you to create arbitrary forms that work in GUI and console mode [p. 195]. A rich set of configurable actions handles a variety of tasks and is extensible with the API [p. 212].

- **Generation of native launchers**

  install4j generates native launchers for console, GUI and service executables [p. 97]. These launchers offer variety of features such as flexible module and classpath configuration, version-specific VM parameters [p. 84], icons, splash screens and much more. At runtime, there is launcher API [p. 216] that interacts with some of these feature and with the variable system of the installer.

- **Auto-update functionality**

  The requirements for automatic updates [p. 114] are very individual, so install4j offers a template-base mechanism for update-downloaders. Update downloaders are fully configurable installer applications with their own flow of screens and actions, that can handles interactive auto-update, mandatory auto-update at startup and background update.

- **Bundling of Java Runtime Environments**

  Bundling a Java runtime [p. 89] is made easy with the pre-build JRE bundles and the bundle creation tools in install4j. JRE bundles can also be downloaded on the fly if no JRE installation is found.

The install4j UI is delivered as a desktop application. Building installers is not only possible in the IDE, but also with the command line compiler [p. 220] as well as the plugins for Gradle [p. 225], Maven [p. 230] and Ant [p. 239].

How do I continue?

The "Concepts" section is intended to be read in sequence, with later help topics building on the content of previous ones. The sections at the end are optional readings that should be consulted if you need certain features.

We appreciate your feedback. If you feel that there's a lack of documentation in a certain area or if you find inaccuracies in the documentation, please don't hesitate to contact us at support@ej-technologies.com.

A Concepts

A.1 Projects Overview

Project files

A project in install4j is the collection of all information required to build media files, the deliverables that can be distributed to the target platforms. A project is saved to a single XML file with an `.install4j` extension. Project files are platform-independent, you can open and compile them on any supported platform. The compilation step will produce the media files from the project definition. All paths that you enter in install4j are saved as absolute paths by default. This allows you to move the project file to a different location on your computer and the compilation will still work. If you wish to use your project file on multiple computers or platforms or compile your launchers with automatic build agents, it is more convenient to use relative paths. On the "General Settings->Project Options" step, install4j provides an option to convert all paths to relative paths when you save your project.

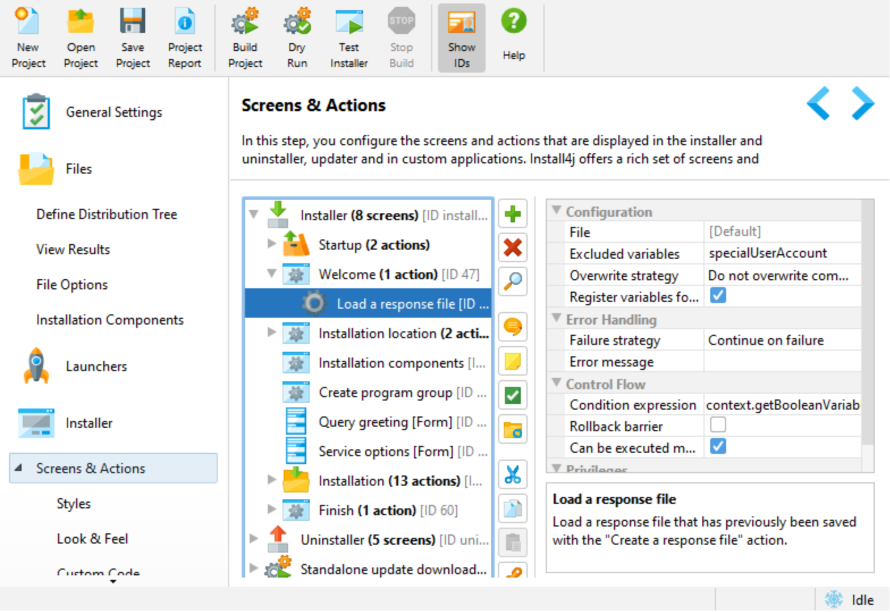
install4j keeps a list of recently opened projects under *Project->Reopen*. By default, install4j opens the last project on startup. This behavior can be changed in the preferences dialog by choosing *Project->Preferences* from the main menu. You can pass the name of a project file as a command line parameter to install4j to open it on startup. Also, the command line compiler [p. 220] expects the project file name as its argument.

Contents of a project

The following paragraphs give a high-level overview of the elements that you can configure in install4j. Each of the configuration sections in install4j as shown in the screenshots below represents a top-level concept in install4j.

Typically, a project defines the distribution of a single application. An application has an automatically generated application ID [p. 208] that allows installers to recognize previous installations.

At the core of the project definition is the sequence of installer screens and actions [p. 24]. They determine what the users see, what information they can enter and what the installer does. install4j offers a lot of flexibility regarding the configuration of of your installer. Besides creating traditional application installers, install4j is equally suited to create small applications that modify the target system in some way.


New Project
Open Project
Save Project
Project Report
Build Project
Dry Run
Test Installer
Stop Build
Show IDs
Help
General Settings
Files
Define Distribution Tree
View Results
File Options
Installation Components
Launchers
Installer
Screens & Actions
Styles
Look & Feel
Custom Code
Installer (8 screens) [ID install...
Startup (2 actions)
Welcome (1 action) [ID 47]
Load a response file [ID ...
Installation location (2 acti...
Installation components [I...
Create program group [ID ...
Query greeting [Form] [ID ...
Service options [Form] [ID ...
Installation (13 actions) [I...
Finish (1 action) [ID 60]
Uninstaller (5 screens) [ID uni...
Standalone update download...
Idle

Screens & Actions

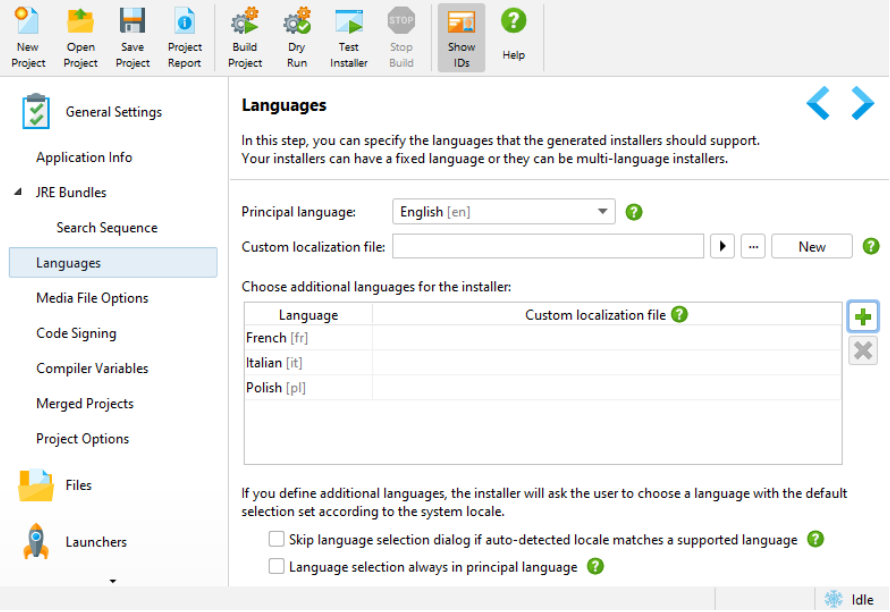
In this step, you configure the screens and actions that are displayed in the installer and uninstaller, updater and in custom applications. Install4j offers a rich set of screens and

Configuration	
File	[Default]
Excluded variables	specialUserAccount
Overwrite strategy	Do not overwrite com...
Register variables fo...	☑
Error Handling	
Failure strategy	Continue on failure
Error message	
Control Flow	
Condition expression	context.getBooleanVariab
Rollback barrier	☐
Can be executed m...	☑
Privileges	

Load a response file

Load a response file that has previously been saved with the "Create a response file" action.

The install4j runtime is localized into many languages. You can configure your installers to support one or multiple languages [p. 79].



Languages

In this step, you can specify the languages that the generated installers should support. Your installers can have a fixed language or they can be multi-language installers.

Principal language: English [en]

Custom localization file:

Choose additional languages for the installer:

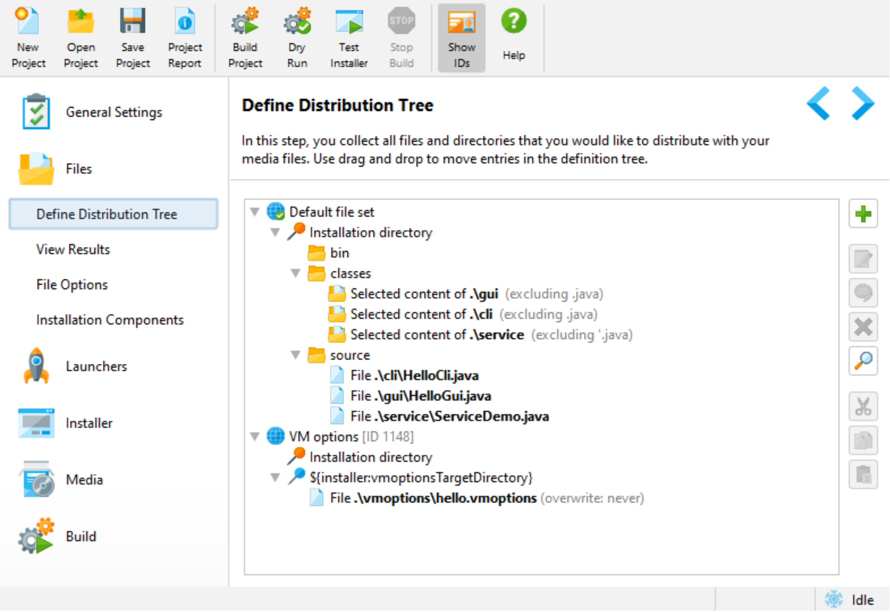
Language	Custom localization file
French [fr]	
Italian [it]	
Polish [pl]	

If you define additional languages, the installer will ask the user to choose a language with the default selection set according to the system locale.

☐ Skip language selection dialog if auto-detected locale matches a supported language

☐ Language selection always in principal language

Most installers install files to a dedicated directory and optionally to several existing directories on the target computer. That's what the "Files" section [p. 14] in the install4j IDE is for. Here, you define a "distribution tree", and optionally "installation components" which can also be downloaded on demand [p. 134].



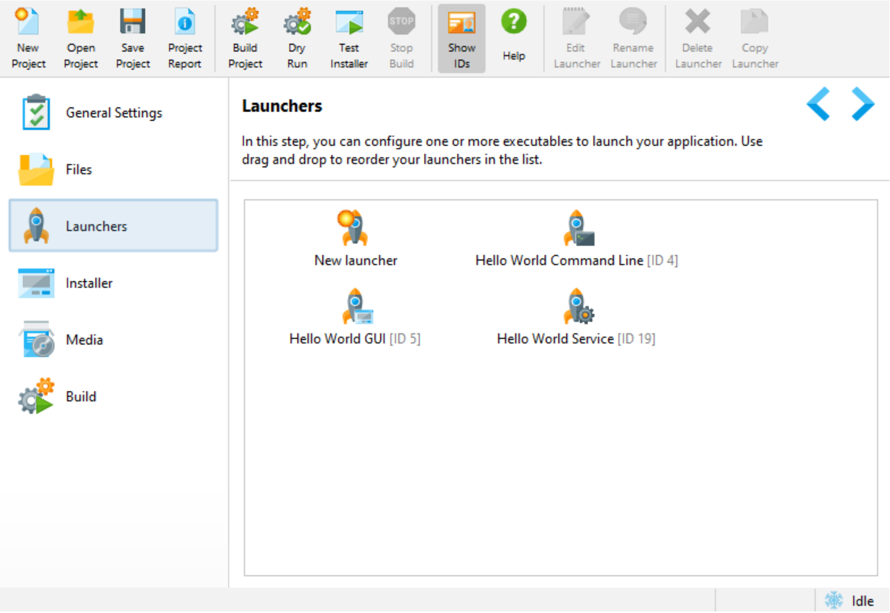
Define Distribution Tree

In this step, you collect all files and directories that you would like to distribute with your media files. Use drag and drop to move entries in the definition tree.

```
Default file set
  Installation directory
    bin
    classes
      Selected content of .\gui (excluding .java)
      Selected content of .\cli (excluding .java)
      Selected content of .\service (excluding *.java)
    source
      File .\cli\HelloCli.java
      File .\gui\HelloGui.java
      File .\service\ServiceDemo.java
VM options [ID 1148]
  Installation directory
  ${installer:vmoptionsTargetDirectory}
    File .\vmoptions\hello.vmoptions (overwrite: never)
```

The actual installation of these files is handled by the "Install files" action which is part of the default project template. If your installers should not install any files, you can remove that action and ignore the "Files" configuration section. When the "Install files" action is executed, it creates an uninstaller. The uninstaller offers the same flexibility as the installer and is configured in the same way.

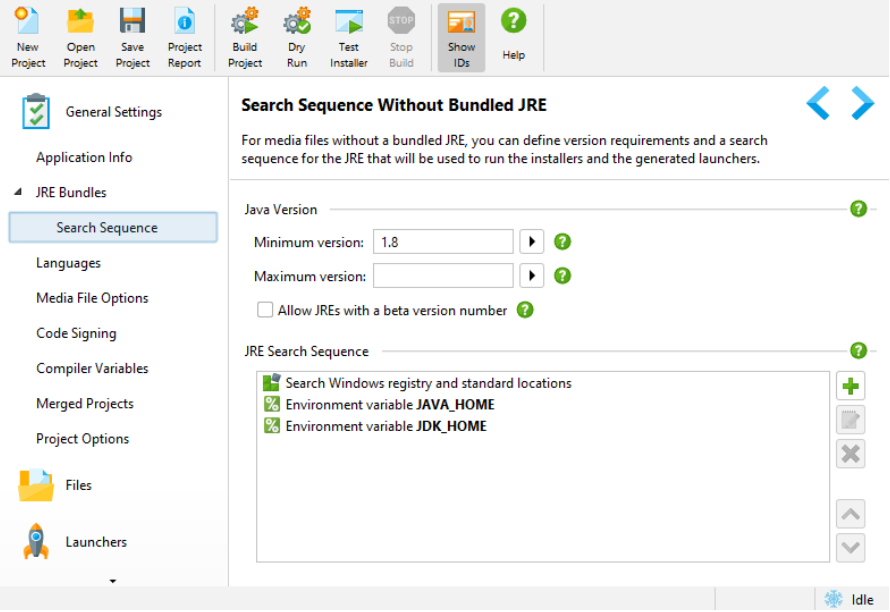
Unless the installed files are only static data, you will need application launchers to allow the user to start your application. You can define one or several application launchers in the "Launchers" section [p. 36]. Launchers generated by install4j have a rich set of configuration options including an optional splash screen or advanced features like a single instance mode. Configured launchers can also be "services" that run independently of logged-on users. install4j offers special installation screens and actions for services.



Launchers

In this step, you can configure one or more executables to launch your application. Use drag and drop to reorder your launchers in the list.

install4j has many advanced features concerning bundling of JREs or the runtime-detection of an installed JRE. Bundling of JREs [p. 89] is configured on the "JRE bundles" step and can be refined on a per-media file basis. If you do not wish to bundle a JRE, you define Java version constraints and a search sequence [p. 36] for both your installers and your generated launchers. In this way, you ensure that the launchers run with the same JRE as your installers.



Search Sequence Without Bundled JRE

For media files without a bundled JRE, you can define version requirements and a search sequence for the JRE that will be used to run the installers and the generated launchers.

Java Version

Minimum version: 1.8

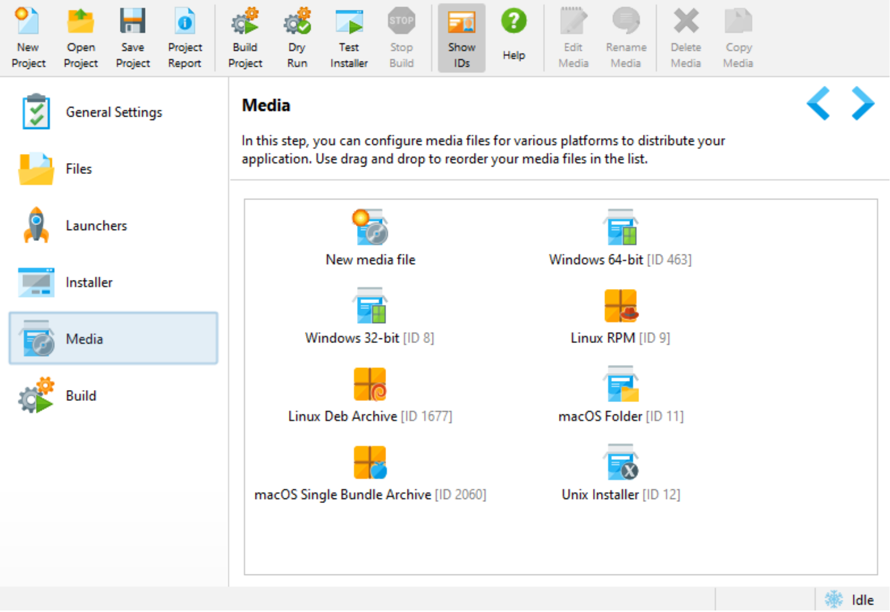
Maximum version:

☐ Allow JREs with a beta version number

JRE Search Sequence

- Search Windows registry and standard locations
- Environment variable **JAVA_HOME**
- Environment variable **JDK_HOME**

Finally, the media file definitions define the actual executables that you distribute. They capture platform-specific information and provide several ways to override project settings. You typically define one media file for each platform. Multiple media files for the same platform can be added as needed. Media files are either installers or archives. Archives simply capture the launchers and the distribution tree. They are a limited way to create a distribution and might not be suitable if you rely on the flexibility that is offered by installers.



Media

In this step, you can configure media files for various platforms to distribute your application. Use drag and drop to reorder your media files in the list.

- New media file
- Windows 64-bit [ID 463]
- Windows 32-bit [ID 8]
- Linux RPM [ID 9]
- Linux Deb Archive [ID 1677]
- macOS Folder [ID 11]
- macOS Single Bundle Archive [ID 2060]
- Unix Installer [ID 12]

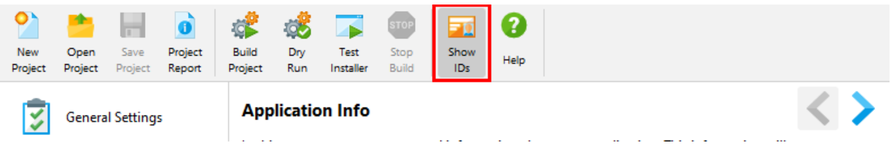
Project reports

install4j projects can become quite complex, especially the definition of the installer can be very hierarchical with hundreds of nested elements each of which may have important configuration in their properties. In order to check all your projects settings on a single page, or to print out your project definition, install4j offers a project report. The action to create such a report is available in the toolbar. When you generate a report, an HTML file is written to disk together with a directory named `install4j_images` that holds all referenced image files.

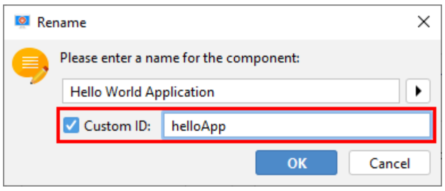
If you are looking for certain text value in a property or a particular piece of code in one of your scripts, use the search functionality in the browser when viewing the exported report to cover all parts of the project.

IDs of project elements

All elements in projects that can be referenced at runtime, like installation components, launchers, screens, actions, form components or media files have an automatically assigned ID. You can toggle the display of IDs globally in the tool bar. You may need to use IDs when using the API in scripts. Scripts are written in plain Java in a code editor provided by install4j.



If you would rather not reference automatically generated IDs in your scripts, you can specify your own custom IDs. Custom IDs can be assigned by using the "Rename" action for the selected element and selecting the "Custom ID" check box in the rename dialog. Custom IDs must not start with a number. The numeric internal ID is never discarded. If you disable the custom ID at a later point, the ID will be reverted to the previous numeric ID.

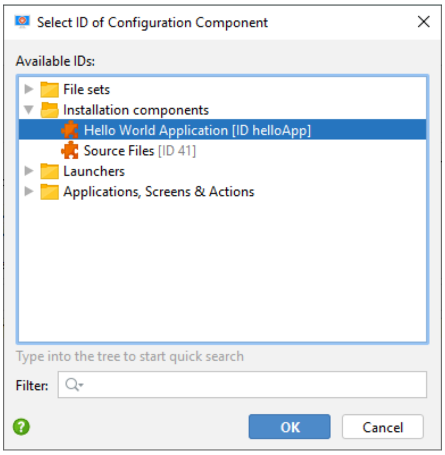


Please enter a name for the component:

Hello World Application

☑ Custom ID: helloApp

The "Insert ID" action in the script editor inserts custom IDs instead of the numeric IDs. All `get...ById()` methods in the API accept both the custom ID and the internal numeric ID. This means that you can set a custom ID without breaking anything in the project.


Select ID of Configuration Component
Available IDs:
File sets
Installation components
Hello World Application [ID helloApp]
Source Files [ID 41]
Launchers
Applications, Screens & Actions
Type into the tree to start quick search
Filter:
OK
Cancel

A.2 Building Projects

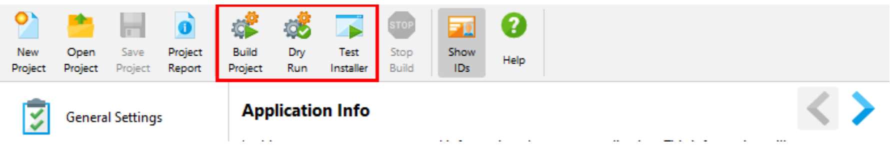
You can build a project from the IDE or from the command line. The command line compiler executable is `bin/install4jc` and takes the project name as an argument. On macOS, that directory is inside the application bundle and can be shown in the Finder with the "Tools" tool bar button. That same tool bar button also allows you to create symlinks for all command line tools in `/usr/local/bin` so they can be directly invoked in a terminal.

There are plugins for Gradle [p. 225], Maven [p. 230] and Ant [p. 239] for configuring the build in a way that is idiomatic for the respective build systems. In the end, all plugins invoke the command line compiler and for each command line compiler option there is a corresponding setting in the build system plugins.

When you start a build, install4j will check if all required information has been entered. If the build has been started from the install4j IDE, the corresponding step will be activated and the offending setting will be focused, so it is recommended to try out your builds in the IDE first.

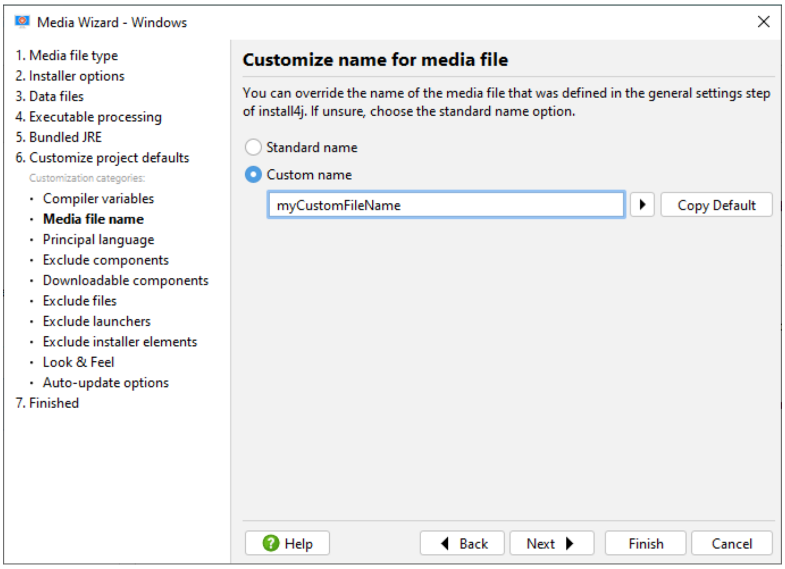
Build modes

There are three different build modes that correspond to different tool bar buttons in the install4j IDE or different command line options in the command line compiler.



When a **regular build** is started, the media files [p. 126] are built and placed in the media file output directory that is configured on the "General Settings->Media File Options" step.

Previous media files are overwritten, but a single build may not produce the same media file twice. On the "Customize project defaults->Media file name" step of the media wizard you can adjust the media file name if the global pattern resolves to the same name for multiple media files. You can also use a compiler variable [p. 63] for the media file output directory and override it for each media file to avoid name clashes.



Customize name for media file

You can override the name of the media file that was defined in the general settings step of install4j. If unsure, choose the standard name option.

○ Standard name
● Custom name

myCustomFileName

If you just want to check if the build will not produce any errors or warnings, you can start a **dry run**. The media files will be built in the temporary directory but not moved to their final location. For command line builds, use the `--test` option.

Building media files can take a long time, especially if you package a lot of files that have to be collected and compressed. To facilitate faster development, install4j offers to **build an installer incrementally**. The corresponding command line option is `--incremental`.

This build mode is intended for testing changes that you make in the installer configuration [p. 148] such as adding, removing or modifying screens, actions and form components.

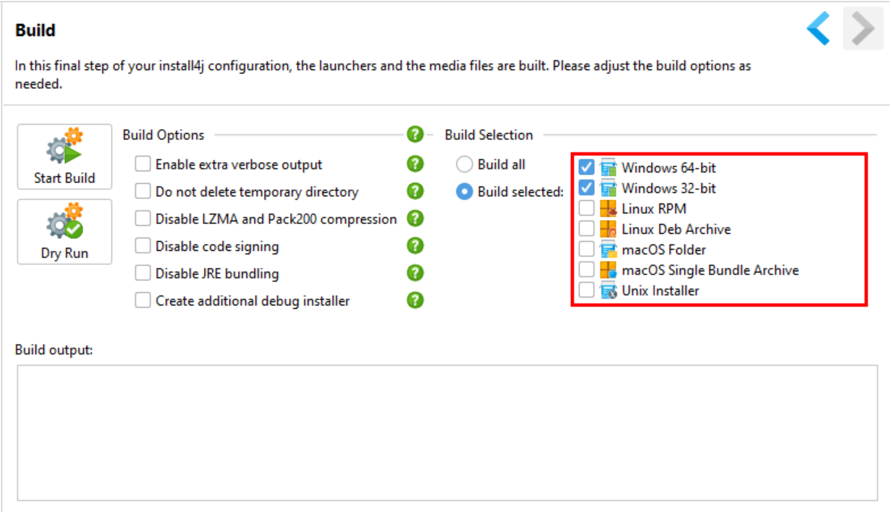
The action looks for the first media file in the "Media" step that can be run on the current platform and has an installer media file type [p. 126]. The media file must be already built, otherwise the action will terminate with an error message.

All scripts are recompiled and the installer configuration files are regenerated. The installed files are taken from the full build of the media file. If you change the definition of the distribution tree [p. 14] and expect to see these changes in the installer, you have to rebuild the media file with a regular build.

When the build is complete, the installer is started so you can try out your changes immediately. With respect to a full build, the compilation time is reduced substantially, typically to a couple of seconds. A full build can take several minutes, depending on the amount of files that are included and the selected type of compression.

Selective building of media files

Instead of building all media files, you can build only a subset by explicitly selecting the desired media files on the "Build" step.



Build

In this final step of your install4j configuration, the launchers and the media files are built. Please adjust the build options as needed.

Build Options

☐ Enable extra verbose output
☐ Do not delete temporary directory
☐ Disable LZMA and Pack200 compression
☐ Disable code signing
☐ Disable JRE bundling
☐ Create additional debug installer

Build Selection

○ Build all
● Build selected:

☑ Windows 64-bit
☑ Windows 32-bit
☐ Linux RPM
☐ Linux Deb Archive
☐ macOS Folder
☐ macOS Single Bundle Archive
☐ Unix Installer

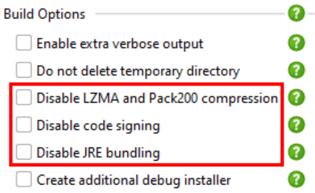
Build output:

This selection is persistent, but the command line build will still build all media files unless you pass the `--build-selected` option. This allows you to build a suitable media file in the IDE for testing without impacting the command line build on your build server.

To specify media files from the command line, pass the `--build-ids=ID[,ID]` or the `--media-types=T[,T]` option. IDs of media files are visible in the "Media" step if the "Show IDs" tool bar toggle button is selected. Selecting media files by their media type ID is useful if you build different media files on different platforms. The `--list-media-types` command line option prints the full list of supported media types and exits.

Faster builds during development

During development, you can speed up your build by compromising on the size of the produced media files. By switching off LZMA and Pack200 compression [p. 126], builds times can be reduced by 50% and more. By disabling JRE bundling, the generated installer will start up faster, because the JRE does not have to be unpacked. Finally, disabling code signing will prevent dialogs that ask for keystore passwords from being shown.



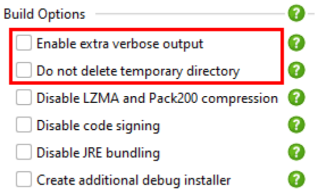
Build Options

☐ Enable extra verbose output
☐ Do not delete temporary directory
☐ Disable LZMA and Pack200 compression
☐ Disable code signing
☐ Disable JRE bundling
☐ Create additional debug installer

All these options for making builds faster are also available for the command line compiler, the corresponding options are `--faster` for disabling advanced compressions, `-disable-bundling` for ignoring JRE bundles and `--disable-signing` for building without code signing.

Trouble-shooting build failures

By default, basic progress information is shown in the build output and warning messages are highlighted. Any error will stop the build and the command line compiler will exit with a non-zero return code. For debugging purposes, there are two options that give access to more detailed information.



Build Options

☐ Enable extra verbose output
☐ Do not delete temporary directory
☐ Disable LZMA and Pack200 compression
☐ Disable code signing
☐ Disable JRE bundling
☐ Create additional debug installer

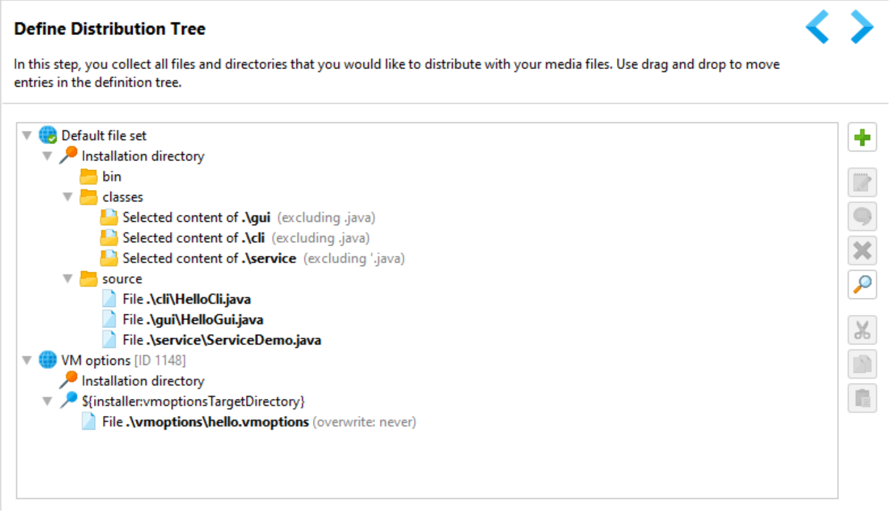
With the `--verbose` option, install4j prints more information about interesting events during the build. For example, all compiler variable replacements are shown in detail. If the source of an error message is not clear, switching on verbose mode can give you more context about the compilation phase that caused the failure. In addition, a compilation failure that occurs while verbose mode is enabled will print the entire stack trace to the build output.

Secondly, the install4j compiler prepares its artifacts in a temporary directory which is deleted after the build completes. With the `--preserve` option you can ask install4j to keep this temporary directory so that you can inspect intermediate artifacts.

A.3 Distributing Files

In the "Files" step of the install4j IDE, you define your distribution tree, collecting files from different places to be distributed in the generated media files. In addition, you can optionally define installation components.

On the "Define Distribution Tree" step, you add and edit the structural elements that make up the distribution tree. You can create your own directory structure and "mount" directories from your file system or add single files into arbitrary directories. With drag and drop and double-clicking on nodes you can modify an existing distribution tree.

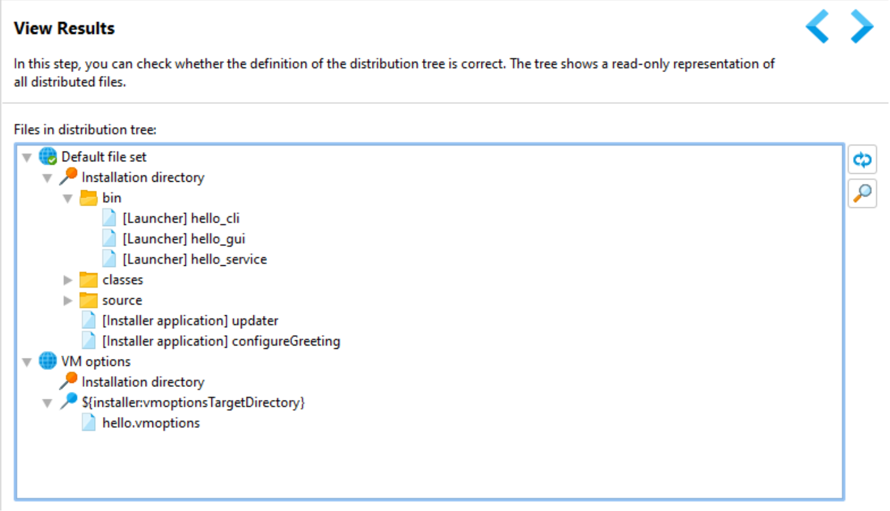


Define Distribution Tree

In this step, you collect all files and directories that you would like to distribute with your media files. Use drag and drop to move entries in the definition tree.

```
Default file set
  Installation directory
    bin
    classes
      Selected content of .\gui (excluding .java)
      Selected content of .\cli (excluding .java)
      Selected content of .\service (excluding '.java)
    source
      File .\cli\HelloCli.java
      File .\gui\HelloGui.java
      File .\service\ServiceDemo.java
VM options [ID 1148]
  Installation directory
  ${installer:vmoptionsTargetDirectory}
    File .\vmoptions\hello.vmoptions (overwrite: never)
```

On the "View Results" step, you then see the actual file tree as it will be collected and distributed by the generated media files [p. 126]. Go to this step to check whether your actions on the "Define Distribution Tree" step actually produce the desired results.



View Results

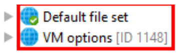
In this step, you can check whether the definition of the distribution tree is correct. The tree shows a read-only representation of all distributed files.

Files in distribution tree:

```
Default file set
  Installation directory
    bin
      [Launcher] hello_cli
      [Launcher] hello_gui
      [Launcher] hello_service
    classes
    source
    [Installer application] updater
    [Installer application] configureGreeting
VM options
  Installation directory
  ${installer:vmoptionsTargetDirectory}
    hello.vmoptions
```

Root container nodes

The top-level nodes in the distribution tree are called **file sets**. There is one "Default file set" node that cannot be deleted or renamed. The relative paths of all files that are added to a file set must be unique. See the help topic on file sets and installation components [p. 20] for more information on how to use file sets.

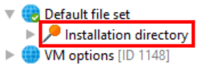


Within a single file set, it causes an error at build time if the installation paths for two files collide. For example, if you have added the contents of two different directories into the same folder in the distribution tree and both directories contain a file `file.txt`, building the project will fail with a corresponding error message. In this case, you have to exclude the file in one of the directory entries. This is only an issue for files, sub-directory hierarchies on the other hand are merged and can overlap between multiple directory entries and explicitly added folders.

You can create new file sets with the *New File Set* action in the add menu on the right side. Each file set has its own "Installation directory" root. If you define custom roots that should be present in multiple file sets, you have to duplicate them.

The child nodes of a file set are called **installation roots**. Their location is resolved when the installer runs. There are two types of roots:

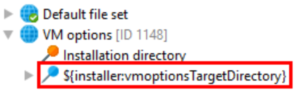
- The **default root** of the distribution tree is labeled "Installation directory" and has a special icon. This is the directory where your application will be installed on the target system. The actual directory location is dependent on user actions at the time of installation. In regular installers, a user can select an arbitrary directory where the application should be installed. For Linux package media files, a user can override the default directory with command line parameters. For archives, the files are simply extracted into a common top-level directory.

  For installers, the installation directory will only be created if you execute an "Install files" action in the installer configuration [p. 148]. By default, the "Install files" action is added to the "Installation" screen. If your installer should not create an installation directory, you can ignore this root and remove the "Install files" action.

  More information on the various installer modes is available in the corresponding help topic [p. 195].

- If your application needs to install files into directories outside the main installation directory, you can add **custom roots** to the distribution tree. This is done with the *New Root* action in the add menu on the right side or in the context menu. The actual location of this root is defined by its name and has to resolve to a valid directory at runtime.

  There are several possibilities for using custom roots. The name of a custom root can be

  - **a fixed absolute path known at compile-time**

    This works for custom environments where there is a fixed policy for certain locations. For example, if you have to install some files to `D:\apps\myapp`, you can enter that path as the name for your custom root.

    If you build installers for different platforms, that root is likely to be different for each platform. In that case, you can use a compiler variable [p. 63] for the name of the custom root and override its value for each media file on the "Customize project defaults->Compiler variables" step of the media wizard.

- **an installer variable that you resolve at runtime**

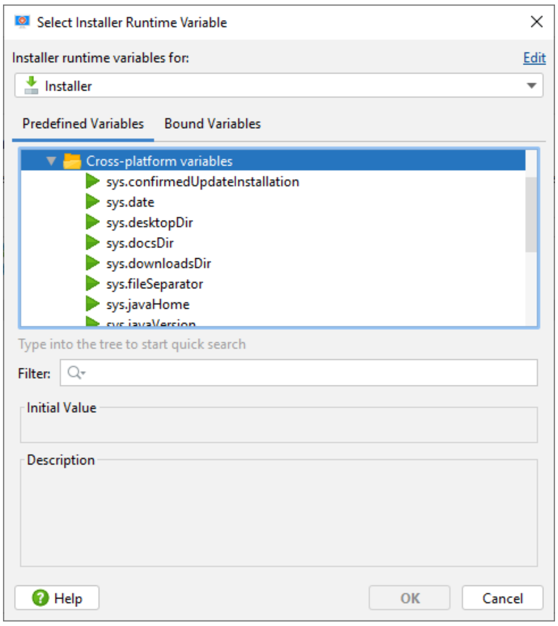
  If you would like to install files into the directory of an already installed application, such as a plugin for your own application, you can use an installer variable that you resolve at runtime. Installer variables have an `installer:` prefix, such as `${installer:rootDir}`, and can be set in a variety of ways [p. 63].

  The most common case would be to add a "Directory selection" screen to the screen sequence [p. 148] and set its variable name property to the variable that you have used as the name of the custom root. For the above example, that would be `rootDir`, without the `${installer:...}` variable syntax.

  Alternatively, you could use a "Set a variable" action to determine the location programmatically.

- **a pre-defined installer variable**

  install4j offers several variables for "magic folders" that point to common directories, such as `${installer:sys.userHome}` which resolves to the user home directory or `${installer:sys.system32Dir}` which resolves to the `system32` directory on Windows. Have a look at the "Cross-platform variables" category in the installer variables selector for a list of variables that are suitable for all platforms.



If a custom installation root is not bound at runtime or if it points to an invalid directory, the contained files will not be installed and there will be no error messages. If you require error handling, you can use a "Run a script" action before the "Install files" action with the appropriate error message and failure strategy.

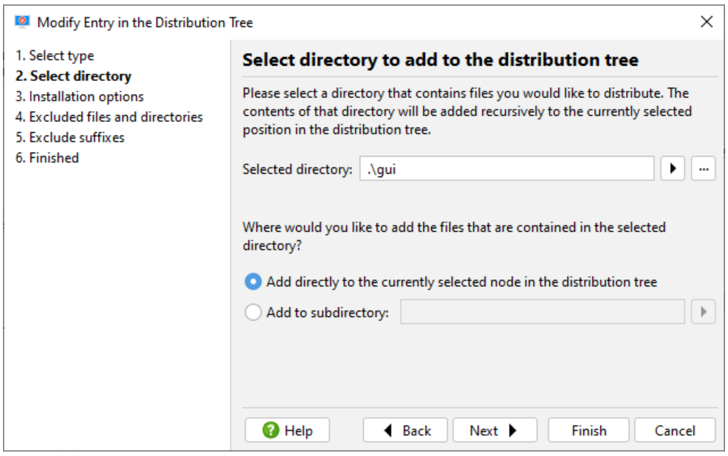
For archive media file types, custom installation roots are not installed. If you require these custom roots for your installation, you cannot use archives.

An alternative way to redirect installed files to different directories is to use the "Directory resolver" property of the "Install files" actions. Also, the "File filter" property of that action can be used to conditionally install files. The use of these properties is only recommended if you require their full flexibility. Otherwise, using custom installation roots and installation components [p. 20] is a better approach.

Content nodes

Adding files to the distribution tree is done with the *Add Files And Directories* action in the add menu on the right side or in the context menu. In the first step of the file wizard you choose the source or the files:

- With a **directory** entry, you recursively add the contents of a selected directory. You have the possibility of excluding certain files and subdirectories and exclude files based on their file suffix. In the configuration wizard you can override the default settings for the overwrite and uninstall policies as well as the Unix file and directory modes.



Select directory to add to the distribution tree

Please select a directory that contains files you would like to distribute. The contents of that directory will be added recursively to the currently selected position in the distribution tree.

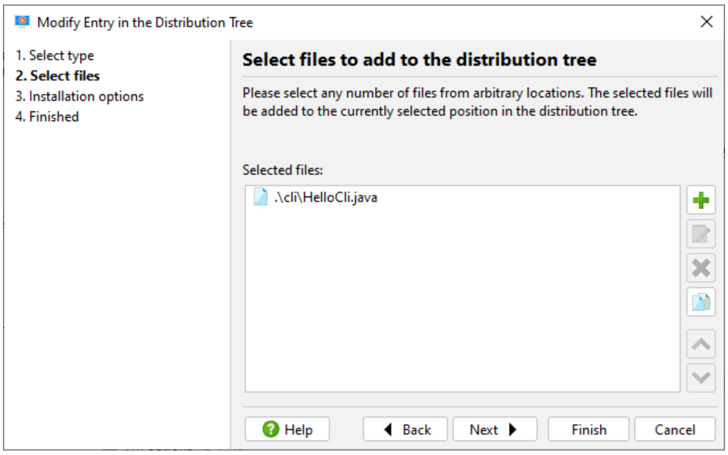
Selected directory: .\gui

Where would you like to add the files that are contained in the selected directory?

● Add directly to the currently selected node in the distribution tree

○ Add to subdirectory:

- Alternatively, you can add **a number of single files**, possibly from different locations, into a single directory. Each selected file will be added as a separate node that has its own settings and can be moved independently in the distribution tree.



Select files to add to the distribution tree

Please select any number of files from arbitrary locations. The selected files will be added to the currently selected position in the distribution tree.

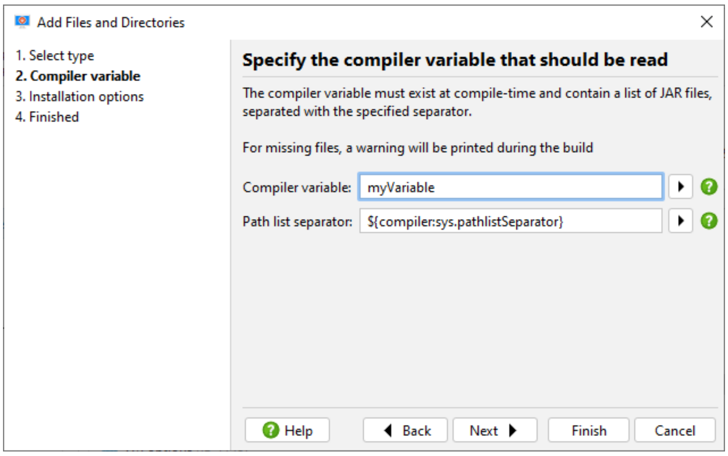
Selected files:

.\cli\HelloCli.java

With the Copy action you can add a file list from the system clipboard. The file list must consist of file entries that are separated by line breaks or the standard path separator (";" on Windows and ":" on Unix). Each file entry can either be absolute or relative. On the first occurrence of a relative path, a directory chooser is shown where you select the root directory against which all further relative paths should be resolved.

- Finally, files can be **passed externally** through a compiler variable. This can be useful if you collect lists of files in your build tool and want to use that information to dynamically build the distribution tree. The command line compiler [p. 220] as well as the Gradle [p. 225], Maven [p. 230] and Ant [p. 239] plugins have mechanisms for setting compiler variables for the build.

  The string that separates different files in the variable value is configurable and set to the platform-specific path separator by default.



Specify the compiler variable that should be read

The compiler variable must exist at compile-time and contain a list of JAR files, separated with the specified separator.

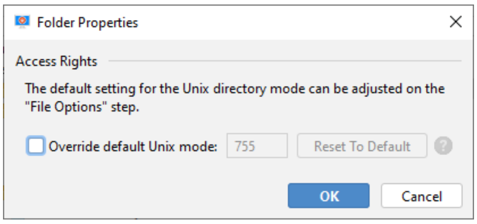
For missing files, a warning will be printed during the build

Compiler variable: myVariable

Path list separator: ${compiler:sys.pathlistSeparator}

Folder nodes

Fixed folder nodes can occur below the root nodes to build nested directory structures. Using the "Edit entry" action on a fixed folder node allows you to edit the unix mode of the folder.



The default setting for the Unix directory mode can be adjusted on the "File Options" step.

☐ Override default Unix mode: 755

Usually, a directory structure will be copied from a staged distribution directory, but fixed folders are useful under several circumstances. For example, if you want to apply different top-level prefix directories, you can add corresponding folder.

Also, fixed folders and single files in fixed folders have a higher precedence than folders and files from directory entries. In this way, you override settings for certain folders or files. For example, if a "contents of a directory" node includes the file `a/b/c.txt`, you can manually add nested folders a and b and then add the single file node `c.txt`. You could then set a different overwrite or uninstall policy for the file. Also, you could override the Unix mode of the directories.

Compiler variables as directory or file names

Using compiler variables [p. 63] as directory or file names in the distribution tree allows you to make compile-time conditional includes. The following rules apply:

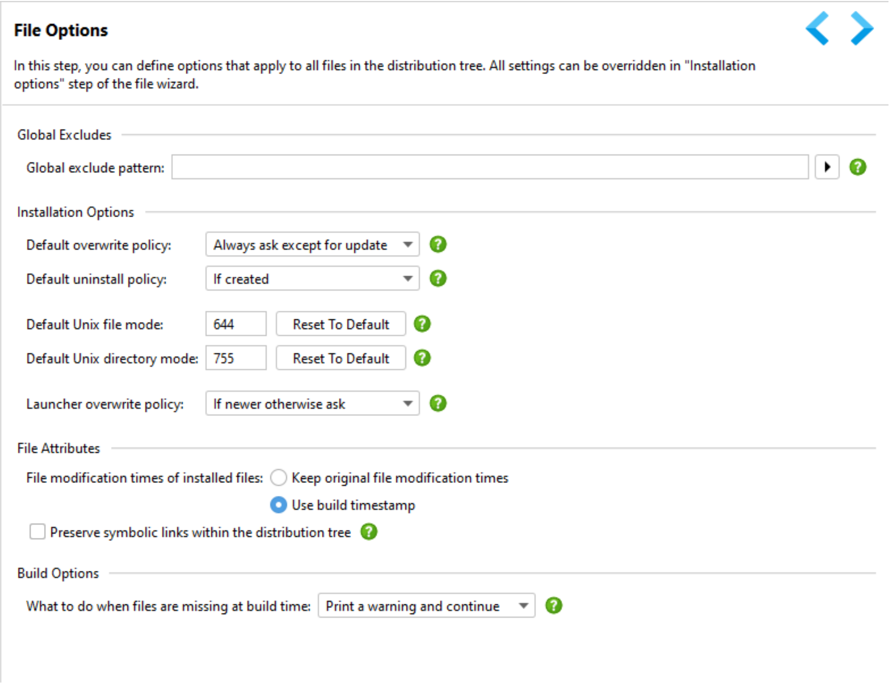
- if a directory node resolves to the empty string after variable replacement, the directory and any contained entries will not be included in the distribution.
- if the source directory of a "contents of directory" node resolves to the empty string after variable replacement, no files will be included by that entry.
- if the file name of a single file node resolves to the empty string after variable replacement, no file will be included.

For conditions that are evaluated at runtime or for adding platform dependent files, you should use files sets [p. 20] instead.

File options

On the "File options" step, a number of settings determine the behavior of the installer and uninstaller. When files are already present, you can choose a number of strategies for the "Install files" actions by changing the "Default overwrite policy". Similarly, the "Uninstall files" action decides what to do for installed files based on the "Default uninstall policy" setting. On Unix, the "Install files" action assigns permissions to installed files and directories as configured in the default Unix file and directory modes on this step. All these options can be overridden in the configuration of the content nodes.

Other available options concern the compilation phase. You can choose the source of the file modification times, specify a global pattern of files and directories that should be ignored when collecting files and select a strategy for what should happen if some specified files are missing at build time.



File Options

In this step, you can define options that apply to all files in the distribution tree. All settings can be overridden in "Installation options" step of the file wizard.

Global Excludes

Global exclude pattern:

Installation Options

Default overwrite policy: Always ask except for update

Default uninstall policy: If created

Default Unix file mode: 644 Reset To Default

Default Unix directory mode: 755 Reset To Default

Launcher overwrite policy: If newer otherwise ask

File Attributes

File modification times of installed files: ○ Keep original file modification times

● Use build timestamp

☐ Preserve symbolic links within the distribution tree

Build Options

What to do when files are missing at build time: Print a warning and continue

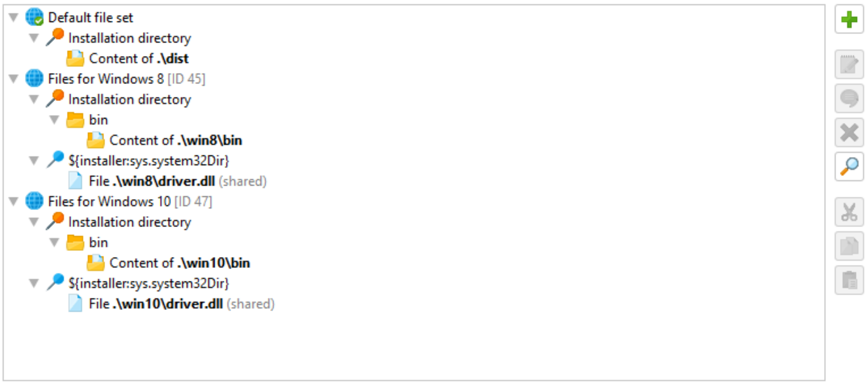
A.4 File Sets And Installation Components

install4j offers two mechanisms to group files: File sets and installation components. File sets are configured in the distribution tree [p. 14] and can be used in a variety of use cases as building blocks for your installers. Installation components are presented to the user at runtime and mark certain parts of the distribution tree that have to be installed if the user chooses an installation component.

Both file sets and installation components are optional concepts that can be ignored if they are not required for an installer project: There is always a "Default file set" to which you can add files in the distribution tree and on the "Installation components" step you do not have to add any components.

File sets

File sets are a way to group files in the distribution tree. When you need to select files in other parts of the install4j IDE, you can select the file set node instead of selecting single files and directories. Each file set has a special "Installation directory" child node that maps to the installation directory selected by the user at run time. Custom installation roots are defined separately for different file sets. If you require the same installation root in two different file sets, you simply define the same root twice.



- Default file set
  - Installation directory
    - Content of **.\dist**
- Files for Windows 8 [ID 45]
  - Installation directory
    - bin
      - Content of **.\win8\bin**
  - ${installer:sys.system32Dir}
    - File **.\win8\driver.dll** (shared)
- Files for Windows 10 [ID 47]
  - Installation directory
    - bin
      - Content of **.\win10\bin**
  - ${installer:sys.system32Dir}
    - File **.\win10\driver.dll** (shared)

The installation of file sets can be toggled programmatically at run time. The code snippet to disable the installation of a file set at run time is

```
context.getFileSetById("123").setSelected(false);
```

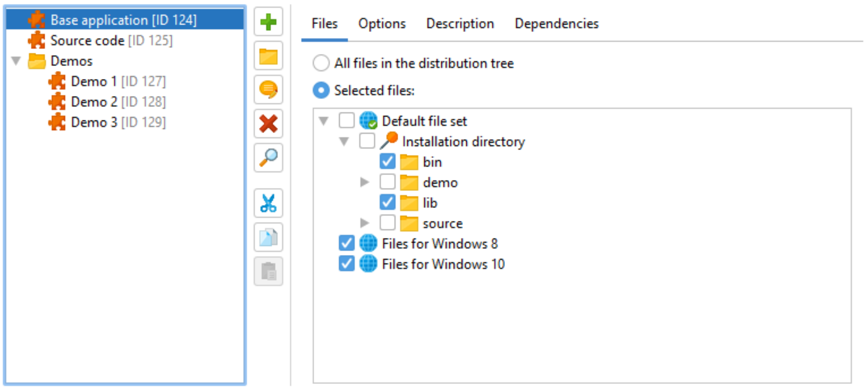
if the ID of the file set is "123". You could insert this snippet into a "Run script" action that is placed before the "Install files" action on the Installer->Screens & Actions step [p. 148]. File set IDs can be displayed by toggling the "Show IDs" tool bar button.

A common use case is to exclude platform-specific files from certain media files. You can define file sets for different platforms and exclude all unneeded file sets in the "Customize project defaults->Exclude files" step in the media wizard. This is an example of how to use file sets at design time in the install4j IDE.

Within one file set, all relative paths must be unique. However, the same relative path can be present in different file sets. Suppose you have different DLL files for Windows 8 and for Windows 10 and higher. You can create two file sets so that the installer contains both alternative versions. Once you find out whether you run on Windows 8 or on Windows 10 and higher, you can disable the file set that should not be installed with the code snippet shown above. By default, all included file sets are installed. If the same relative path occurs twice, it is undefined which version is used. In this case you have to make sure to disable the file sets that are not appropriate.

Installation components

If you define installation components. the installer can ask the user which components should be installed. In the configuration of an installation component you mark the files that are required for this component. A single file or directory can be required by multiple installation components.



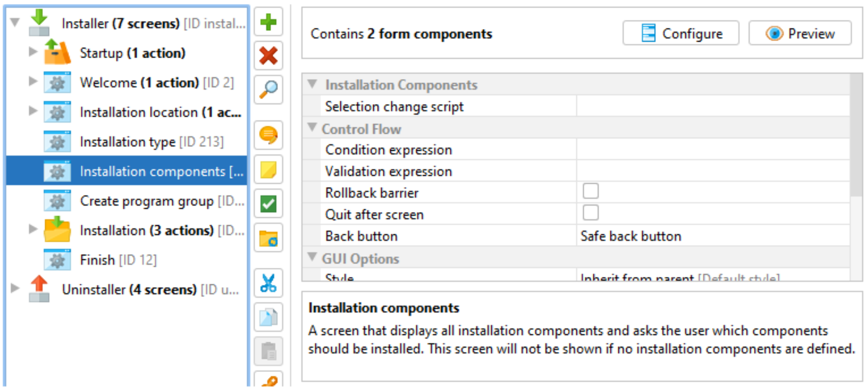
Base application [ID 124]
Source code [ID 125]
Demos
  Demo 1 [ID 127]
  Demo 2 [ID 128]
  Demo 3 [ID 129]

○ All files in the distribution tree
● Selected files:

☐ Default file set
  ☐ Installation directory
    ☑ bin
    ☐ demo
    ☑ lib
    ☐ source
☑ Files for Windows 8
☑ Files for Windows 10

Installation components are defined in a folder hierarchy. This means you can have groups of installation components that are enabled or disabled with a single click. Most options in the configuration of an installation component are used by the "Installation components" screen [p. 163]. They decide how the installation component is presented to the user, whether it should be initially selected or mandatory, and if it has dependencies on other installation components that should be automatically selected. To internationalize the name of the component for different media files, use custom localization keys [p. 63].

The user will only be able to choose installation components if a "Installation components selector" form component is present somewhere in the installer. The "Installation components" screen that is part of the default project template contains that form component ans is only displayed at runtime if you have defined any installation components.



Installer **(7 screens)** [ID instal...
  Startup **(1 action)**
  Welcome **(1 action)** [ID 2]
  Installation location **(1 ac...**
  Installation type [ID 213]
  Installation components [...
  Create program group [ID...
  Installation **(3 actions)** [ID...
  Finish [ID 12]
Uninstaller **(4 screens)** [ID u...

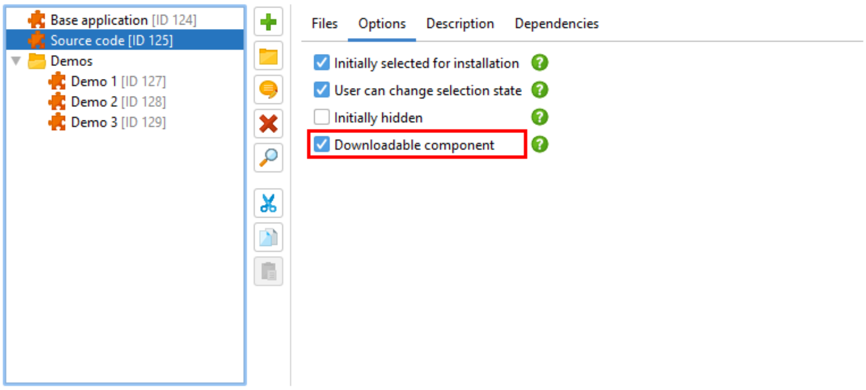
Contains **2** form components

Installation Components	
Selection change script	
Control Flow	
Condition expression	
Validation expression	
Rollback barrier	☐
Quit after screen	☐
Back button	Safe back button
GUI Options	
Style	Inherit from parent [Default style]

Installation components

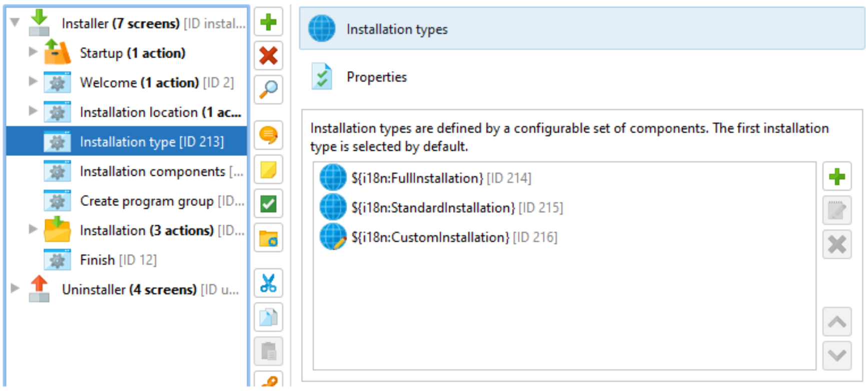
A screen that displays all installation components and asks the user which components should be installed. This screen will not be shown if no installation components are defined.

Another important feature of installation components is that they can be marked as "downloadable". If you configure the download option [p. 134] in the "Data files" step of the media wizard, separate data files will be created for the downloadable components.


Base application [ID 124]
Source code [ID 125]
Demos
Demo 1 [ID 127]
Demo 2 [ID 128]
Demo 3 [ID 129]

☑ Initially selected for installation
☑ User can change selection state
☐ Initially hidden
☑ Downloadable component

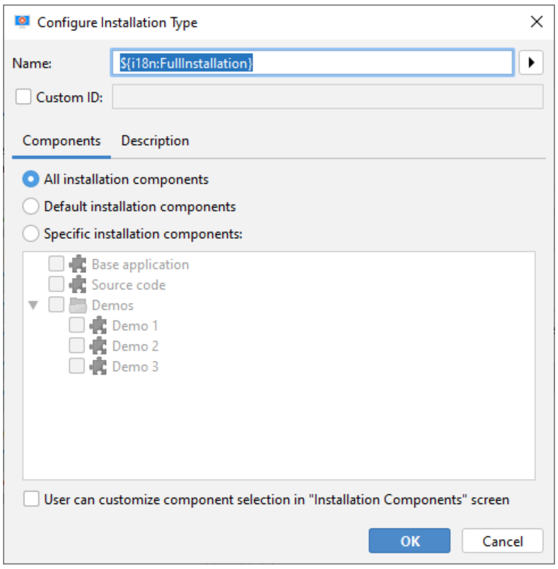
install4j also offers a two-step selection for installation components: In the first step, the user is asked for the desired "installation type". An installation type is a certain selection of installation components. Typical installation type sets are [Full, Minimum, Customize] or [Server, Client, All]. The display and the configuration of installation types is handled by the "Installation type" screen.


Installer **(7 screens)** [ID instal...
Startup **(1 action)**
Welcome **(1 action)** [ID 2]
Installation location **(1 ac...**
Installation type [ID 213]
Installation components [...
Create program group [ID...
Installation **(3 actions)** [ID...
Finish [ID 12]
Uninstaller **(4 screens)** [ID u...

Installation types are defined by a configurable set of components. The first installation type is selected by default.

${i18n:FullInstallation} [ID 214]
${i18n:StandardInstallation} [ID 215]
${i18n:CustomInstallation} [ID 216]

For each configured installation type, you can decide whether the user should be able to further customize the associated installation component selection in the "Installation components" screen or not. If the installation type is not customizable, the installer variable `sys.preventComponentCustomization` is set to `true` and a subsequent "Installation components" screen is not displayed.



Configure Installation Type

Name: ${i18n:FullInstallation}

☐ Custom ID:

● All installation components

○ Default installation components

○ Specific installation components:

☐ Base application
☐ Source code
☐ Demos
  ☐ Demo 1
  ☐ Demo 2
  ☐ Demo 3

☐ User can customize component selection in "Installation Components" screen

The IDs of installation components can be used in expressions, scripts and custom code if you want to check whether the installation component has been selected for installation or not. A typical condition expression for an action would be

```
context.getInstallationComponentById("123").isSelected()
```

if the ID of the component is "123". In this way you can conditionally execute actions depending on whether a component is selected or not.

A.5 Screens And Actions

With screens and actions you configure two separate aspects of the installer: the user interface that is displayed by your installer and uninstaller on the one hand and the actual installation and uninstallation on the other hand. Each screen can have a list of actions attached that are executed when the user advances to the next screen.

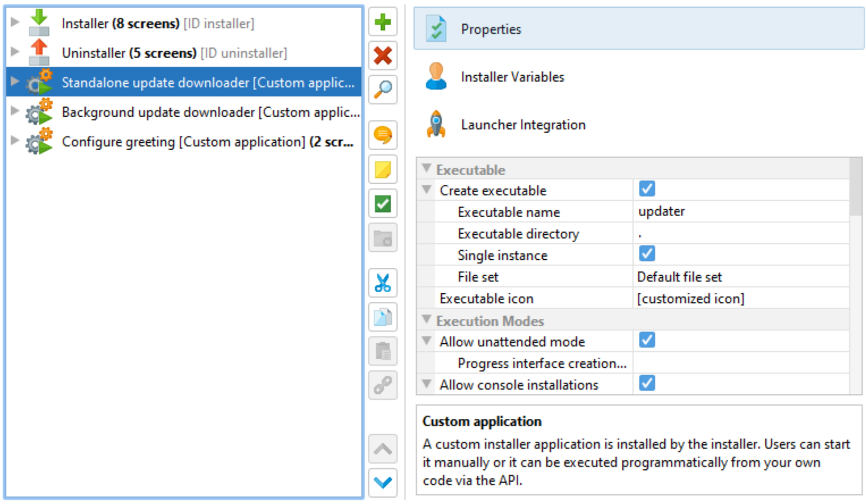
install4j offers a wide variety of pre-defined screens and actions that you can arrange according to your needs. Some of these screens and actions are generic and can be used as programming elements, such as the "Form" [p. 46] screen and the "Run script" action.

While this chapter presents an overview of the concepts of the screen and action system, a later section in the documentation [p. 148] discusses how to configure the related beans in the install4j IDE in detail.

Installer applications

Building an install4j project creates media files which are either installers or archives. An installer is defined by a sequence of screens and actions and is executed when the user executes the media file. Installers usually install an uninstaller which removes the installation. The uninstaller, too, is a freely configurable sequence of screens and actions. Archives do not have an installer or uninstaller and the user extracts the contained data with other tools.

In addition to the installer and uninstaller, you can define custom installer applications [p. 154] that are added to the distribution tree. These custom installer applications can use the same screens and actions that the installer can use. Unlike installer and uninstaller, they are also added to archives. They can be used to write separate maintenance applications for your installations that are either invoked directly by the user or programatically by your application.


Installer (8 screens) [ID installer]
Uninstaller (5 screens) [ID uninstaller]
Standalone update downloader [Custom applic...
Background update downloader [Custom applic...
Configure greeting [Custom application] (2 scr...
Properties
Installer Variables
Launcher Integration

Executable	
Create executable	☑
Executable name	updater
Executable directory	.
Single instance	☑
File set	Default file set
Executable icon	[customized icon]
Execution Modes	
Allow unattended mode	☑
Progress interface creation...	
Allow console installations	☑

Custom application

A custom installer application is installed by the installer. Users can start it manually or it can be executed programmatically from your own code via the API.

The most common use case for custom installer applications is to create auto-updaters. Auto-updaters are described in detail in a separate help topic [p. 114].

Executing first-run tasks for archives

Another important use-case for custom installer applications is to create a first-run installer for archives. While there is no need to install files to the installation directory in the case of an archive, there will usually be screens and actions that set up the environment of your application.

In order to avoid the duplication of screens and actions, install4j offers the possibility to create links to screens and actions. In this way, a custom installer application can include a partial set of the screens and actions in the installer. Such a first-run installer should be added to the `.install4j` runtime directory so that it is not exposed as part of the application. This is done by specifying its "Executable directory" property as the empty string.

Such a first-run installer application is invoked programatically with the `com.install4j.api.launcher.ApplicationLauncher` utility class. To determine whether any of the generated launchers of an installed archive are run for the first time, call

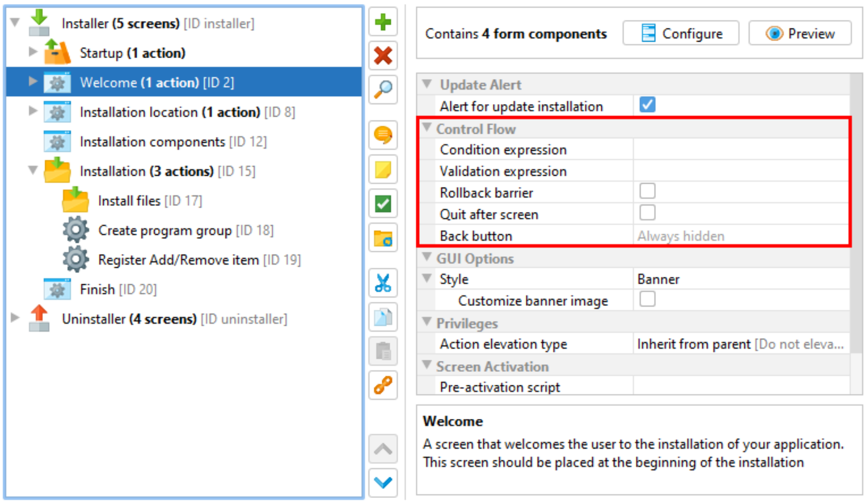
```
ApplicationLauncher.isNewArchiveInstallation()
```

at the beginning of your main method. If it returns `true`, call `launchApplication` or `launchApplicationInProcess` to execute the installer application. Check the Javadoc for detailed information about this API.

Control flow

At runtime, install4j instantiates all screens and actions and organizes the screen flow and action execution. There are a number of aspects regarding this control flow that you can customize in the install4j IDE.

Both screens [p. 163] and actions [p. 169] have an optional "Condition expression" property that can be used to conditionally show screens and execute actions. Screens have a "Validation expression" property that is invoked when the user clicks on the "Next" button allowing you to check whether the user input is valid and whether to advance to the next screen. These are the most commonly used hooks in the control flow for "programming" the installer.


Installer (5 screens) [ID installer]
Startup (1 action)
Welcome (1 action) [ID 2]
Installation location (1 action) [ID 8]
Installation components [ID 12]
Installation (3 actions) [ID 15]
Install files [ID 17]
Create program group [ID 18]
Register Add/Remove item [ID 19]
Finish [ID 20]
Uninstaller (4 screens) [ID uninstaller]
Contains 4 form components
Configure
Preview

Update Alert	
Alert for update installation	☑
Control Flow	
Condition expression	
Validation expression	
Rollback barrier	☐
Quit after screen	☐
Back button	Always hidden
GUI Options	
Style	Banner
Customize banner image	☐
Privileges	
Action elevation type	Inherit from parent [Do not eleva...
Screen Activation	
Pre-activation script	

Welcome

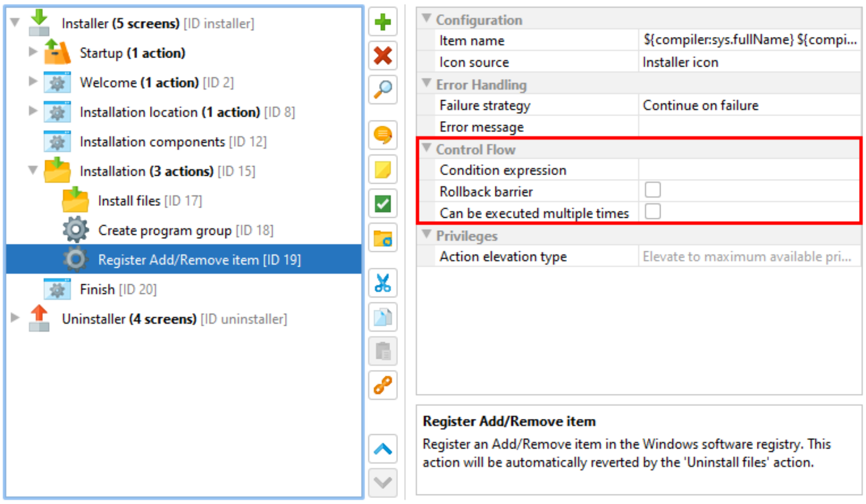
A screen that welcomes the user to the installation of your application. This screen should be placed at the beginning of the installation

All "expression" properties in install4j can be simple Java expressions or scripts of Java code as described in the help topic on scripts [p. 29].

Another hook into the control flow regarding screens is the ability to declare every screen as a "Finish" screen, meaning that the "Next" button will be replaced with a "Finish" button and the installer will quit after that button is pressed. Consider applying the "Banner" style to the screen in that case because it alerts the user that a special screen has been reached.

If you use a series of screens to get user input, users expect to be able to go back to previous screens in order to review or change their input. This is fine as long as no actions are attached to the screen. When actions have been executed, the question arises what should happen if the user goes back to a screen with actions and clicks on "Next" again.

By default, install4j executes actions only once, but that may not be what you want if the actions operate on the user input in a screen. Because install4j has no way of knowing what should happen in this case, it applies a "Safe back button" policy by default: if the previous screen had actions attached, the back button is not visible. You can change this policy for each screen, either making the back button always visible or always hidden. The "Can be executed multiple times" property of each action is relevant in the case where you you make the back button always visible for the next screen.


Installer (5 screens) [ID installer]
Startup (1 action)
Welcome (1 action) [ID 2]
Installation location (1 action) [ID 8]
Installation components [ID 12]
Installation (3 actions) [ID 15]
Install files [ID 17]
Create program group [ID 18]
Register Add/Remove item [ID 19]
Finish [ID 20]
Uninstaller (4 screens) [ID uninstaller]

Configuration
Item name: ${compiler:sys.fullName} ${compi...
Icon source: Installer icon
Error Handling
Failure strategy: Continue on failure
Error message:
Control Flow
Condition expression:
Rollback barrier: ☐
Can be executed multiple times: ☐
Privileges
Action elevation type: Elevate to maximum available pri...

Register Add/Remove item

Register an Add/Remove item in the Windows software registry. This action will be automatically reverted by the 'Uninstall files' action.

Rollback behavior

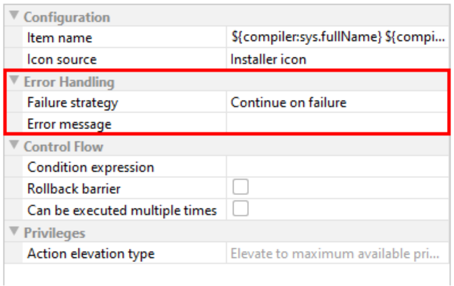
At any time in the installation sequence the user can hit the "Cancel" button. The only exception in the standard screens is the "Display progress" form template screen where the "Cancel" button has been disabled. install4j is able to completely roll back any modification performed by its standard actions.

However, the expectation of a user might not be that the installation is rolled back. Consider a series of post-installation screens that the user doesn't feel like filling out. Depending on the installer, the user may feel that installation will work even if the installer is cancelled at that point. A complete rollback would then not be desirable. For this purpose, install4j offers the concept of a "rollback barrier". Any action or screen can be a rollback barrier which means that any actions before and including that action or screen will not be rolled back if the user cancels later on.

By default, only the "Installation screen" is a rollback barrier. This means that if the user cancels while the actions attached to teh installation screen are running, everything is rolled back. If the user cancels on any of the following screens, nothing that was performed on or before the installation screen is rolled back. With the "Rollback barrier" property of actions and screens you can make this behavior more fine-grained and customize it according to your own needs.

Error handling

Every action has two possible outcomes: failure or success. If an action succeeds the next action is invoked. When the last action of a screen is reached, the next screen is displayed. What should happen if an action doesn't succeed? This depends on how important the action is to your installation. If your application will not be able to run without the successful execution of this action, the installer should fail and initiate a rollback. However, many actions are of peripheral importance, such as the creation of a desktop link. Declaring that the installer has failed because a desktop link could not be created and rolling back the entire installation would be counterproductive. That's why the failure of an action is ignored by install4j by default. If a possible failure of an action is critical, you can configure its "Failure strategy" to either ask the user on whether to continue or to quit immediately.



Configuration

Item name	${compiler:sys.fullName} ${compi...
Icon source	Installer icon

Error Handling

Failure strategy	Continue on failure
Error message	

Control Flow

Condition expression	
Rollback barrier	☐
Can be executed multiple times	☐

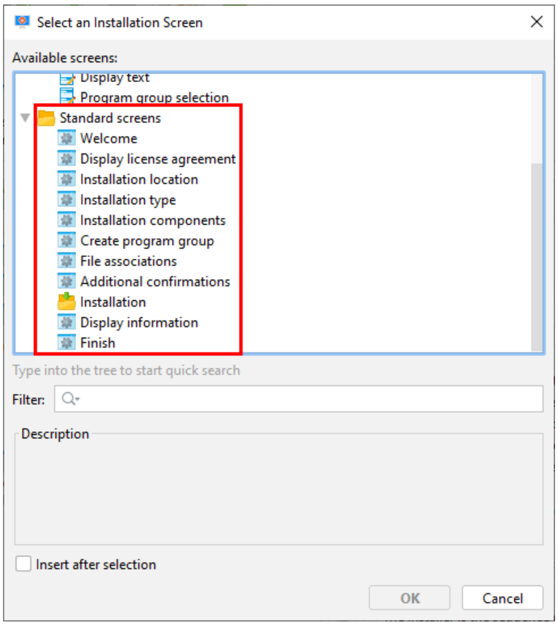
Privileges

Action elevation type	Elevate to maximum available pri...

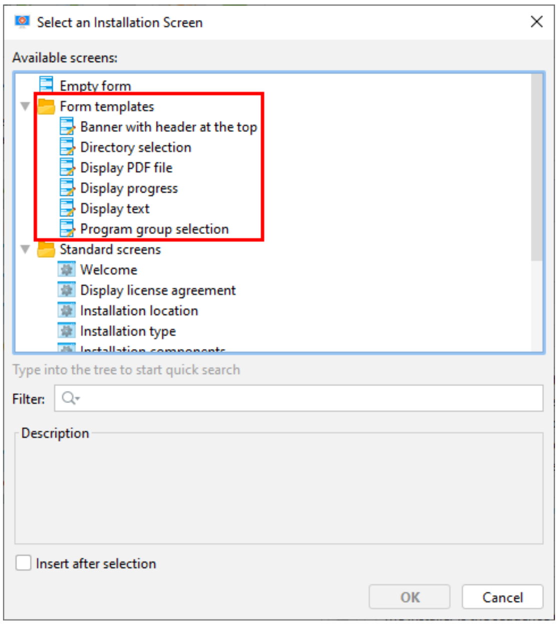
Standard actions in install4j fail silently, for example the "Create a desktop link" action will not display an error message if the link could not be created. For all available failure strategies, you can configure an error message that is displayed in the case of failure. The "Install files" action has its own, more granular failure handling mechanism that is automatically invoked after the installation of each file.

Standard screens and form templates

install4j offers a series of standard screens that are fully localized and serve a specific purpose. These standard screens have a preferred order, when you insert such a screen it will insert itself automatically in the correct position. This order is not mandated, you can re-order the screens in any way you like, however they may not yield the desired result anymore. If for example you place the "Services" screen after the screen with the "Install service" actions (typically the "Installation" screen), the "Services" screen will not be able to modify the service installations anymore and the default values are used.



The form templates don't have a fully defined purpose, their messages are configurable and empty by default. For example the "Display progress" screen is similar to the "Installation" screen, however the title and the subtitle are configurable. For templates also do not have any restriction with respect to how many times they can occur. While the "Installation" screen (and other screens) can occur only once for an installer, the "Display progress" screen could be used multiple times.



Form templates are built with form components and can be a starting point for developing your own screen. Forms allow you to freely define the contents of a screen and are described in a separate help topic [p. 46].

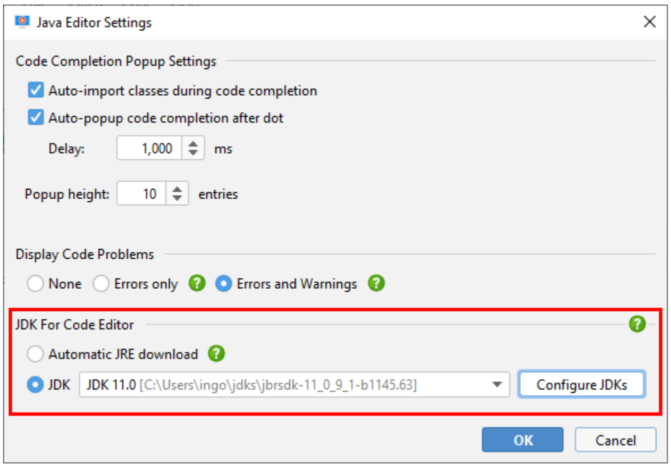
A.6 Scripts

All configurable beans on the Installer->Screens & Actions [p. 148] step have script properties that allow you to customize their behavior, such as executing some code when a button is clicked or a custom initialization of a text field. Also, control flow in the screen and action system is done with scripts and expressions.

Design-time JDK

By default, install4j uses the bundled JRE [p. 89] for compiling scripts up to the Java major version that install4j runs with itself. For JRE bundles with higher Java major versions, install4j uses the current JRE instead.

For special requirements, you can invoke "Settings->Java Editor Settings" in the script editor and select a different JDK for that purpose. The list of available design-time JDKs is saved globally for your entire install4j installation and not for the current project. The only information saved in your project is the name of the JDK configuration. In this way, you can bind a suitable JDK on other installations and on other platforms.



☑ Auto-import classes during code completion

☑ Auto-popup code completion after dot

Delay: 1,000 ms

Popup height: 10 entries

○ None ○ Errors only ● Errors and Warnings

○ Automatic JRE download

● JDK JDK 11.0 [C:\Users\ingo\jdks\jbrsdk-11_0_9_1-b1145.63]

The design-time JDK is used for the following purposes:

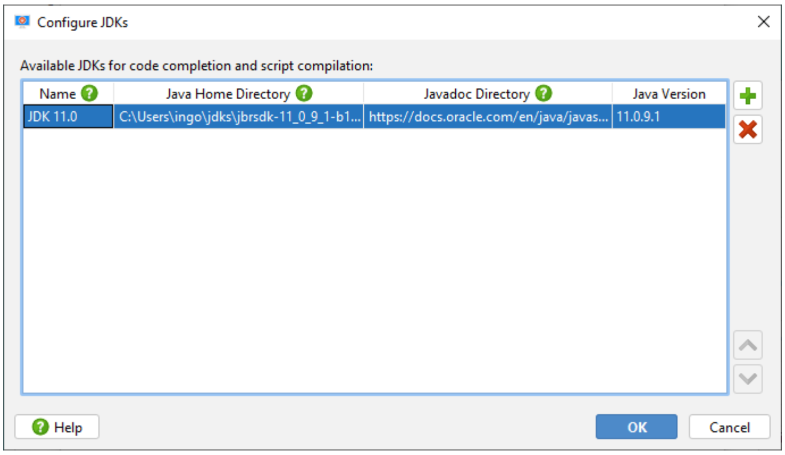
- **Code completion**

  The Java code editor will show completion proposals for classes and methods in the JDK runtime library from the design-time JDK.

- **Context-sensitive Javadoc help**

  If the design-time JDK from the bundled JRE configuration is used, the corresponding Javadoc from the Oracle web site is shown.

  If you manually configure a design-time JDK, you can enter a Javadoc directory to get context-sensitive Javadoc help in the code editor for all classes in the JDK runtime library. By default, context-sensitive Javadoc help is only available for the install4j API.



Available JDKs for code completion and script compilation:

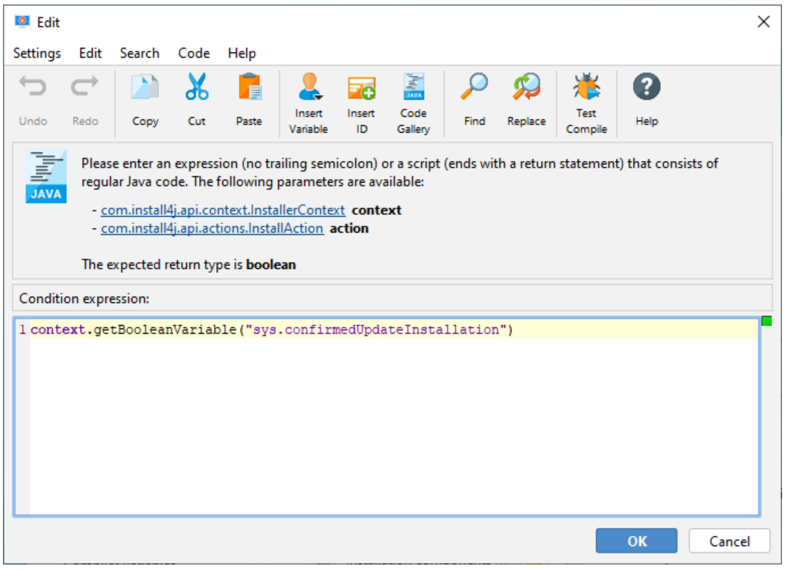
Name	Java Home Directory	Javadoc Directory	Java Version
JDK 11.0	C:\Users\ingo\jdks\jbrsdk-11_0_9_1-b1...	https://docs.oracle.com/en/java/javas...	11.0.9.1

- **Code compilation**

  install4j uses a bundled eclipse compiler, so it does not use the compiler from the design-time JDK. However, it needs a runtime library against which scripts entered in the installer configuration [p. 24] are compiled. The version of that JDK should correspond to the minimum Java version for the project. This is automatically the case if the design-time JDK from the bundled JRE configuration is used. For a manually selected design-time JRE, if its minimum Java version is higher than the minimum Java version of the project, runtime errors can occur if you accidentally use newer classes and method.

The code editor

The Java code editor is shown for script properties on the Installer->Screens & Actions [p. 148] step for any configurable bean including screens, actions, form components and groups, or when you edit the code for static fields and methods on the Installer->Screens & Actions->Custom Code [p. 152] step.



Please enter an expression (no trailing semicolon) or a script (ends with a return statement) that consists of regular Java code. The following parameters are available:

- com.install4j.api.context.InstallerContext **context**
- com.install4j.api.actions.InstallAction **action**

The expected return type is **boolean**

Condition expression:

```
context.getBooleanVariable("sys.confirmedUpdateInstallation")
```

The box above the text editor shows the available parameters as well as the required return type. If parameters or return type are classes - and not primitive types - they will be shown as hyperlinks. Clicking on such a hyperlink opens the Javadoc in the external browser.

To get more information on classes from the `com.install4j.*` packages, choose *Help->Show API Documentation* from the menu and read the help topic for the install4j API [p. 212].

A number of packages can be used without using fully-qualified class names. Those packages are:

- java.util.*
- java.io.*
- javax.swing.*
- com.install4j.api.*
- com.install4j.api.beans.*
- com.install4j.api.context.*
- com.install4j.api.events.*
- com.install4j.api.screens.*
- com.install4j.api.actions.*
- com.install4j.api.formcomponents.*
- com.install4j.api.update.*
- com.install4j.api.windows.*
- com.install4j.api.unix.*

You can put a number of import statements as the first lines in the text area in order to avoid using fully qualified class names. For example:

```
import java.awt.Color;
import java.awt.EventQueue;

EventQueue.invokeLater(() -> {
    JTextField textField =
(JTextField)formEnvironment.getFormComponentById("123").getConfigurationObject();
    textField.setBackground(Color.RED);
});
```

If the gutter icon in the top right corner of the dialog is green, your script is going to compile unless you have disabled error analysis in the Java editor settings that are accessible in the menu of the script editor dialog.

In some situations, you may want to try the actual compilation. Choosing *Code->Test Compile* from the menu will compile the script and display any errors in a separate dialog. Saving your script with the *OK* button will not test the syntactic correctness of the script. When your install4j project is compiled, the script will also be compiled and errors will be reported.

Expressions or scripts

Java code properties can either be expressions or scripts. install4j automatically detects whether you have entered an expression or a script.

An expression does not have a trailing semicolon and evaluates to the required return type. For example:

```
!context.isUnattended() && !context.isConsole()
```

The above example would work as the condition expression of an action and skip the action for unattended or console installations.

A script consists of a series of Java statements with a return statement of the required return type as the last statement. For example:

```
if (!context.getBooleanVariable("enterDetails")) {
  context.goForward(2, true, true);
}
return true;
```

The above example would work as the validation expression of a screen. If the variable with name "enterDetails" is not set to `true`, it would skip two screens forward, checking the conditions of the target screen as well as executing the actions of the current screen.

Script parameters

The primary interface to interact with the installer or uninstaller is the **context** which is nearly always among the available parameters. The context provides information about the current installation and gives access to variables, screens, actions and other elements of the installation or uninstallation. The parameter is of type

- `com.install4j.api.context.InstallerContext` for screens and actions in the installation mode
- `com.install4j.api.context.UninstallerContext` for screens and actions in the uninstallation mode
- `com.install4j.api.context.Context` for form components.

Apart from the context, the available parameters include the action, screen or form component to which the Java code property belongs. If you know the implementation class, you can cast to it and modify the object as needed.

Many other useful static methods are also contained in the class `com.install4j.api.Util`, for example OS detection methods or methods to display messages in a way that works for all installer modes:

```
if (Util.isMacOS()) {
    Util.showWarningMessage("This warning is only shown on macOS");
}
```

Editor features

The Java editor offers the following code assistance powered by the eclipse platform:

- **Code completion**

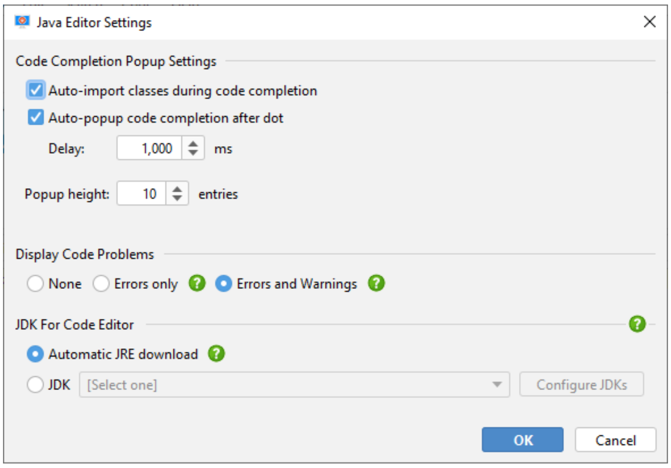
  Pressing `CTRL-Space` brings up a popup with code completion proposals. Also, typing a dot (".") shows this popup after a delay if no other character is typed.

  While the popup is displayed, you can continue to type or delete characters with `Backspace` and the popup will be updated accordingly. "Camel-hump completion" is supported, i.e. typing `NPE` and hitting `CTRL-Space` will propose `NullPointerException` among other classes. If you accept a class that is not automatically imported, the fully qualified name will be inserted.

  The completion popup can suggest:

- v variables and default parameters. Default parameters are displayed in bold font.
- p packages (when typing an import statement)
- c classes
- f fields (when the context is a class)
- m methods (when the context is a class or the parameter list of a method)

You can configure code completion behavior in the Java editor settings.



Java Editor Settings

Code Completion Popup Settings

☑ Auto-import classes during code completion

☑ Auto-popup code completion after dot

Delay: 1,000 ms

Popup height: 10 entries

Display Code Problems

○ None ○ Errors only ● Errors and Warnings

JDK For Code Editor

● Automatic JRE download

○ JDK [Select one]

- **Problem analysis**

  The code that you enter is analyzed on the fly and checked for errors and warning conditions. Errors are shown with red underlines in the editor and with red stripes in the right gutter. Warnings, such as unused variable declarations, are shown with a yellow background in the editor and with yellow stripes in the right gutter. Hovering the mouse over an error or warning in the editor as well as hovering the mouse over a stripe in the gutter area displays the error or warning message.

  The status indicator at the top of the right gutter is green if there are no warnings or errors, yellow if there are warnings but no errors and red if there are errors. In the latter case the code will not compile and the installer cannot be generated.

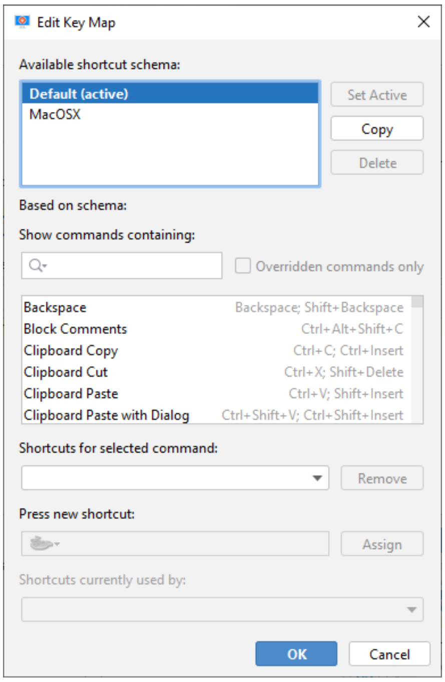
  You can configure the threshold for problem analysis in the Java editor settings.

- **Context-sensitive Javadoc**

  Pressing `SHIFT-F1` opens the browser with the Javadoc page that describes the element at the cursor position. Javadoc for the Java runtime library can only be displayed if a design-time JDK is configured and a valid Javadoc location is specified in the design-time JDK configuration.

Key bindings

All key bindings in the Java code editor are configurable. The key map editor is displayed by choosing *Settings->Key map* from the menu in the Java code editor dialog. On macOS, that menu is shown as a "hamburger" menu on the right side of the tool bar.


Edit Key Map
Available shortcut schema:
Default (active)
MacOSX
Set Active
Copy
Delete
Based on schema:
Show commands containing:
☐ Overridden commands only
Shortcuts for selected command:
Remove
Press new shortcut:
Assign
Shortcuts currently used by:
OK
Cancel

Backspace	Backspace; Shift+Backspace
Block Comments	Ctrl+Alt+Shift+C
Clipboard Copy	Ctrl+C; Ctrl+Insert
Clipboard Cut	Ctrl+X; Shift+Delete
Clipboard Paste	Ctrl+V; Shift+Insert
Clipboard Paste with Dialog	Ctrl+Shift+V; Ctrl+Shift+Insert

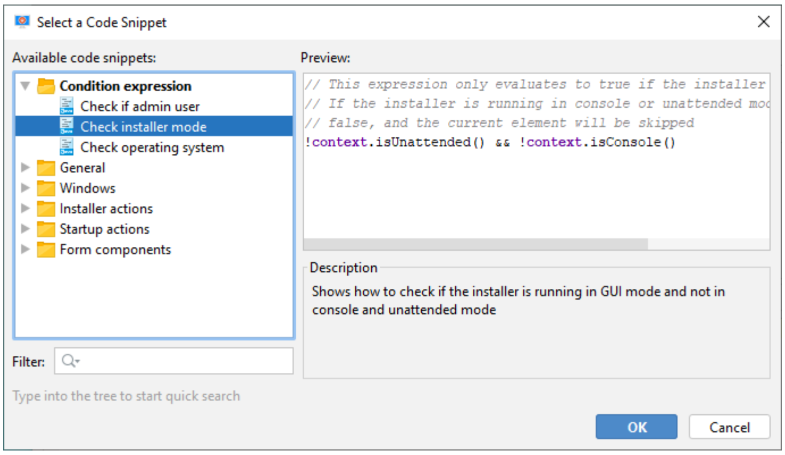
The active key map controls all key bindings in the editor. By default, the **[Default]** key map is active. The default key map cannot be edited directly, to customize key bindings, you first have to copy it. Except for the default key map, the name of a key map can be edited by double-clicking on it.

When assigning new keystrokes or removing existing key strokes from a copied map, the changes to the base key map will be shown as "overridden" in the list of bindings. The key map editor also features search functionality for locating bindings as well a conflict resolution mechanism.

Key bindings are saved in the file `$CONFIG_DIR/install4j/v9/editor_keymap.xml` where `$CONFIG_DIR` is `%USERPROFILE%\AppData\Local` on Windows, `$HOME/.config` on Linux and `$HOME/Library/Application Support` on macOS. This file only exists if the default key map has been copied. When migrating an install4j installation to a different computer, you can copy this file.

Code gallery

The Java code editor offers a code gallery containing useful snippets that show you how to get started with using the install4j API. The code gallery is displayed with the "Code gallery" tool bar button in the script editor.



```
// This expression only evaluates to true if the installer
// If the installer is running in console or unattended mo
// false, and the current element will be skipped
!context.isUnattended() && !context.isConsole()
```

Shows how to check if the installer is running in GUI mode and not in console and unattended mode

You can either copy a portion of the script with `CTRL-C` or click *OK* to insert the entire script at the current cursor position.

Not all code snippets are directly usable in the script that you are editing. Also, some script properties have special code snippets that are only shown for this property. If such code snippets exist, they are displayed in bold in a separate category with the name of the script property.

Installer variables and scripts

Screens, actions and form components are wired together with installer variables that can be set and retrieved with little code snippets that make use of the `context` parameter that is available for most scripts. Any object can be used as the value for a variable, for a condition you can use boolean values. In a "Run script" action, you could set a boolean variable like this:

```
boolean myCondition = ...
context.setVariable("myCondition", myCondition);
```

Instead of calling `setVariable` in a "Run script" action, you can also use a "Set a variable" action where the return value of the script is saved to an installer variable.

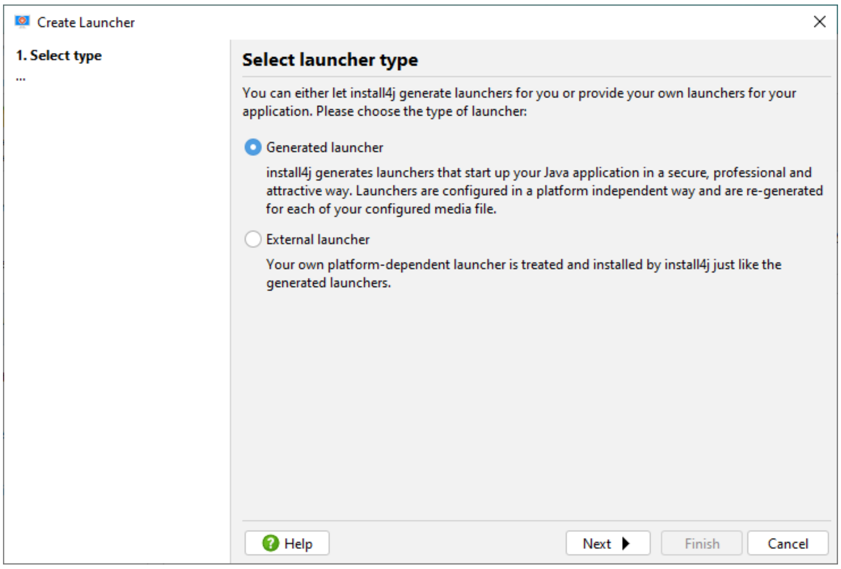
Getting installer variables is done with the `context.getVariable(String variableName)` method. The convenience method `context.getBooleanVariable(String variableName)` makes it easier to check conditions and write them as expressions without a return value:

```
context.getBooleanVariable("myCondition")
```

To use installer variables with a string value in text properties of actions, screens and form components, write them as `${installer:myVariableName}` or use the variable selector button that inserts them with the correct syntax.

A.7 Generated Launchers

Launchers are responsible for starting your application. There are two types of launchers:



Select launcher type

You can either let install4j generate launchers for you or provide your own launchers for your application. Please choose the type of launcher:

● Generated launcher

install4j generates launchers that start up your Java application in a secure, professional and attractive way. Launchers are configured in a platform independent way and are re-generated for each of your configured media file.

○ External launcher

Your own platform-dependent launcher is treated and installed by install4j just like the generated launchers.

- **Generated launchers**

  install4j can generate native launchers that start your application. For example, on Windows, an `.exe` file will be created that among other things takes care of finding a suitable JRE, displaying appropriate error messages if required and then starts your application. Using launchers generated by install4j has numerous advantages as compared to using home-grown batch files and shell scripts.

  Each launcher definition is compiled separately for each defined media file [p. 126]. This means that for the majority of all cases, a single launcher definition will be sufficient to start your application. If, for example, your distribution contains two GUI applications and a command line application, you have to define 3 launchers, regardless of how many media files you define.

  When your application is started with a launcher generated by install4j, you can query the system property `install4j.appDir` to get the installation directory and and `install4j.exeDir` to get the directory where the launcher resides. Use calls like

  ```
  System.getProperty("install4j.appDir")
  ```

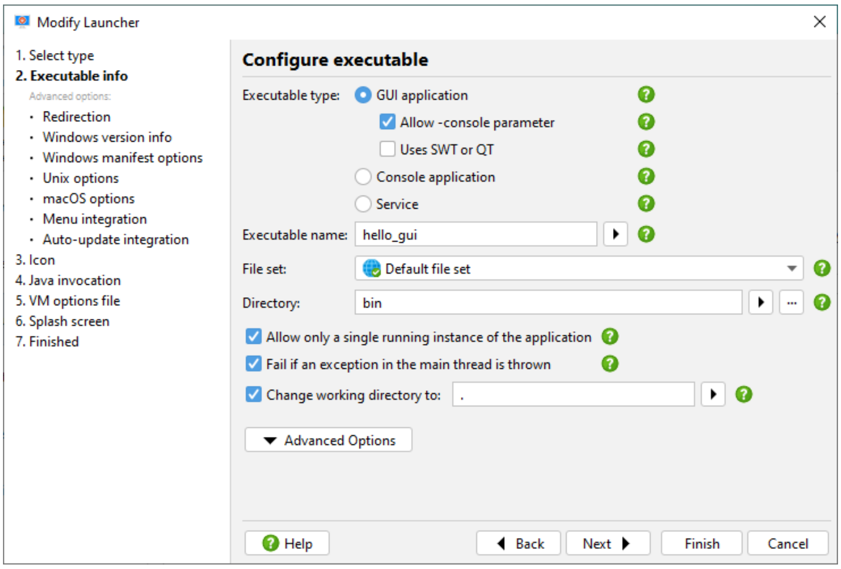
  to access these values.

- **External launchers**

  If you already have an external launcher for your application, you can let install4j use that launcher instead of generating one. Because external launchers are most likely platform-dependent, you will have to add external launchers for each platform that is targeted by your media files. Make sure to exclude the irrelevant launchers in your media file definitions in this case.

Types of generated launchers

Executables created by install4j can be either GUI applications, console applications or service applications.



Configure executable

Executable type: ● GUI application

☑ Allow -console parameter

☐ Uses SWT or QT

○ Console application

○ Service

Executable name: hello_gui

File set: Default file set

Directory: bin

☑ Allow only a single running instance of the application

☑ Fail if an exception in the main thread is thrown

☑ Change working directory to: .

There is no terminal window associated with a **GUI application**. If stdout and stderr are not redirected on the "Executable info->Redirection" step of the launcher wizard, both streams are inaccessible for the user. This corresponds to the behavior of `javaw(.exe)`.

On Windows, if you launch the executable from a console window, a GUI application can neither write to or read from that console window. Sometimes it might be useful to use the console, for example for seeing debug output or for simulating a console mode with the same executable. In that case you can select the `Allow -console parameter` check box. If the user supplies the `-console` parameter when starting the launcher from a console window, the launcher will try to acquire the console and redirect stdout and stderr to it. If you redirect stderr and stdout in the "Executable->Redirection" step, that output will not be written to the console.

A **console application** has an associated terminal window. If a console application is opened from the Windows explorer, a new terminal window is opened. If stdout and stderr are not redirected on the "Executable info->Redirection" step of the launcher wizard, both streams are printed on the terminal window. This corresponds to the behavior of `java(.exe)`.

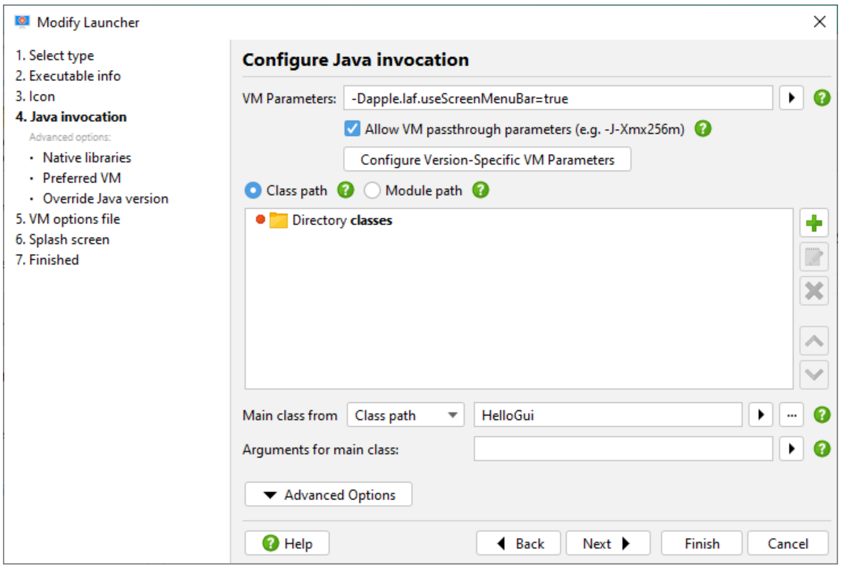
Finally, a **service** runs independently of logged-on users and can be run even if no user is logged on at all. A service cannot rely on the presence of a console, nor can it open windows. On Microsoft Windows, a service executable will be compiled by install4, on macOS a launch daemon will be created and on Unix-like platforms a start/stop script will be generated.

When a service is started, the `main` method of the configured main class will be called. To handle the shutdown of your service, you can use the `Runtime.addShutdownHook()` method to register a thread that will be executed before the JVM is terminated.

For information on how services are installed or uninstalled, see the help topic on services [p. 97].

Java invocation

The most important configuration of a launcher is done on the "Java invocation" step of the launcher wizard and revolves around replicating the arguments you would pass to the Java launcher in a batch file:



VM Parameters: -Dapple.laf.useScreenMenuBar=true

☑ Allow VM passthrough parameters (e.g. -J-Xmx256m)

● Class path ○ Module path

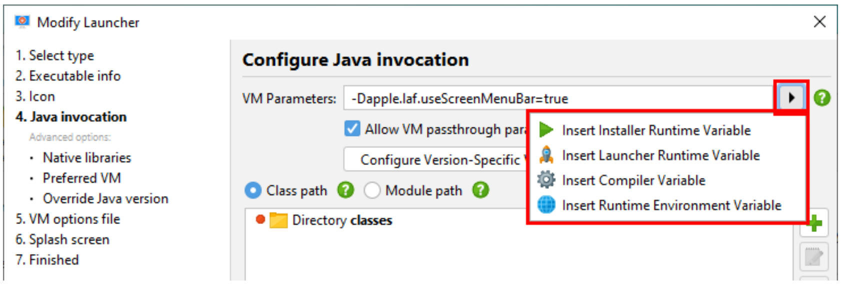
Directory **classes**

Main class from: Class path — HelloGui

Arguments for main class:

- ### VM parameters

  You can provide a fixed list of VM parameters to your launcher and also add version-specific VM parameters. Fixed VM parameters can contain compiler, launcher and installer variables [p. 63].

  VM Parameters: -Dapple.laf.useScreenMenuBar=true

  ☑ Allow VM passthrough par

  ● Class path ○ Module path

  Directory **classes**

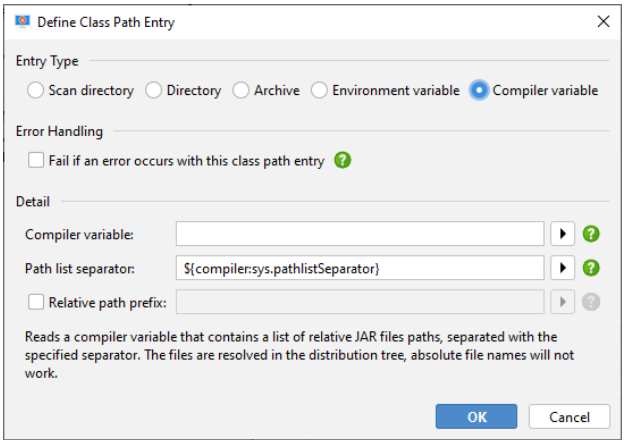
  Compiler variables are replaced at build time, launcher variables are replaced by the launcher so that the VM sees the replaced value from the very beginning, and installer variables are replaced in the main method. This means that using installer variables is not suitable for setting certain kinds of VM parameters like `-Xmx`, but can be useful for replacing system properties that are only used by your code or by libraries.

  See the separate help topic on VM parameters [p. 84] for more information on the various ways to set VM parameters for launchers.

- ### Module or class path

  On the "Java invocation" step of the launcher wizard you can configure both the module path and the class path. These settings correspond to the `--module-path` and the `-cp` parameters of the standard Java launcher. The module path is only applicable for Java 9 and higher. Like for the standard Java launcher, you can add directories, single archives or directories with archives. In addition, you can add archives from environment variables and from compiler variables.

  The compiler variable entry is useful if the set of JAR files that should be added to the module path or class path is calculated in your build system and these JAR files are not staged to a fixed set of directories that you could reference in install4j. In that case, the the command line compiler [p. 220] as well as the plugins for Gradle [p. 225], Maven [p. 230] and Ant [p. 239] can set a compiler variable externally where the single JAR files are separated by a configurable separator.



Entry Type

○ Scan directory ○ Directory ○ Archive ○ Environment variable ● Compiler variable

Error Handling

☐ Fail if an error occurs with this class path entry

Detail

Compiler variable:

Path list separator: ${compiler:sys.pathlistSeparator}

☐ Relative path prefix:

Reads a compiler variable that contains a list of relative JAR files paths, separated with the specified separator. The files are resolved in the distribution tree, absolute file names will not work.

- ## Main class

  For Java 9 and higher, you can choose a main class from either the module or the class path. If you choose the module path option, the syntax for the main class is `<module name>/<class name>` and corresponds to the `--module` parameter of the standard Java launcher. The chooser dialog shows all the available main classes and inserts the correct value automatically.

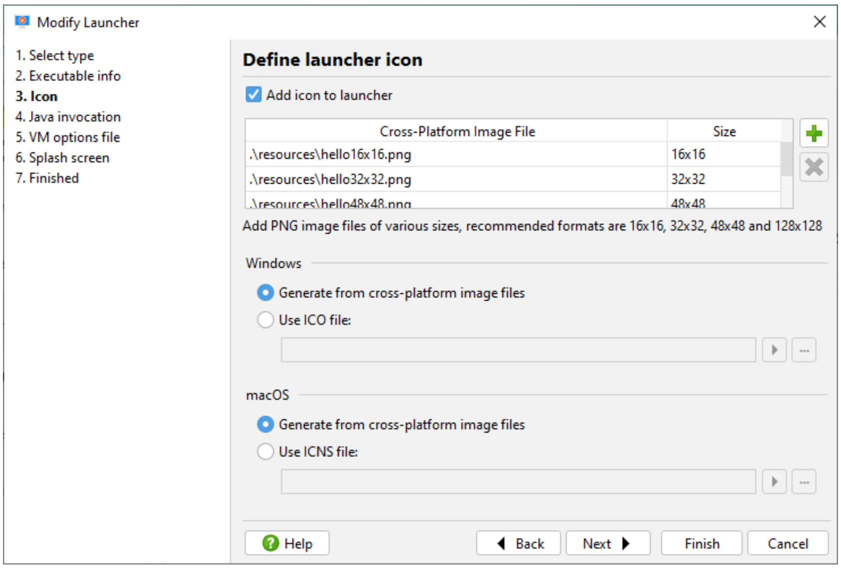
- ## Arguments

  Like VM parameters, the list of fixed arguments supports compiler, launcher and installer variables. Arguments on the command line are appended to the fixed list of arguments.

Cross-platform launcher features

Generated launchers optionally support a **single instance mode** on all supported platforms. You can use the launcher API [p. 216]to register a startup handler that receives the command line parameters if the launcher is started more than once. In this way, you can handle file associations with a single application instance. GUI launchers on macOS are always in single instance mode because that is a fundamental property of application bundles.

Icons for launchers can be generated from a set of PNG files. On Windows, an `.ico` file and on macOS an `.icon` file is compiled, on Linux the generated `.desktop` file references the PNG images. You can also provide pre-built ICO and ICNS files.



Define launcher icon

☑ Add icon to launcher

Cross-Platform Image File	Size
.\resources\hello16x16.png	16x16
.\resources\hello32x32.png	32x32
.\resources\hello48x48.png	48x48

Add PNG image files of various sizes, recommended formats are 16x16, 32x32, 48x48 and 128x128

Windows

● Generate from cross-platform image files

○ Use ICO file:

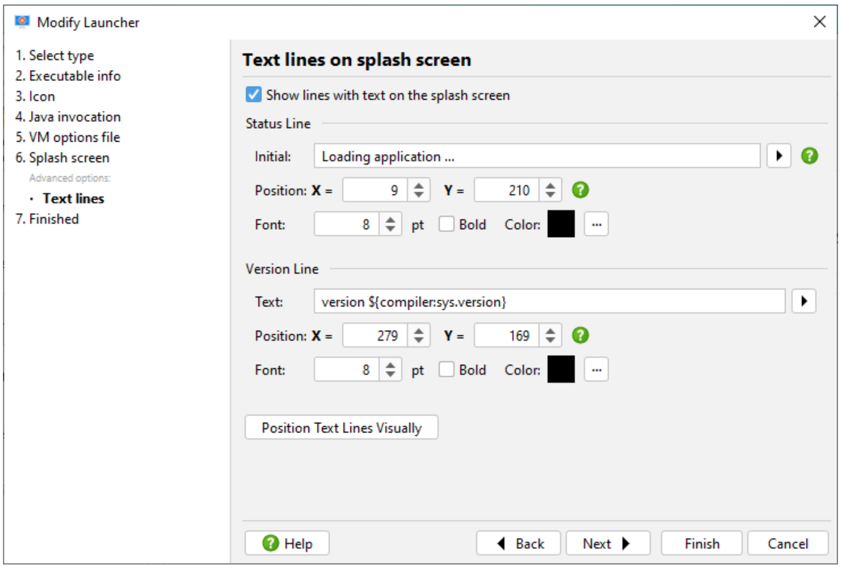
macOS

● Generate from cross-platform image files

○ Use ICNS file:

A **splash screen** image can be configured on the "Splash screen" step of the launcher wizard. The `-splash` command line parameter does not work for the generated executables, because it is part of the standard Java launchers and not of the Java runtime itself. An exception is the argument `-J-splash:none` which is emulated by install4j Windows launchers to disable the splash screen from the command. The splash screen supports additional high DPI images with a @2x suffix in the file name.

In addition to the standard splash screen image, install4j allows you to position two lines of text on top of the splash screen image, a version line and a status line. The status line can be updated from your launcher with the launcher API [p. 216].



Text lines on splash screen

☑ Show lines with text on the splash screen

Status Line

Initial: Loading application ...

Position: X = 9 Y = 210

Font: 8 pt ☐ Bold Color: ■

Version Line

Text: version ${compiler:sys.version}

Position: X = 279 Y = 169

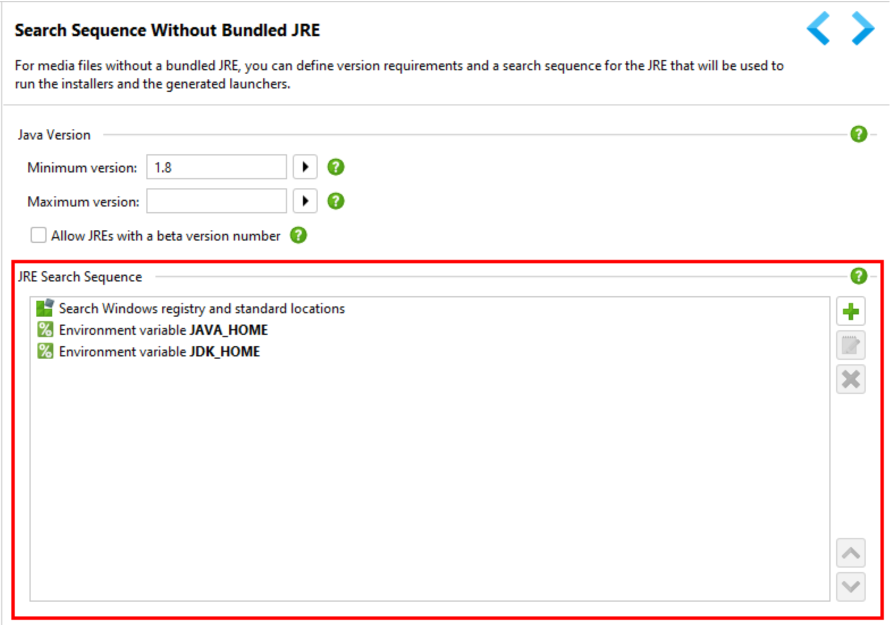
Font: 8 pt ☐ Bold Color: ■

If your code loads **native libraries** via `System.load(...)` or if a native library loads dependent libraries, the native library path has to be modified to include the directories where these native libraries are located. In batch or shell scripts you would do this in a platform-specific way, modifying `PATH` on Windows, `DYLD_LIBRARY_PATH` on macOS, `LD_LIBRARY_PATH` on Linux and a variety of other variable names on different Unix variants.

In install4j, you can use the "Java invocation->Native libraries" step of the launcher wizard to specify such directories, and the launcher will take care that the appropriate environment variable is modified. These directories end up in the `java.library.path` system property in your launcher. If you need different directories for different media files, use a compiler variable for the directory name and override it for each media file.

JRE search sequence

By default, launchers use the bundled JRE [p. 89]. In case you do not bundle a JRE, the JRE search sequence determines how install4j searches for a JRE on the target system. New configurations get a pre-defined default search sequence.



Search Sequence Without Bundled JRE

For media files without a bundled JRE, you can define version requirements and a search sequence for the JRE that will be used to run the installers and the generated launchers.

Java Version

Minimum version: 1.8

Maximum version:

☐ Allow JREs with a beta version number

JRE Search Sequence

- Search Windows registry and standard locations
- Environment variable **JAVA_HOME**
- Environment variable **JDK_HOME**

Apart from searching the Windows registry, well-known standard installation locations and paths in environment variables, you can also configure a relative directory in your distribution tree. This is useful if you distribute your own JRE for a launcher that is not provided through a JRE bundle managed by install4j.

install4j has a special mechanism which allows you to bundle JREs with your media files. If you choose a particular JRE for bundling [p. 89] in one of the media file wizards [p. 126], this JRE will always be used first and you do not need to adjust the search sequence yourself.

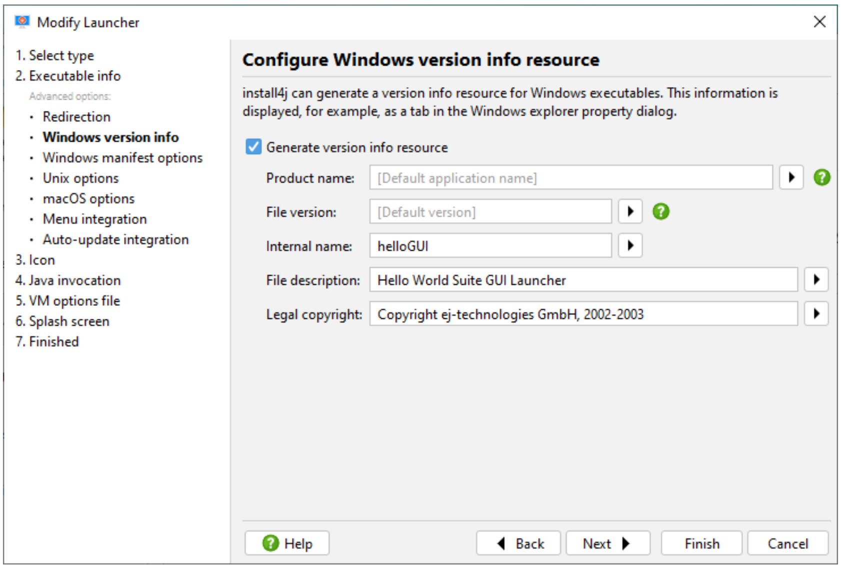
If you do not bundle a JRE and a launcher has special Java version requirements that differ from those of the other launchers, you can override them on the "Java invocation->Override Java version" step of the launcher wizard.

If you have problems with JRE detection at runtime, see the help topic on error handling [p. 210] for a description on how to get diagnostic information.

Windows-specific features

A **version info resource** will enable the Windows operating system to determine meta information about your executable. This information is displayed in various locations. For example, when opening the property dialog for the executable in the Windows explorer, a "Version" tab will be present in the property dialog if you have chosen to generate the version info resource.

The version info resource consists of several pieces of information. If you check `Generate version info resource` on the "Executable->Windows version info" step of the launcher wizard, there are several fields whose values must be entered. The "original file name", the "company name", the "product name" and the "product version" fields in the version info resource are filled in automatically by install4j and cannot be configured.



Configure Windows version info resource

install4j can generate a version info resource for Windows executables. This information is displayed, for example, as a tab in the Windows explorer property dialog.

☑ Generate version info resource

Product name: [Default application name]

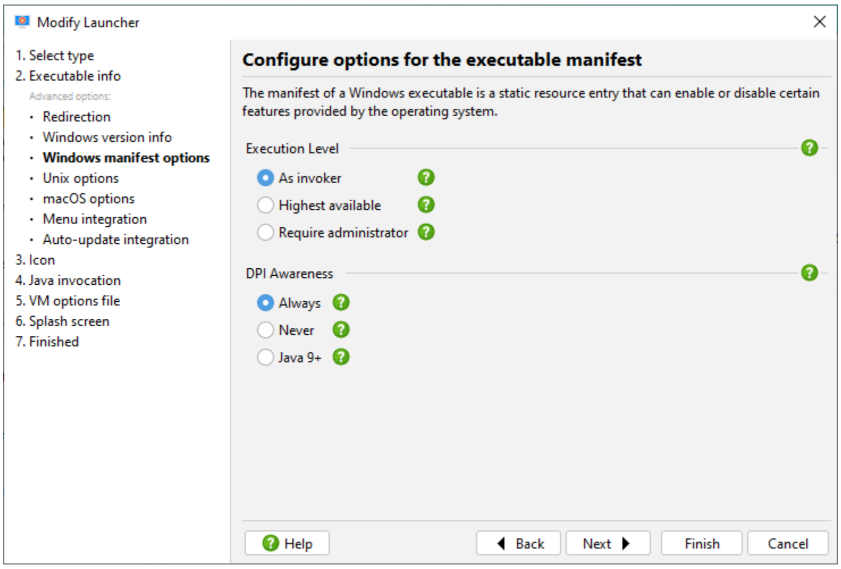
File version: [Default version]

Internal name: helloGUI

File description: Hello World Suite GUI Launcher

Legal copyright: Copyright ej-technologies GmbH, 2002-2003

On the "Executable->Windows manifest options" step you can adjust the contents of the executable manifest, a static resource in the executable that controls some Windows features.



Configure options for the executable manifest

The manifest of a Windows executable is a static resource entry that can enable or disable certain features provided by the operating system.

Execution Level

● As invoker
○ Highest available
○ Require administrator

DPI Awareness

● Always
○ Never
○ Java 9+

With an **execution level** other than "As invoker", you can ask Windows to show a UAC prompt and run the launcher with elevated privileges.

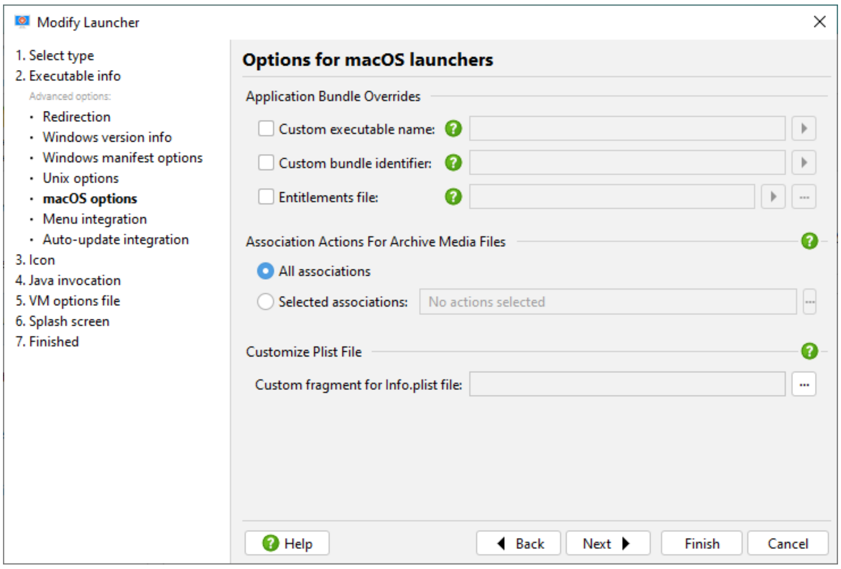
The **DPI awareness** controls whether Windows will scale up pixels in a GUI if high DPI is used. By default, DPI awareness is enabled if the minimum Java version of your project is at least Java 9.

On Windows, executables can be 64-bit or 32-bit. A 64-bit executable can only run with a 64-bit JVM and a 32-bit executable can only run with a 32-bit JVM. By default, 64-bit executables are generated, but you can switch to 32-bit executables in the "Installer options" step of the Windows media wizard.

macOS-specific features

By default, the generated application bundle for a GUI application uses the "Executable name" property from the "Executable info" step of the launcher wizard. If you choose compact names as appropriate for Windows and Unix, you may not be happy with the appearance in the Finder on macOS.

On the "Executable info->macOS options" step, you can specify a **localizable application bundle name**. If you specify an i18n variable as the application bundle name, such as `${i18n:myLauncherName}`, install4j will name the application bundle directory with the resolved value for the principal language [p. 79] of your project. In addition, it will take the values for all additional configured languages and set up the appropriate localization in the application bundle.



Options for macOS launchers

Application Bundle Overrides

☐ Custom executable name:

☐ Custom bundle identifier:

☐ Entitlements file:

Association Actions For Archive Media Files

● All associations

○ Selected associations: No actions selected

Customize Plist File

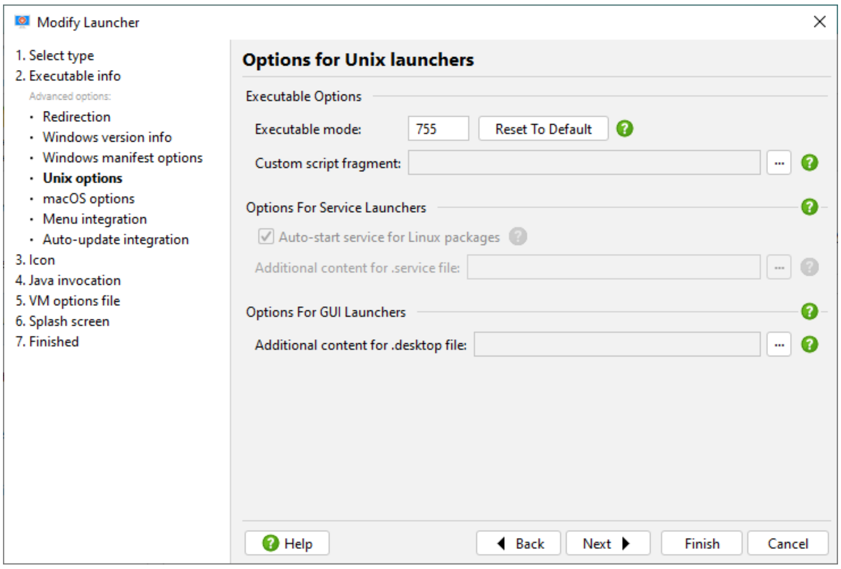
Custom fragment for Info.plist file:

On macOS, file associations and URL handlers are not registered with calls to an API that is provided by the operating system, but by adding special entries to the `Info.plist` file of the application bundle. This is why macOS single bundle archives can handle "Create a file association" and "Register a URL handler" actions at compile-time. By default, associations for all such actions that are contained in the installer configuration on the "Installer->Screens & Actions" step are added to the `Info.plist` file. Optionally, you can choose that only selected actions should be included.

Many advanced modifications of the behavior of an application bundle can be done by adding entries to the `Info.plist` file. On the macOS Options step you can specify a fragment that is added to the default `Info.plist` file. For services, this fragment is written to the launcher plist file.

Modifying launcher shell scripts and secondary start files

Launchers on Unix as well as command line and service launchers on macOS are shell scripts that invoke the standard Java launcher. To include your own modifications, you can specify a fragment that is inserted just before the `java` invocation.



Executable mode: 755

☑ Auto-start service for Linux packages

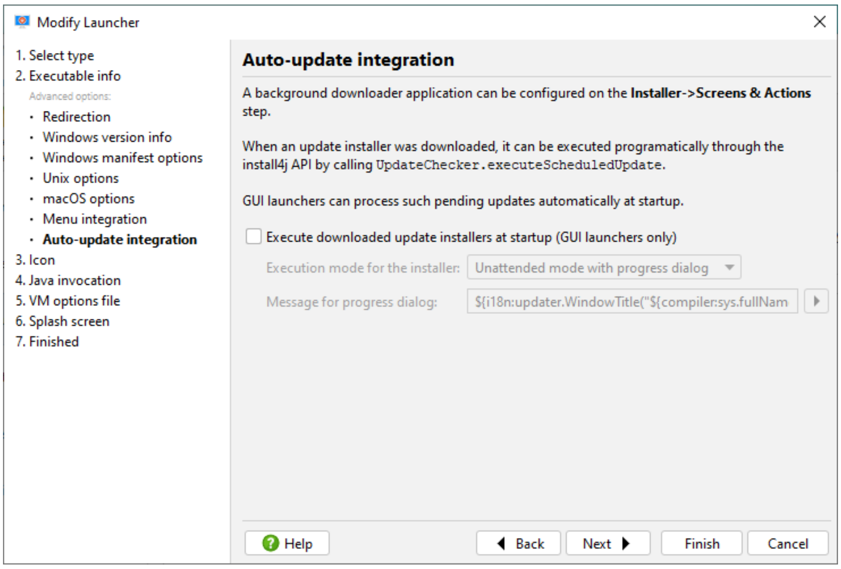
On Linux, two conditions require the generation of additional start files for a launcher and in both cases you can add additional content to them:

- The integration of a GUI launcher into a desktop environment requires the generation of a `.desktop` file. You may want to add additional content to that file to customize the interaction with the desktop environment.
- In the case of a service launcher, a `.service` file is generated if systemd is detected. To configure advanced aspects of systemd execution you can add additional content to that file.

Auto-update integration

In the Installer->Screens & Actions [p. 148] step, you can add a "Background updater" installer application that runs in the background and automatically downloads an updater installer. Such a background updater will not execute the downloaded update installer because that would disrupt the work of the user. Instead, it executes a "Schedule update installation" action to register the downloaded updated installer for later execution.

For GUI launchers, you can select the `Execute downloaded updater installers at startup` check box in the "Executable info->Auto update integration" step of the launcher wizard. When this GUI launcher is started and a downloaded update installer has been scheduled for installation, the update installer will be executed. By default, the execution mode of the update installer is set to "Unattended mode with progress dialog" with a configurable message.



Auto-update integration

A background downloader application can be configured on the **Installer->Screens & Actions** step.

When an update installer was downloaded, it can be executed programatically through the install4j API by calling `UpdateChecker.executeScheduledUpdate`.

GUI launchers can process such pending updates automatically at startup.

☐ Execute downloaded update installers at startup (GUI launchers only)

Execution mode for the installer: Unattended mode with progress dialog

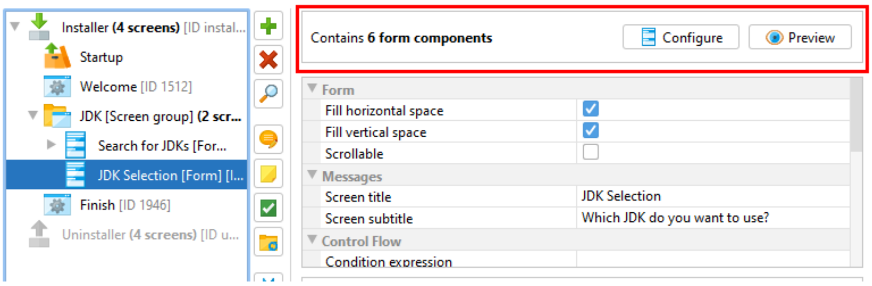
Message for progress dialog: ${i18n:updater.WindowTitle("${compiler:sys.fullNam

For more on auto-update functionality, see the corresponding help topic [p. 114].

A.8 Form Screens

Most screens in install4j contain a configurable form. In these screens, you can configure a list of form components [p. 183] along the vertical axis of the form. install4j provides you with properties to control the initialization of form components and the way the user selection is bound to installer variables [p. 63]. With this facility you can easily generate good-looking installer screens that display arbitrary data to the user and request arbitrary information to be entered.

Most standard screens are built with form components and form templates are starting points for your own customizations. Also, you can add empty form screens and add form components to them. For screens that have a configurable form, a header is shown above the screen configuration [p. 163] that shows the number of contained form components as well as buttons for editing them and showing a preview of the form.

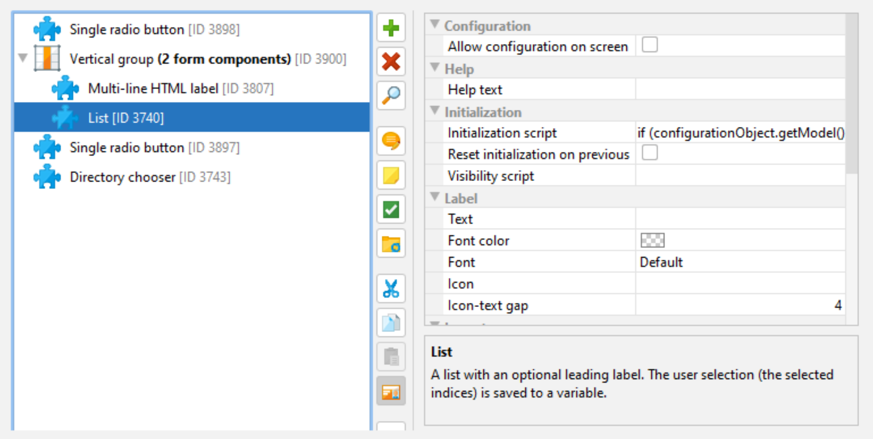


- Installer **(4 screens)** [ID instal...
  - Startup
  - Welcome [ID 1512]
  - JDK [Screen group] **(2 scr...**
    - Search for JDKs [For...
    - JDK Selection [Form] [I...
  - Finish [ID 1946]
- Uninstaller **(4 screens)** [ID u...

Contains **6 form components**

Form	
Fill horizontal space	☑
Fill vertical space	☑
Scrollable	☐
Messages	
Screen title	JDK Selection
Screen subtitle	Which JDK do you want to use?
Control Flow	
Condition expression	

The actual configuration of the form components is performed in a separate dialog:



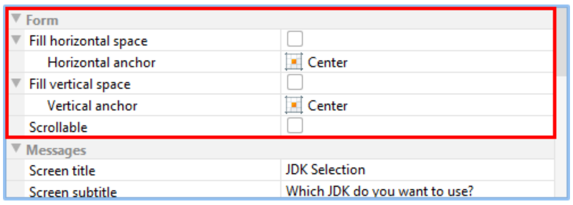
- Single radio button [ID 3898]
- Vertical group **(2 form components)** [ID 3900]
  - Multi-line HTML label [ID 3807]
  - List [ID 3740]
- Single radio button [ID 3897]
- Directory chooser [ID 3743]

Configuration	
Allow configuration on screen	☐
Help	
Help text	
Initialization	
Initialization script	if (configurationObject.getModel()
Reset initialization on previous	☐
Visibility script	
Label	
Text	
Font color	▨
Font	Default
Icon	
Icon-text gap	4

List

A list with an optional leading label. The user selection (the selected indices) is saved to a variable.

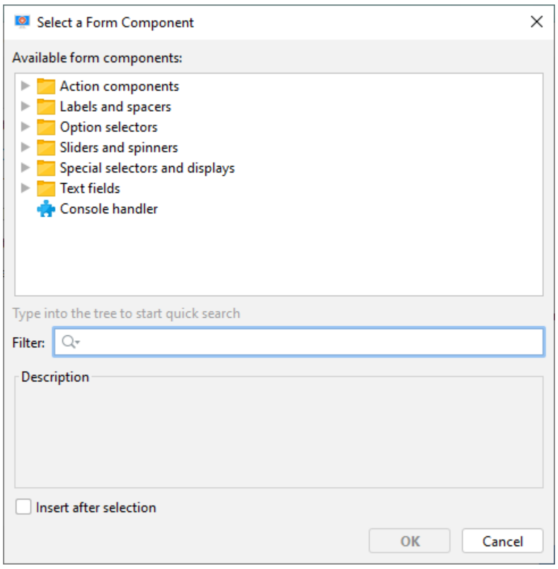
Screens can layout the contained form in different ways, but for plain form screens, you can configure this with properties of the containing screen. By default, a form is top-aligned and fills the entire available horizontal space. For example, for a set of radio buttons that should be centered horizontally and vertically, the "Fill horizontal space" and "Fill vertical space" properties of the screen must be set to "false" and the horizontal and vertical anchor properties must be set to "Center".



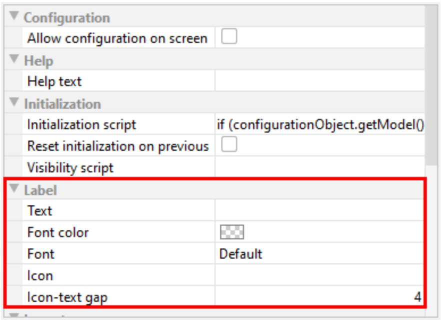
Form	
Fill horizontal space	☐
Horizontal anchor	Center
Fill vertical space	☐
Vertical anchor	Center
Scrollable	☐
Messages	
Screen title	JDK Selection
Screen subtitle	Which JDK do you want to use?

Form components

install4j offers a large number of form components that represent most common components available in Java and some other special components that are useful in the context of an installer.



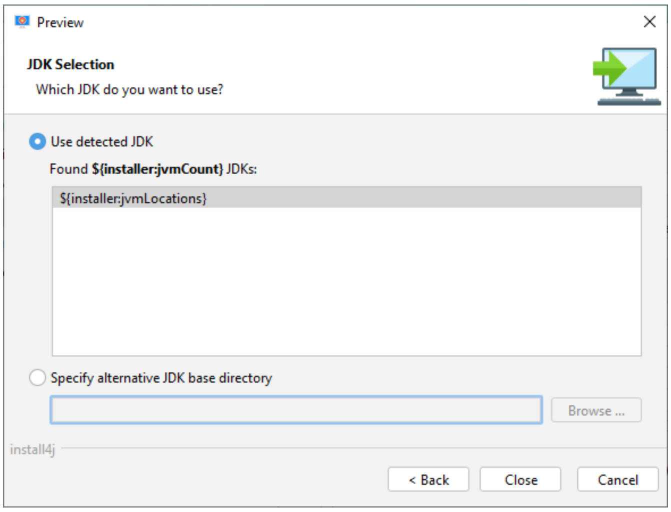
All components that expect user input have an optional leading label. The components themselves are left-aligned on the entire form. If you leave the label text empty, the form component will occupy the entire horizontal space of the form.



Configuration	
Allow configuration on screen	☐
Help	
Help text	
Initialization	
Initialization script	if (configurationObject.getModel()
Reset initialization on previous	☐
Visibility script	
Label	
Text	
Font color	
Font	Default
Icon	
Icon-text gap	4

Every form component has configurable insets. For vertical gaps that are meant to separate groups of form components, consider using a "Vertical spacer" form component since it makes the grouping clearer and allows to to reorder form components more easily.

You can preview your form at any time with the *Preview Form* button. The preview dialog performs all variable replacements of compiler variables and custom localization keys, but not of installer variables. Also, no initialization scripts or screen activation scripts are run. The preview is intended to give you quick feedback about visual aspects of your form. At runtime, the look and feel may be different.



JDK Selection

Which JDK do you want to use?

● Use detected JDK

Found **${installer:jvmCount}** JDKs:

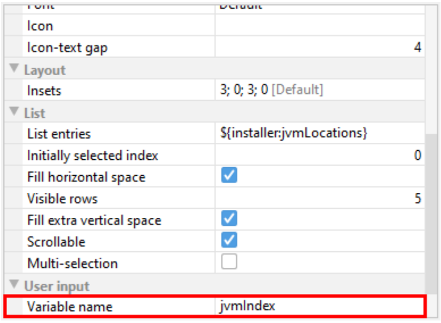
${installer:jvmLocations}

○ Specify alternative JDK base directory

Every form component always has its preferred vertical height. For some form components such as the "List" form component, this preferred vertical size is configurable. If the vertical extent of the form exceeds the available vertical space, a scroll bar is shown. If you want such a form component to fill the entire available vertical space, you can select the "Fill vertical space" property for the form component and deselect the "Scrollable" property of the form screen. In that case, there will be no scroll bar for the form.

User input

If a form component can accept user input, you need some way to access the user selection afterwards. install4j saves user input for such form components to the installer variable [p. 63] whose name is specified in the "Variable name" property. That variable can then be used later on, for example in condition expressions for screens and actions.



Icon	
Icon-text gap	4
▼ Layout	
Insets	3; 0; 3; 0 [Default]
▼ List	
List entries	${installer:jvmLocations}
Initially selected index	0
Fill horizontal space	☑
Visible rows	5
Fill extra vertical space	☑
Scrollable	☑
Multi-selection	☐
▼ User input	
Variable name	jvmIndex

If you have a check box that saves its user input to a variable called "userSelection", the condition expression

```
context.getBooleanVariable("userSelection")
```

will skip the screen or action for which that condition expression is used. The user selection in form components is written to the variables before the validation expression for the screen is called. If you have a text field that saves its input to the variable "fileName", the validation expression

```
Util.showOptionDialog("Do you really want to delete " + context.getVariable("fileName"),
    new String[] {"Yes", "No"}, JOptionPane.QUESTION_MESSAGE) == 0
```

used on the same screen will block the advance to the next screen if the user answers with "No".

The values of installer variables accommodate the general type `java.lang.Object`. Every form component saves its user input in its naturally corresponding data type, for example:

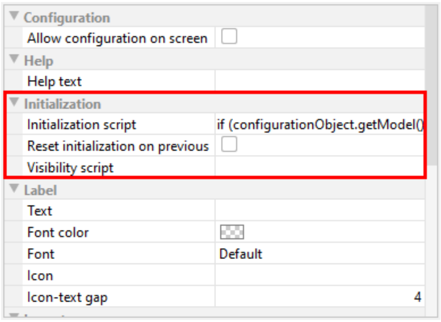
- For check boxes, the type `java.lang.Boolean` is used. For this special case the context offers the convenience method `getBooleanVariable`.
- For text fields, the type `java.lang.String` is used.
- For drop down lists the type `java.lang.Integer` is used to save the selected index.
- For date spinners, the type `java.lang.Date` is used.

The description of the value type for each form component that accepts user input is shown in the registry dialog when you select the form component.

Initialization

For each form component, install4j offers several properties that allow you to customize its initial state. However, you may want to access the properties of the underlying UI component or use a more complex logic for modifying the form component.

For this purpose, the "Initialization script" property is provided. Form components can expose a well-known component in the initialization script that allows you to perform these modifications. This so-called "configuration object" is usually contained in the form component itself. For example a "Check box" form component exposes a `configurationObject` parameter of type `javax.swing.JCheckBox` and a "Text field" form component exposes a `javax.swing.JTextField`.



Configuration	
Allow configuration on screen	☐
Help	
Help text	
Initialization	
Initialization script	if (configurationObject.getModel()
Reset initialization on previous	☐
Visibility script	
Label	
Text	
Font color	▨
Font	Default
Icon	
Icon-text gap	4

As with actions and screens [p. 24] in general, the possibility that the user moves back and forth in the screen sequence presents a dilemma to install4j. Any form components that accepts user input has a configurable initial value and any form component can have an initialization script. This initialization is performed when the user enters the screen for the first time. Should this initialization be performed again when the user moves back and then enters the screen once again? Since install4j does not know, it initializes every form component only once by default. This policy can be changed with the "Reset initialization on previous" property for each form component.

Depending on factors such as the correct platform, user input in the previous screen or whether the installer runs in console mode, some form components may not be applicable and should be hidden. In the "Visibility script", you can detect such conditions and return `false` to hide the form components.

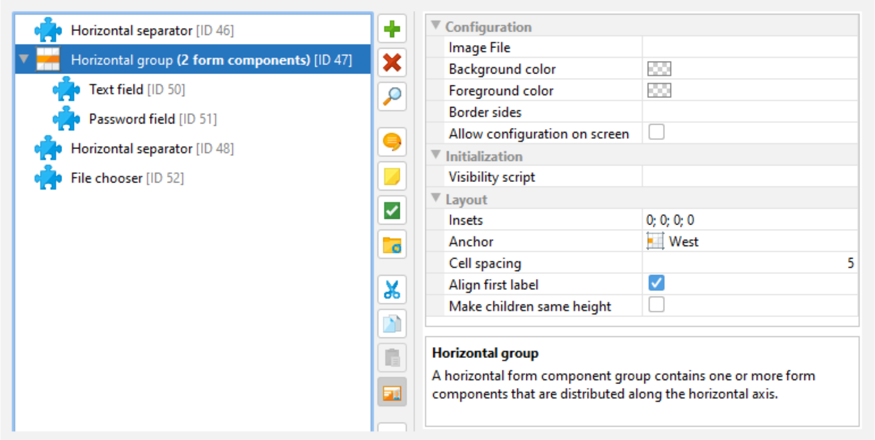
A.9 Layout Groups

A layout group is an element in a form screen [p. 46]. It contains a number of form components and other layout groups. With layout groups you can achieve virtually any kind of visual layout.

There are two different kinds of layout groups: vertical and horizontal groups. A horizontal group puts the contained elements side by side, while a vertical group organizes them from top to bottom. Essentially, the top-level of a form screen is a vertical layout group itself.

Use case: Side by side

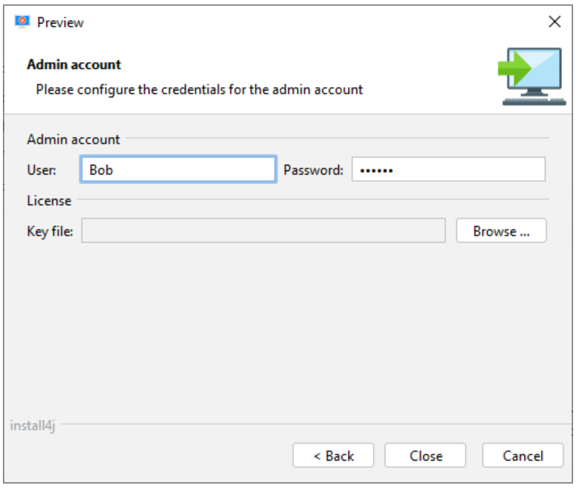
Putting two form components side by side is done with a single horizontal group:



- Horizontal separator [ID 46]
- Horizontal group **(2 form components)** [ID 47]
  - Text field [ID 50]
  - Password field [ID 51]
- Horizontal separator [ID 48]
- File chooser [ID 52]

Configuration	
Image File	
Background color	▨
Foreground color	▨
Border sides	
Allow configuration on screen	☐
Initialization	
Visibility script	
Layout	
Insets	0; 0; 0; 0
Anchor	West
Cell spacing	5
Align first label	☑
Make children same height	☐

Horizontal group
A horizontal form component group contains one or more form components that are distributed along the horizontal axis.



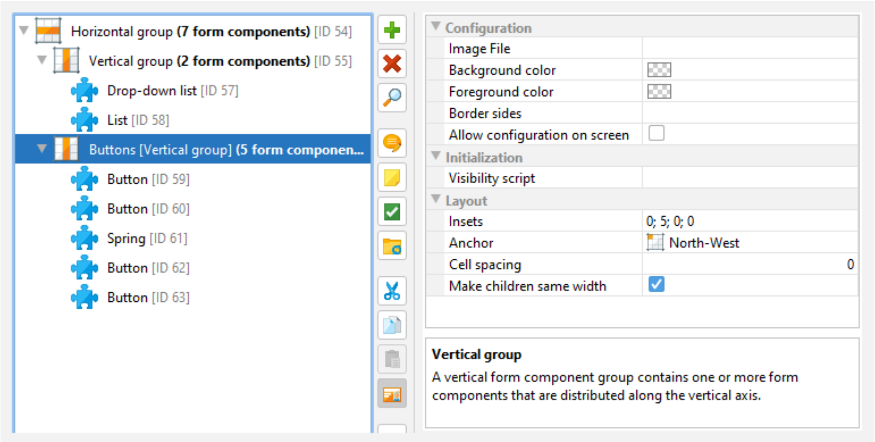
Admin account
Please configure the credentials for the admin account

Admin account
User: Bob Password: ••••••
License
Key file:

The leading labels of the first form component in the horizontal layout group ("User:") and those of the form components on the same level as the horizontal group ("Key file:") are aligned. There is a property on the horizontal layout group to switch off this alignment.

Use case: Two columns

Two columns of form components are realized with two vertical layout groups inside a horizontal layout group:

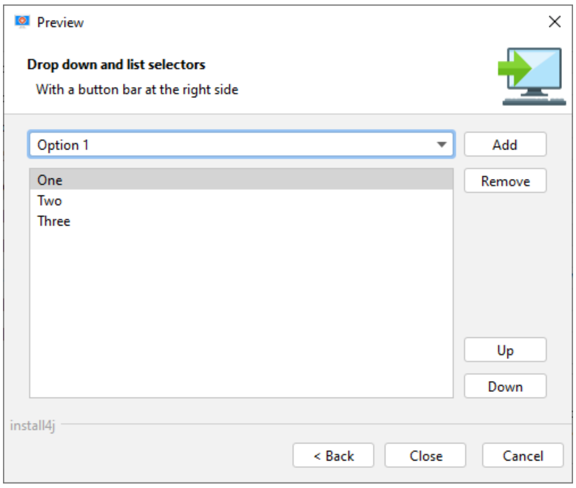


- Horizontal group **(7 form components)** [ID 54]
  - Vertical group **(2 form components)** [ID 55]
    - Drop-down list [ID 57]
    - List [ID 58]
  - Buttons [Vertical group] **(5 form componen...**
    - Button [ID 59]
    - Button [ID 60]
    - Spring [ID 61]
    - Button [ID 62]
    - Button [ID 63]

Configuration	
Image File	
Background color	▨
Foreground color	▨
Border sides	
Allow configuration on screen	☐
Initialization	
Visibility script	
Layout	
Insets	0; 5; 0; 0
Anchor	North-West
Cell spacing	0
Make children same width	☑

Vertical group

A vertical form component group contains one or more form components that are distributed along the vertical axis.



Preview

Drop down and list selectors

With a button bar at the right side

Option 1

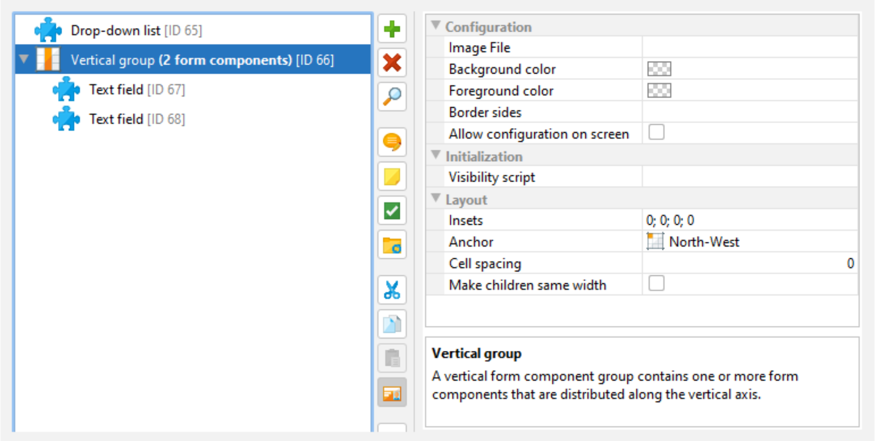
One
Two
Three

install4j

In this case the second column with the buttons takes up a fixed amount of horizontal space, because buttons do not automatically grow beyond their preferred size. In order to make all buttons of equal size, the "Make children same width" property has been selected. Two buttons are aligned at the top of the column, two buttons at the bottom. This is achieved with a "Spring" form component after the second button that has its axis set to "Vertical". It pushes all further components to the bottom.

Use case: Breaking label alignment

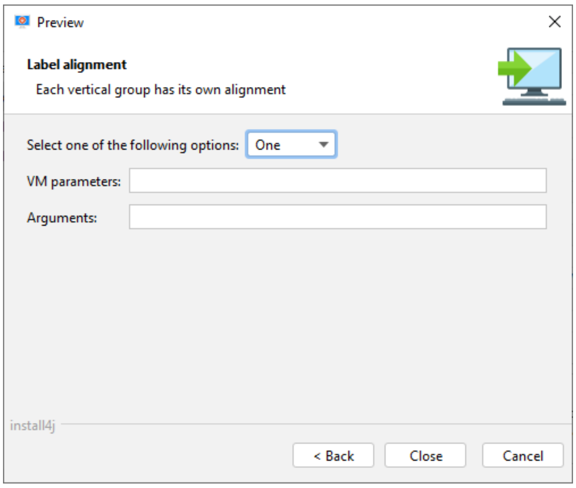
Alignment of leading labels can be broken by introducing vertical layout groups:



Drop-down list [ID 65]
Vertical group **(2 form components)** [ID 66]
Text field [ID 67]
Text field [ID 68]

Configuration	
Image File	
Background color	
Foreground color	
Border sides	
Allow configuration on screen	☐
Initialization	
Visibility script	
Layout	
Insets	0; 0; 0; 0
Anchor	North-West
Cell spacing	0
Make children same width	☐

Vertical group
A vertical form component group contains one or more form components that are distributed along the vertical axis.



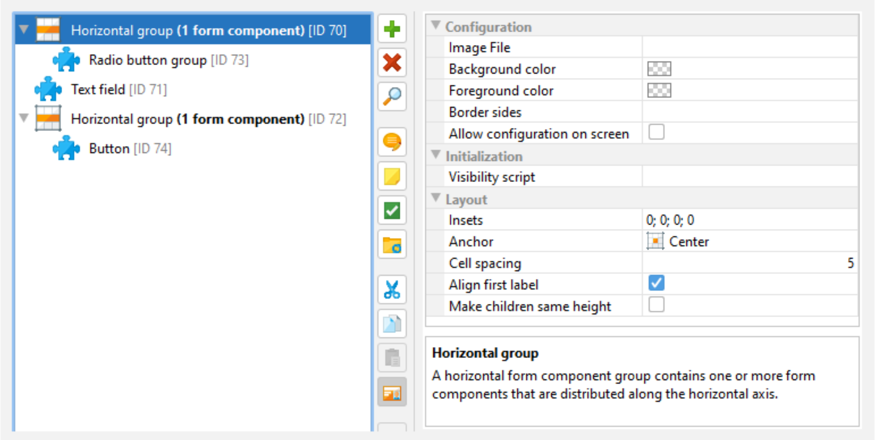
Label alignment
Each vertical group has its own alignment

Select one of the following options: One
VM parameters:
Arguments:

Here, the long leading label of the first form component does not enlarge the leading labels of the two text field form components. The latter are aligned only among themselves.

Use case: Center and right alignment

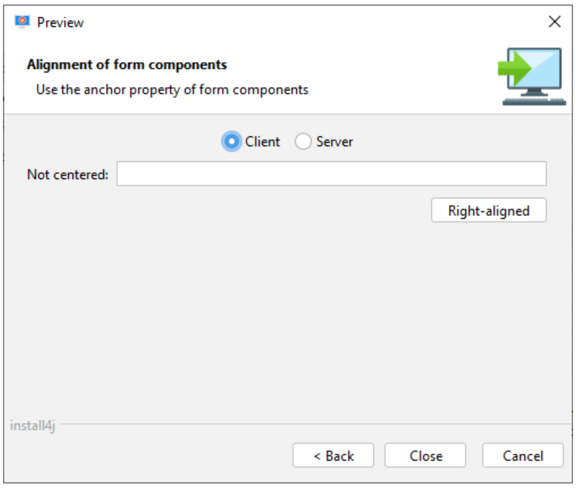
Single form components can be centered or right-aligned if you enclose them in a horizontal layout group and set the "Anchor" property on the layout group accordingly.



Horizontal group **(1 form component)** [ID 70]
Radio button group [ID 73]
Text field [ID 71]
Horizontal group **(1 form component)** [ID 72]
Button [ID 74]

Configuration	
Image File	
Background color	
Foreground color	
Border sides	
Allow configuration on screen	☐
Initialization	
Visibility script	
Layout	
Insets	0; 0; 0; 0
Anchor	Center
Cell spacing	5
Align first label	☑
Make children same height	☐

Horizontal group
A horizontal form component group contains one or more form components that are distributed along the horizontal axis.



For the layout group with the radio button group, the anchor has been set to "Center", for that with the button the anchor has been set to "East". This only works with form components that do not grow horizontally. Some form components that do grow horizontally can be restricted to a fixed horizontal size, such as the text field by specifying a non-zero column count.

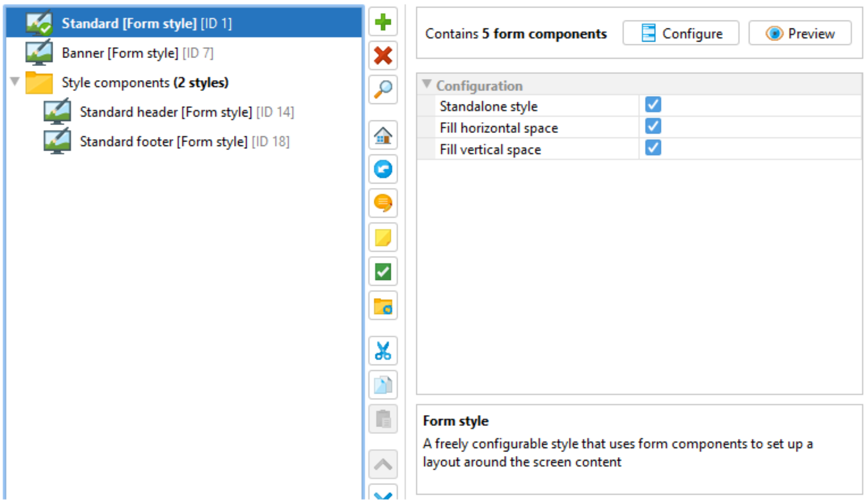
A.10 Styles

Install4j has a flexible model for styling the UI of installer applications that allows you to arrange content and styling elements in arbitrary ways. While there is an API to do this programatically, you can configure form styles in the install4j IDE without any custom code. Form styles use the same foundation as form components [p. 183] for screens. All default styles are created with form styles, so the details of the default styles can we tweaked very easily and new styles can be developed by starting with the default styles.

Configuring styles

Styles are configured on a per-project basis. On the "Installer->Screens & Actions->Styles" step of the install4j IDE, all available styles are listed. When you add a style, it can either be a configurable form style, or a style implementation from your custom code. Styles are either standalone or not. A non-standalone style cannot be used directly, but is only available for nesting into other styles.

One single style is marked as the default style and is shown with a bold font. With the "Set As Default" action you can change the default style. Styles can be grouped into folders for organizing them according to your individual preferences. For example, in the default styles, the nested styles are grouped into a separate folder whereas the standalone styles are located at the top level.



Standard [Form style] [ID 1]
Banner [Form style] [ID 7]
Style components **(2 styles)**
- Standard header [Form style] [ID 14]
- Standard footer [Form style] [ID 18]

Contains **5 form components**

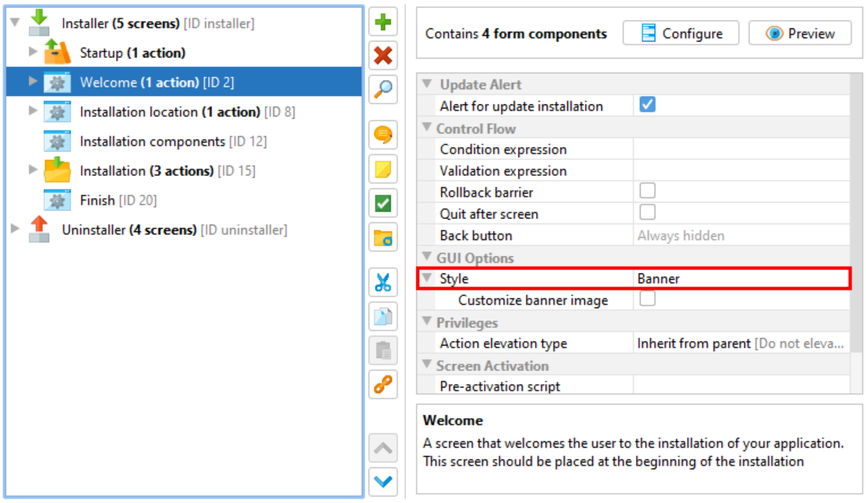
Configuration

Standalone style	☑
Fill horizontal space	☑
Fill vertical space	☑

Form style

A freely configurable style that uses form components to set up a layout around the screen content

On the "Installer->Screens & Actions" step of the install4j IDE, you can apply styles. Installer applications, screen groups and screens all have a "Style" property. For installer applications, this is property is set to "Default". You can change it to any standalone style. For screen groups and screens, the "Style" property is set to "Inherit from parent". The property also indicates which style is actually inherited. Alternatively, you can choose to explicitly set a style for the selected element. Any screen groups and screens below it will now inherit this style.



Installer (5 screens) [ID installer]
  Startup (1 action)
  Welcome (1 action) [ID 2]
  Installation location (1 action) [ID 8]
  Installation components [ID 12]
  Installation (3 actions) [ID 15]
  Finish [ID 20]
Uninstaller (4 screens) [ID uninstaller]

Contains 4 form components

Update Alert	
Alert for update installation	☑
Control Flow	
Condition expression	
Validation expression	
Rollback barrier	☐
Quit after screen	☐
Back button	Always hidden
GUI Options	
Style	Banner
Customize banner image	☐
Privileges	
Action elevation type	Inherit from parent [Do not eleva...
Screen Activation	
Pre-activation script	

Welcome

A screen that welcomes the user to the installation of your application. This screen should be placed at the beginning of the installation

Some screens have a preference for a particular style. For example, the "Welcome" and "Finish" screens want their style set to "Banner". When adding such a screen, the IDE matches the style by name. In this example, if no style named "Banner" is available, the default style is used. Otherwise, install4j keeps track of style associations by ID and you can rename styles without breaking any associations.

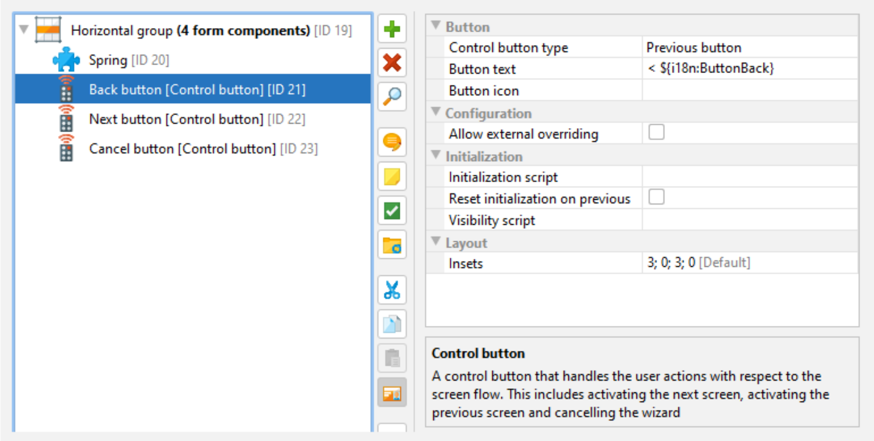
If you delete a style, all its style associations are broken. Compiling the installer will now fail and you will have to visit all installer applications, screen groups and screens where this style was explicitly selected and choose a new style.

Should you want to return to the default styles, there is a "Reset Styles To Default" action for that purpose. Existing style associations are matched by name in that case, so style associations with the "Banner" style survive this reset, for example.

Form styles

A restricted set of the form components that are available for building form screens [p. 183] can be used to build form styles. Form components that take user input are not suitable for styles because styles have a different life-cycle than screens.

In addition, form styles can use a set of special form components. The "Screen content" form component contains the UI component of the screen and is changed each time when a screen is activated. When you preview the style, this content area is shown with a placeholder. The "Screen Title" form component shows the title or the subtitle of the screen, depending on its "Title type" property. The "Control button" form component is used for realizing the "Next", "Previous" and "Cancel" buttons.



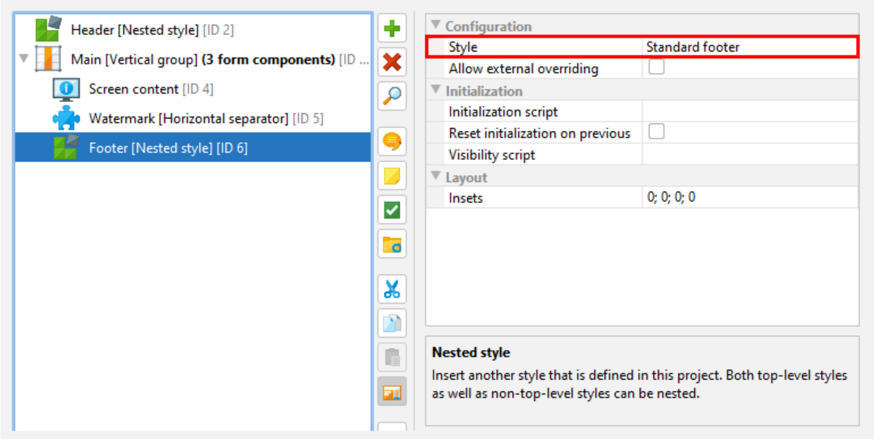
Horizontal group **(4 form components)** [ID 19]
- Spring [ID 20]
- Back button [Control button] [ID 21]
- Next button [Control button] [ID 22]
- Cancel button [Control button] [ID 23]

Button	
Control button type	Previous button
Button text	< ${i18n:ButtonBack}
Button icon	
Configuration	
Allow external overriding	☐
Initialization	
Initialization script	
Reset initialization on previous	☐
Visibility script	
Layout	
Insets	3; 0; 3; 0 [Default]

Control button

A control button that handles the user actions with respect to the screen flow. This includes activating the next screen, activating the previous screen and cancelling the wizard

Finally, the "Nested style" form component allows you to embed another style. In this way you can build a set of styles that share common parts. For example, in the default styles, the navigation buttons at the bottom are the same. With the "Standard Footer" style that is used by both the "Standard" and the "Banner" standalone styles, you have a single place to change its settings.



- Header [Nested style] [ID 2]
- Main [Vertical group] **(3 form components)** [ID ...
  - Screen content [ID 4]
  - Watermark [Horizontal separator] [ID 5]
  - Footer [Nested style] [ID 6]

Configuration	
Style	Standard footer
Allow external overriding	☐
Initialization	
Initialization script	
Reset initialization on previous	☐
Visibility script	
Layout	
Insets	0; 0; 0; 0

Nested style

Insert another style that is defined in this project. Both top-level styles as well as non-top-level styles can be nested.

Graphical styling elements

A key concern of styling is the placement of images, either in the foreground or in the background. Both kinds of placements are handled by layout groups in form styles. For both vertical and horizontal form groups, setting their "Image file" property shows additional properties that allow you to place the image in the layout group. If you place the image in the foreground, it cuts off an entire edge of the rectangle that can get its own background and border. In that way, the image can blend seamlessly into its surroundings.

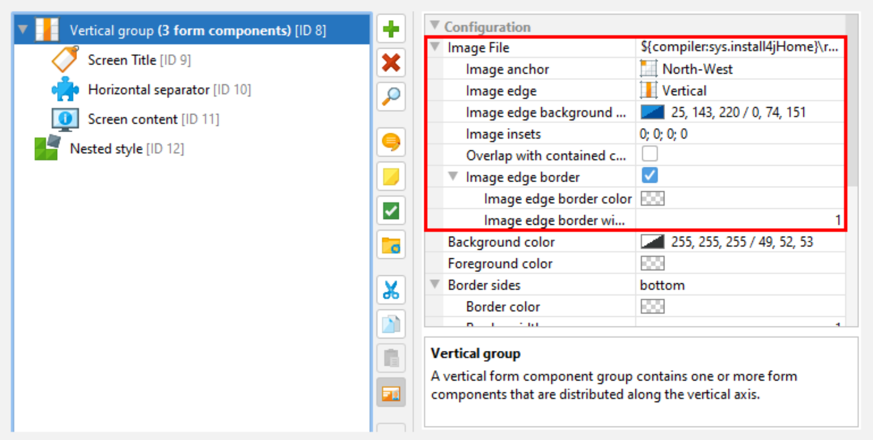


Image File	${compiler:sys.install4jHome}\r...
Image anchor	North-West
Image edge	Vertical
Image edge background ...	25, 143, 220 / 0, 74, 151
Image insets	0; 0; 0; 0
Overlap with contained c...	☐
Image edge border	☑
Image edge border color	▨
Image edge border wi...	1
Background color	255, 255, 255 / 49, 52, 53
Foreground color	▨
Border sides	bottom
Border color	▨

Vertical group

A vertical form component group contains one or more form components that are distributed along the vertical axis.

To place an image into the flow of form components, you can use the "Image insets" property and set its "Icon" property.

Other important styling elements are borders and separators. Again, this is handled by layout groups. With their "Border sides" property you can define which sides of the border should be drawn. Color and thickness of borders are also configurable.

By default, layout groups and form components are transparent, so that the default background color of the window shines through. By setting the "Background color" property of a layout group, you can make it opaque and give it a specific color. The "Foreground color" property sets the font color for contained form components that do not have their color set explicitly.

Overriding properties

Some styles can have elements that are specific to particular screens or particular installer applications. For example, the header image in the "Standard" style or the banner image of the "Banner" style could be required to change for each screen. Instead of duplicating styles in this scenario, install4j allows you to designate certain properties of selected form components and layout groups that should be overridable when the style is applied.

When editing the form components of a form style, each form component has an "Allow external overriding" property. If you select that property, a named overriding entry will be offered when you explicitly apply the style on the "Installer->Screens & Actions" step. With the "Override title" property, you specify the displayed name for the override entry and that name is used for saving the overridden properties. This means that the name must be unique for a single style and that overrides are lost if you change the name. The "Property selection mode" property then lets you select which properties should be overridable, either all properties are overridable, or a list of properties is included or excluded.

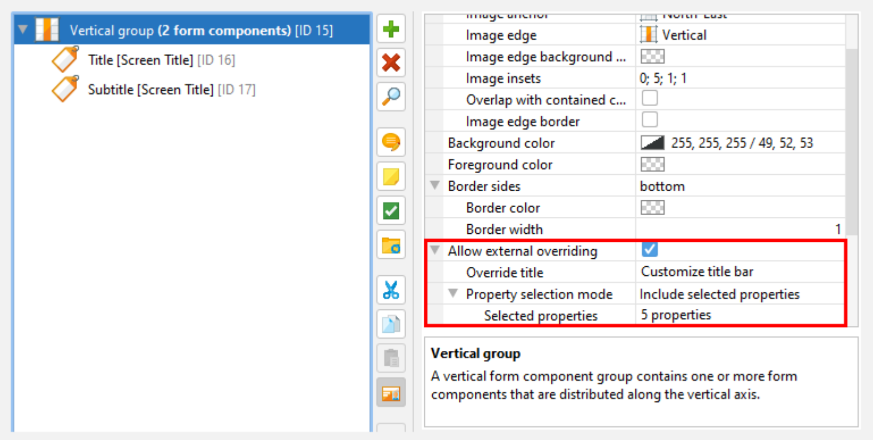
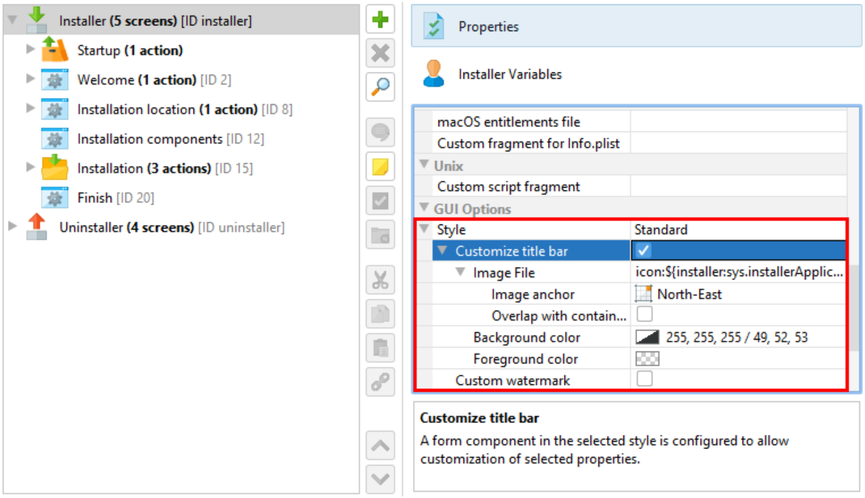

Vertical group **(2 form components)** [ID 15]
- Title [Screen Title] [ID 16]
- Subtitle [Screen Title] [ID 17]

Image edge	Vertical
Image edge background ...	▨
Image insets	0; 5; 1; 1
Overlap with contained c...	☐
Image edge border	☐
Background color	255, 255, 255 / 49, 52, 53
Foreground color	▨
Border sides	bottom
Border color	▨
Border width	1
Allow external overriding	☑
Override title	Customize title bar
Property selection mode	Include selected properties
Selected properties	5 properties

Vertical group

A vertical form component group contains one or more form components that are distributed along the vertical axis.

When you select a style on the "Installer->Screens & Actions" step, install4j scans the style and all its nested styles for form components and layout groups with defined overrides. Each named override is presented as a check box property. If you select the check box, the overridable properties of the form component or layout group are copied and displayed as child properties. You can now change the properties to different values. Note that the overridable properties lose their connection to the default values in the original form component or layout group. If you change a default property value, you have to manually change it in all overrides, if necessary.


Installer **(5 screens)** [ID installer]
- Startup **(1 action)**
- Welcome **(1 action)** [ID 2]
- Installation location **(1 action)** [ID 8]
- Installation components [ID 12]
- Installation **(3 actions)** [ID 15]
- Finish [ID 20]

Uninstaller **(4 screens)** [ID uninstaller]

macOS entitlements file	
Custom fragment for Info.plist	
Unix	
Custom script fragment	
GUI Options	
Style	Standard
Customize title bar	☑
Image File	icon:${installer:sys.installerApplic...
Image anchor	North-East
Overlap with contain...	☐
Background color	255, 255, 255 / 49, 52, 53
Foreground color	▨
Custom watermark	☐

Customize title bar

A form component in the selected style is configured to allow customization of selected properties.

For more complex overriding cases, consider adding a "Nested style" form component and making its "Style" property overridable. When applying such a style, you can substitute a different nested style as appropriate.

API

Under some circumstances, styles are more easily implemented with the API. For example, if you want to have configurable properties that determine the construction of the style or if the styling cannot be realized with the facilities of the form style.

The sample project "customCode" includes a style class `SunnySkyBackgroundStyle` and its associated BeanInfo `SunnySkyBackgroundStyleBeanInfo` that show such an example style. It paints a background image that depends on the window dimensions and continues up to the window border. In the "customCode" project, look for the "Configurable form" screen in in the installer and preview the form in order to see what it looks like.

That example also shows how to implement a style that wraps a user-selectable style. The main style is still the standard style and the "Sunny sky background" style takes the function of a decorator. To make development of such wrappers easier, the API includes a convenience class `com.install4j.api.styles.WrapperStyle`.

Merging styles from other projects

Instead of duplicating styles across projects, you can develop them in one project and merge them into other projects. The merge projects functionality [p. 108] in install4j includes an option to merge styles.

If styles are merged, the "Style" property of installer applications, screen groups and screens shows the merged styles as well, with their names prefixed with the project name that was assigned in the merge settings.

If you link to screens or screen groups of merged projects, they will use their configured styles from the merged project only if style merging is enabled. Otherwise, install4j tries to match a style by name in the main project.

Overriding standard icons

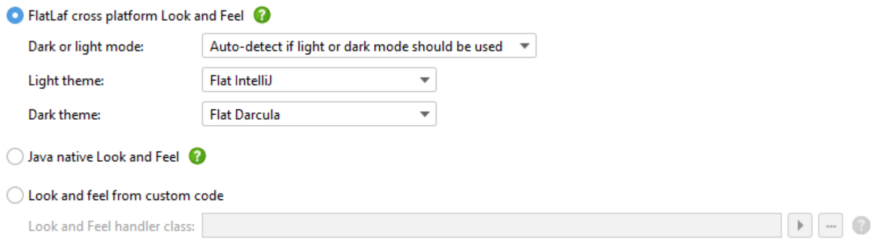
If you would like to change the standard icons in the installer, have a look at the JAR file `resource/i4jruntime.jar` in the install4j installation directory. The package `com.install4j.runtime.installer.frontend.icons` contains all icons that are used by the installer. To replace some or all of these icons with your own version, create a JAR file that contains just the new icon files in the same directory and add it on the "Installer->Screens & Actions->Custom Code" step. The installer will first try to load an icon from the custom code. Failing that, it will fall back to the built-in version.

A.11 Look & Feel

The GUI of the installer, uninstaller and other installer applications is implemented with Java Swing. Swing is themeable and so install4j can offer you choices for the look and feel of the applications that are provided by the runtime. The generated launchers are not affected by these settings.

Configuring the look & feel

The options for the look & feel can be adjusted on the "Installer->Screens & Actions->Look & Feel" step.



● FlatLaf cross platform Look and Feel

Dark or light mode: Auto-detect if light or dark mode should be used

Light theme: Flat IntelliJ

Dark theme: Flat Darcula

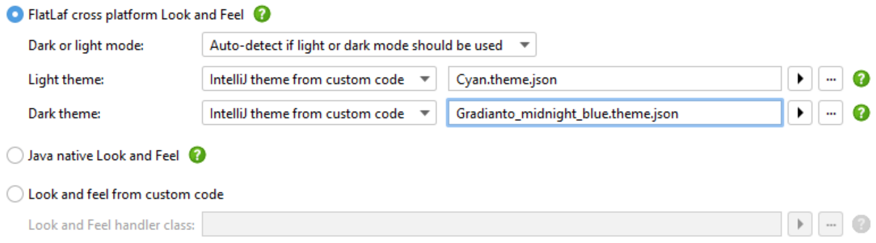
○ Java native Look and Feel

○ Look and feel from custom code

Look and Feel handler class:

The default setting is to use the FlatLaf (1) cross platform Look and Feel which is a flat Look and Feel that works well on all supported platforms and includes a dark mode. Please consider starring it on GitHub (2) as a token of appreciation for the author.

FlatLaf includes four built-in themes, two for light mode and two for dark mode. By default, the themes that look like the IntelliJ IDEA light and dark themes are selected. In addition, FlatLaf supports custom IntelliJ themes. These are based on JSON files and can override UI colors. You can download an IntelliJ theme (3) from the JetBrains plugin repository and add its JAR files on the "Installer->Screens & Actions->Custom Code" step. If the themes plugin is packaged in a ZIP file, you have to extract the ZIP file and add the contained JAR files instead. The contained themes will then show up in the chooser dialog.



● FlatLaf cross platform Look and Feel

Dark or light mode: Auto-detect if light or dark mode should be used

Light theme: IntelliJ theme from custom code | Cyan.theme.json

Dark theme: IntelliJ theme from custom code | Gradianto_midnight_blue.theme.json

○ Java native Look and Feel

○ Look and feel from custom code

Look and Feel handler class:

---

(1) https://www.formdev.com/flatlaf/

(2) https://github.com/JFormDesigner/FlatLaf

(3) https://plugins.jetbrains.com/search?tags=Theme

On Windows 10+ and macos 10.14+, the runtime detects whether dark mode is being used and activates it automatically. If the user switches between light and dark mode, the runtime adjusts to it on the fly. The look and feel configuration offers options to prevent this auto-detection and use either light or dark mode.

For backwards compatibility, you can also select the "Java native look and feel". This is a look and feel that is included the JRE and tries to mimic the native widgets of the operating system with varying success. In some instances, this look and feel may seem out of place as it shows the UI from an older version of the operating system. Also, HiDPI resolutions may not be well supported by this look and feel. For these reasons, using the native look and feel is discouraged and the FlatLaf cross-platform look and feel is recommended instead.

Using a custom look and feel

You can apply your own look and feel by extending the `com.install4j.api.laf.LookAndFeelHandler` class in the install4j API. After adding the compiled class and its dependencies on the "Installer->Screens & Actions->Custom Code" step, you can select the class in the chooser dialog.

The `com.install4j.api.laf.LookAndFeelHandler` implements the `com.install4j.api.laf.LookAndFeelEnhancer` interface that contains methods that help with certain aspects of creating the UI. You can override these methods to change their default behavior.

For example, a tri-state check box is required by the UI of installer applications. Java Swing does not include such a component, but some look and feels add this feature. To avoid using a generic simulation of a tri-state checkbox, the `createTriStateCheckbox` method can be overridden in your implementation of the `com.install4j.api.laf.LookAndFeelHandler`.

A.12 Variables

With variables you can customize many aspects of install4j. They can be used in all text fields and text properties in the install4j IDE as well as from the install4j API [p. 212]. The general variable syntax is

```
${prefix:variableName}
```

where prefix denotes the variable type and is one of

- **compiler**

  Compiler variables are replaced by the install4j compiler when the project is built.
- **installer**

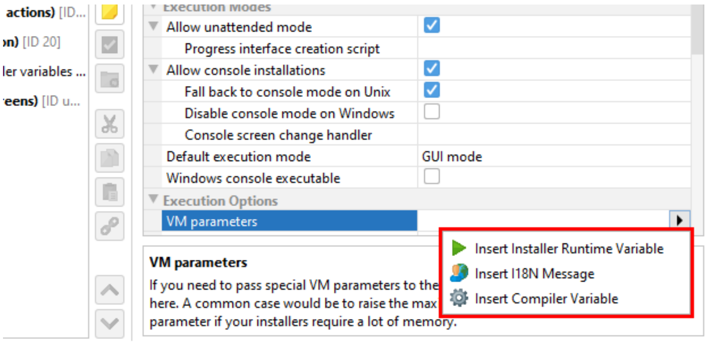
  Installer variables are evaluated when the installer or uninstaller is running.
- **launcher**

  Launcher variables are evaluated when a generated application launcher is started.
- **i18n**

  Custom localization keys are evaluated at runtime and depend on the chosen installer language.
- **(no prefix)**

  Variables with no prefix resolve to runtime environment variables when used in the launcher configuration.

Text fields in the install4j IDE where you can use variables have a ▶ variable selector next to them. In the popup menu, you first choose a variable system from the available variable types. In text properties of an installer element [p. 148] or a form component [p. 183], you can use compiler variables, installer variables and custom localization keys, but not launcher variables.

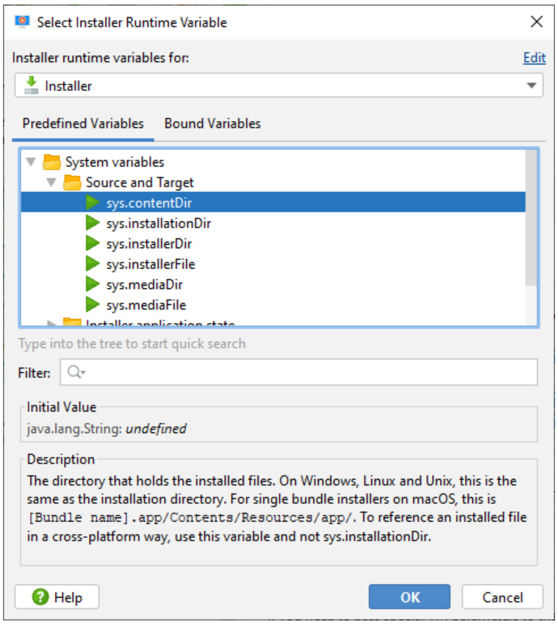


Allow unattended mode	☑
Progress interface creation script	
Allow console installations	☑
Fall back to console mode on Unix	☑
Disable console mode on Windows	☐
Console screen change handler	
Default execution mode	GUI mode
Windows console executable	☐
VM parameters	▶

VM parameters

If you need to pass special VM parameters to the here. A common case would be to raise the max parameter if your installers require a lot of memory.

The variable selection dialog then shows all known variables of the selected variable type.



Installer runtime variables for: Installer

Filter:

Initial Value

java.lang.String: *undefined*

Description

The directory that holds the installed files. On Windows, Linux and Unix, this is the same as the installation directory. For single bundle installers on macOS, this is `[Bundle name].app/Contents/Resources/app/`. To reference an installed file in a cross-platform way, use this variable and not sys.installationDir.

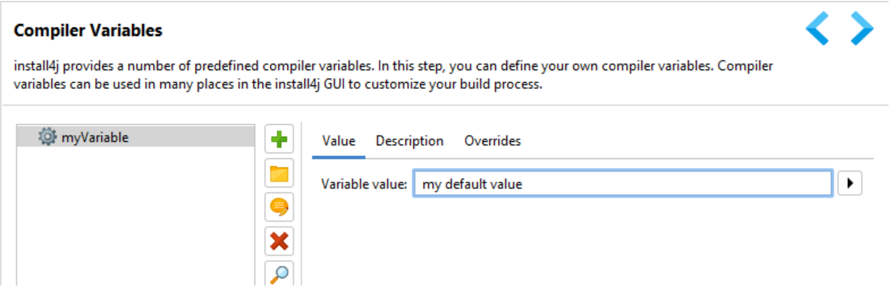
For both compiler and installer variables install4j offers a fixed set of "system variables" that are prefixed with "sys.". These variables are not writable and it is discouraged to use this prefix for your own variables.

Compiler variables

Compiler variables are written as

```
${compiler:variableName}
```

The value of a compiler variable is a string that is known and replaced at compile time. The installer runtime or the generated launchers do not see this variable, but just the value that was substituted at runtime. Compiler variables are defined on the "General Settings->Compiler Variables" step.



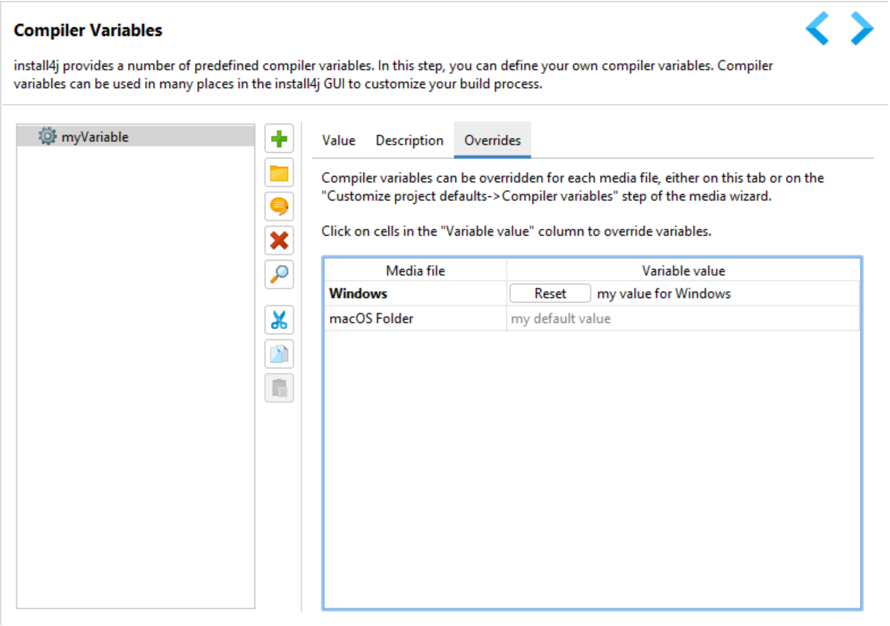
Compiler Variables

install4j provides a number of predefined compiler variables. In this step, you can define your own compiler variables. Compiler variables can be used in many places in the install4j GUI to customize your build process.

Variable value: my default value

You can use compiler variables for various purposes. The most common usage of a compiler variable is the possibility to define a string in one place and use it in many other places. You can then change the string in one place instead of having to look up all of its usages.

An example of this use case is the pre-defined `sys.version` variable that contains the value of the text field where you enter the application version. Another usage for compiler variables is to override certain project settings on a per-media file basis. For example, if you want to include one directory in the distribution tree for Windows but another one for macOS, you can use a compiler variable for that directory and override it for each media file.



Compiler Variables

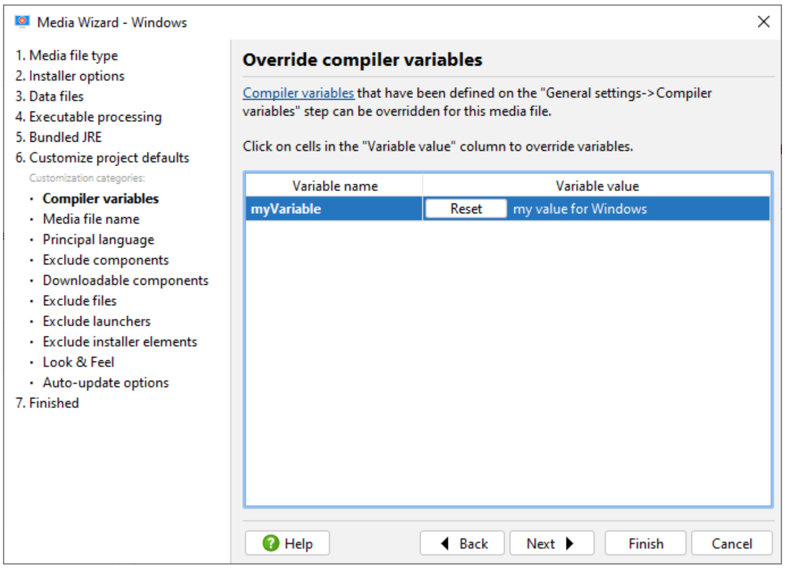
install4j provides a number of predefined compiler variables. In this step, you can define your own compiler variables. Compiler variables can be used in many places in the install4j GUI to customize your build process.

Compiler variables can be overridden for each media file, either on this tab or on the "Customize project defaults->Compiler variables" step of the media wizard.

Click on cells in the "Variable value" column to override variables.

Media file	Variable value
Windows	Reset my value for Windows
macOS Folder	my default value

To quickly override multiple variables for a single media file, you can configure overridden values on the "Customize project defaults->Compiler variables" step of the media wizard.



Override compiler variables

Compiler variables that have been defined on the "General settings->Compiler variables" step can be overridden for this media file.

Click on cells in the "Variable value" column to override variables.

Variable name	Variable value
myVariable	Reset my value for Windows

Finally, compiler variables can be overridden from the command line compiler [p. 220] as well as from the Gradle [p. 225], Maven [p. 230] and Ant [p. 239] plugins.

When you use a compiler variable in your project that is not a system variable, it must be defined in on the "General Settings->Compiler Variables" step. If an unknown variable is encountered, the build will fail. You can use other variables in the value of a variable. Recursive definitions are detected and lead to a failure of the build. It is not possible to define compiler variables with the name of a system variable.

install4j provides a number of system compiler variables:

- **sys.date [Machine-specific variables]**

  The current date in the format `YYYYMMDD` (e.g. "20090910"). The value is set at the start of a build and will not change during a single build.

- **sys.time [Machine-specific variables]**

  The current time in the format `HHMMSS` (e.g. "153012") where HH is the hour in 24-hour format, MM is the minute and SS is the second. The value is set at the start of a build and will not change during a single build.

- **sys.timestamp [Machine-specific variables]**

  The current time as the Unix epoch. This is a long value with the milliseconds since January 1st, 1970 (UTC). The value is set at the start of a build and will not change during a single build.

- **sys.install4jHome [Machine-specific variables]**

  The installation directory of install4j that is used for compiling the media files.

- **sys.install4jVersion [Machine-specific variables]**

  The version of install4j that is used for compiling the media files.

- **sys.fileSeparator [Machine-specific variables]**

  The platform-dependent separator for directories in a file path. On Windows, this is a backslash ("\"), on Unix a forward slash ("/"). The value of this variable is intended to refer to files on the build machine. For a value that is valid at runtime, use `sys.mediaFileSeparator` instead.

- **sys.pathlistSeparator [Machine-specific variables]**

  The platform-dependent separator for lists of directories. On Windows, this is a semicolon (";"), on Unix a colon (":"). The value of this variable is intended to refer to files on the build machine. For a value that is valid at runtime, use `sys.mediaPathlistSeparator` instead.

- **sys.version [Project-specific variables]**

  The version of your application as configured under General Settings->Application Info.

- **sys.shortName [Project-specific variables]**

  The short name of your application as configured under General Settings->Application Info.

- **sys.fullName [Project-specific variables]**

  The full name of your application as configured under General Settings->Application Info.

- **sys.publisher [Project-specific variables]**

  The publisher of your application as configured under General Settings->Application Info.

- **sys.publisherUrl [Project-specific variables]**

  The publisher URL of your application as configured under General Settings->Application Info.

- **sys.languageId [Project-specific variables]**

  The 2-letter ISO 639 code (see https://www.loc.gov/standards/iso639-2/php/code_list.php (1)) for the principal language of the installer. This variable can be overridden on the command line or the ant task which is useful if you build different installers for different languages.

- **sys.javaMinVersion [Project-specific variables]**

  The minimum Java version as configured under General Settings->Java Version

---

(1) https://www.loc.gov/standards/iso639-2/php/code_list.php

- **sys.javaMaxVersion [Project-specific variables]**

  The maximum Java version as configured under General Settings->Java Version

- **sys.applicationId [Project-specific variables]**

  The application ID as configured under Installer->Update Options

- **sys.updatesUrl [Project-specific variables]**

  The URL where auto updaters can download the update descriptor file `updates.xml` as configured under Installer->Auto-Update Options. This variable is usually used in the "Update descriptor URL" property of a "Check for update" action.

- **sys.mediaFileName [Media-specific variables]**

  The file name of the currently compiled media file as configured in the Media section and possibly overridden in "Customize project defaults->Media file name" step of the media wizard.

- **sys.mediaName [Media-specific variables]**

  The display name in the install4j IDE of the currently compiled media file as configured in the Media section. If the default name of the media file is not suitable, you can rename the media file.

- **sys.mediaId [Media-specific variables]**

  The ID of the currently compiled media file as configured in the Media section. This corresponds to the return value of `context.getMediaFileId()`.

- **sys.platform [Media-specific variables]**

  The platform descriptor of the currently compiled media file. One of `windows-x64`, `windows-x32`, `linux`, `unix` or `macos`. The value of this variable depends on your choice in the platform step of the media file wizard.

- **sys.withJre [Media-specific variables]**

  A variable that contains "_with_jre" if a JRE is statically bundled with a media file and the empty string if not. This is useful if media files with and without JRE are built.

- **sys.jreBundleVersion [Media-specific variables]**

  The Java version of the JRE bundle if a JRE bundle is configured for a media file and the empty string if not.

- **sys.jreBundleArch [Media-specific variables]**

  The architecture of the JRE bundle if a JRE bundle is configured for a media file and the empty string if not.

- **sys.mediaFileSeparator [Media-specific variables]**

  The platform-dependent separator for directories in a file path based on the current media set. For Windows media sets, this is a backslash ("\"), for all others a forward slash ("/").

- **sys.mediaPathlistSeparator [Media-specific variables]**

  The platform-dependent separator for lists of directories based on the current media set. For Windows media sets, this is a semicolon (";"), for all others a colon (":").

- **sys.msiProductId [Media-specific variables]**

  The product GUID if a Windows installer is wrapped in an MSI package, otherwise an empty string.

You can access environment variables on the build machine with the syntax

```
${compiler:env.environmentVariableName}
```

where "environmentVariableName" is the name of an environment variable. This is resolved at build time and only works if no compiler variable with the same name is defined on the "General Settings->Compiler Variables" step.

Compiler variable values in the IDE cannot be multi-line strings. If you need to insert a variable with a multi-line string, you can use the text file reference syntax

```
${compiler:file("path/to/file")}
```

where `path/to/file` is either an absolute file path or a path relative to the config file. All text areas that have an adjacent variable selector button offer the "Insert contents of text file" action in its popup menu. The file chooser has an option whether to use a relative or an absolute path in the variable expression.

In order to debug problems with compiler variables, you can switch on the `extra verbose output` flag in the Build step [p. 11]. All variable replacements will then be printed to the build console.

The file path can be a variable expression itself, like in

```
${compiler:file(${compiler:myFile})}
```

so you can override it for each media file or pass it as a parameter to a command line build.

Installer variables

Installer variables are written as

```
${installer:variableName}
```

The value of an installer variable is an arbitrary object that is not known at compile time. Installer variables are replaced at runtime in the installer, the uninstaller and in custom installer applications. They can optionally be predefined in the install4j IDE like compiler variables, but this is not required.

Undefined installer variables come into existence the first time they are defined at runtime. However, it is an error to use an undefined variable. For example, if you use an installer variable in an action, you have to make sure that the installer variable is defined before the action is executed.

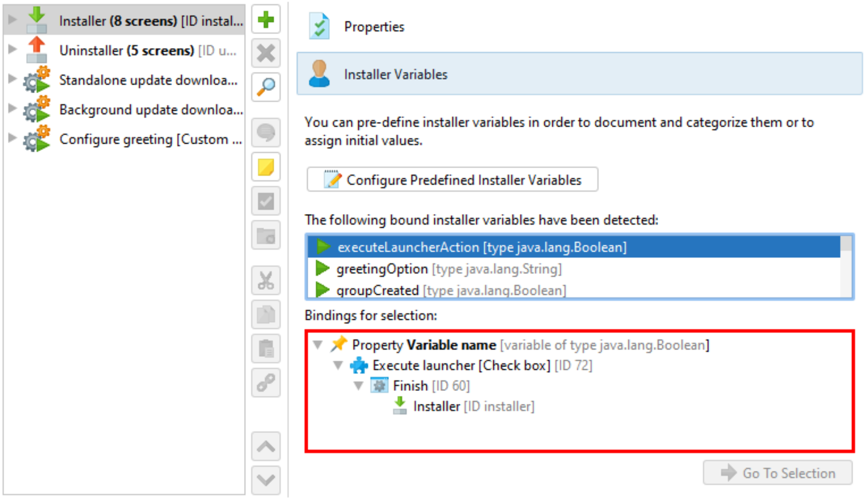
Installer variables are used to wire together actions, screens and form components at runtime. The user input in screens is saved to variables that can be used in the properties of actions. Furthermore, installer variables can be used in condition and validation expressions. Some examples are given in the help topic on form screens [p. 46]. In script properties, you retrieve variables by invoking

```
context.getVariable("variableName")
```

Variable values can be set with the installer API by invoking

```
context.setVariable("variableName", variableValue)
```

You can analyze the bindings of an installer variable on the "Installer Variables" tab of an installer application configuration. That tab will show you a list of bound variables together with all bindings.


Installer **(8 screens)** [ID instal...
Uninstaller **(5 screens)** [ID u...
Standalone update downloa...
Background update downloa...
Configure greeting [Custom ...
Properties
Installer Variables
Configure Predefined Installer Variables
Go To Selection

You can pre-define installer variables in order to document and categorize them or to assign initial values.

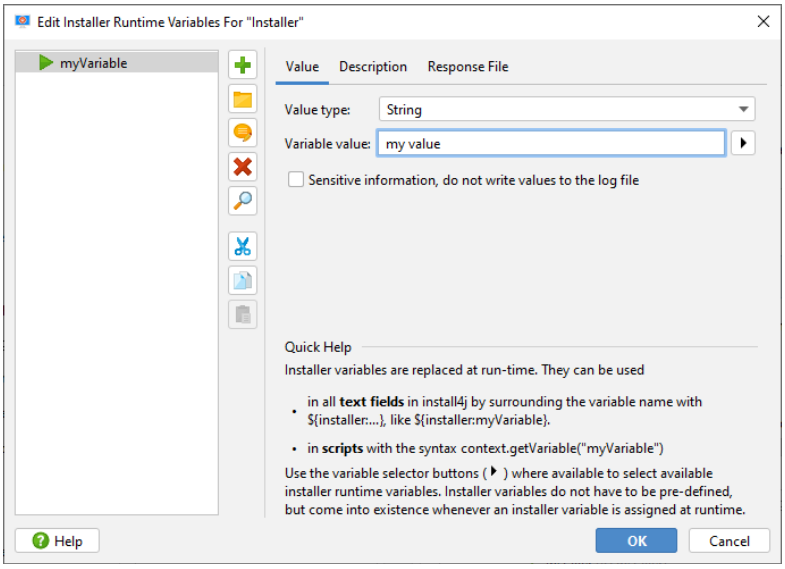
The following bound installer variables have been detected:

- executeLauncherAction [type java.lang.Boolean]
- greetingOption [type java.lang.String]
- groupCreated [type java.lang.Boolean]

Bindings for selection:

- Property **Variable name** [variable of type java.lang.Boolean]
  - Execute launcher [Check box] [ID 72]
    - Finish [ID 60]
      - Installer [ID installer]

In order to document and categorize bound installer variables, you can pre-define them and set descriptions that will be displayed in the installer variable selector in the install4j IDE.



Value type: String

Variable value: my value

☐ Sensitive information, do not write values to the log file

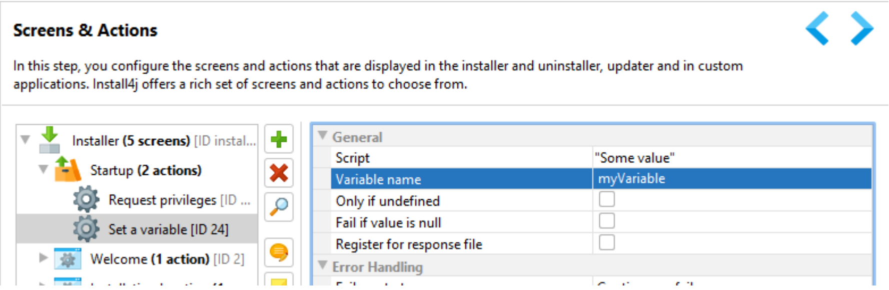
Quick Help

Installer variables are replaced at run-time. They can be used

- in all **text fields** in install4j by surrounding the variable name with ${installer:...}, like ${installer:myVariable}.
- in **scripts** with the syntax context.getVariable("myVariable")

Use the variable selector buttons (▸) where available to select available installer runtime variables. Installer variables do not have to be pre-defined, but come into existence whenever an installer variable is assigned at runtime.

A common scenario is the need to calculate a variable value at runtime with some custom code and use the result as the initial value of a form component. To achieve this, you can add a "Set a variable" action to the startup screen and set its "Variable name" property to some variable name. In this context, install4j expects a variable name and you must not use the `${installer:variableName}` syntax but specify the plain `variableName` only. The return value of the "Script" property is written to the variable.


Screens & Actions

In this step, you configure the screens and actions that are displayed in the installer and uninstaller, updater and in custom applications. Install4j offers a rich set of screens and actions to choose from.

Installer (5 screens) [ID instal...
Startup (2 actions)
Request privileges [ID ...
Set a variable [ID 24]
Welcome (1 action) [ID 2]
General
Error Handling

Script: "Some value"
Variable name: myVariable
Only if undefined: ☐
Fail if value is null: ☐
Register for response file: ☐

For example, if this variable represents the initial directory that is displayed for a "Directory chooser" form component, you set the "Initial Directory" property of that form component to `${installer:variableName}`. In this way you have wired the results of an action with a behavior of a screen.

Another important use of installer variables is in the names of custom installation roots [p. 14]. In most cases, the name of a custom installation root contains an installer variable that is resolved at runtime. Often, one of the system installer variables that represent a "magic" folder can be used, such as `${installer:sys.system32Dir}` for the Windows `system32` directory.

When you use installer variables in properties that display text, such as the screen title or the label properties of form components, a live binding will be created and the displayed text is updated automatically when the variable values change.

Installer variables can be passed to the installer, uninstaller or custom installer applications from the command line prefixed with `-V`:

```
-VmyVar=test "-VmyVarWithSpaces=this is a test"
```

Alternatively, you can specify a property file containing installer variables with `-varfile my.properties`, where the file `my.properties` contains one variable definition per line. The variables that are created will be instances of `java.lang.String`.

install4j provides a number of system installer variables:

- **sys.installationDir [Source and Target]**

  The installation directory for the current installation. The value of this variable can change in the installer as the user selects an installation directory in the "Installation directory" screen or the installation directory is set via `context.setInstallationDirectory(File installationDirectory)`.

  Note that for single bundle installers on macOS, the installation directory is usually just `/Applications`, not a separate subdirectory.

- **sys.contentDir [Source and Target]**

  The directory that holds the installed files. On Windows, Linux and Unix, this is the same as the installation directory. For single bundle installers on macOS, this is `[Bundle name].app/Contents/Resources/app/`. To reference an installed file in a cross-platform way, use this variable and not sys.installationDir.

- **sys.mediaFile [Source and Target]**

  The path of your media file. Not available for uninstallers.

On Unix and for non-MSI Windows installers this is the same as sys.installerFile. For MSI installers, this is the MSI file. On macOS, this is the path to the DMG file. If you want to reference the installer file, use sys.installerFile instead.

- **sys.mediaDir [Source and Target]**

  The path of the directory where your installer file is located. Not available for uninstallers.

  On Unix and for non-MSI Windows installers this is the same as sys.installerDir. For MSI installers, this is the directory where the MSI file is located. On macOS, this is the directory where the DMG file is located. If you want to reference files inside the DMG file, use sys.installerDir instead.

- **sys.installerFile [Source and Target]**

  The path of your installer file. Not available for uninstallers.

  On Unix and for non-MSI Windows installers this is the same as sys.mediaFile. For MSI installers, this is the extracted installer executable. On macOS, this is the path to the installer inside the mounted DMG. If you want to reference the DMG file, use sys.mediaFile instead.

- **sys.installerDir [Source and Target]**

  The path of the directory where your installer file is located. Not available for uninstallers.

  On Unix and for non-MSI Windows installers this is the same as sys.mediaDir. For MSI installers, this is the directory the installer was extracted to. On macOS, this is the path into the mounted DMG. If you want to reference files in the same directory as the DMG file, use sys.mediaDir instead.

- **sys.resourceDir [Installer application state]**

  The directory where the resource files are present that have been configured on the Installer->Custom Code & Resources tab.

- **sys.installationTypeId [Installer application state]**

  The ID of the selected installation type. This is only relevant if the "Installation Type" screen has been added to the installer. The value is `null` as long as no installation type has been selected.

- **sys.version [Installer application state]**

  For installers, the version of your application as configured under General Settings->Application Info. In that case, the variable yields the same value as the compiler variable of the same name. For custom installer applications, the installed version,which might not be the same as the version for which the custom installer application was originally compiled.

- **sys.logFile [Installer application state]**

  The full path to the currently used log file. This is a path in the `TEMP` directory. For installers, this changes after the "Install Files" action, when the log file is moved to a path in the installation directory.

- **sys.responseFile [Installer application state]**

  If a response file is supplied with a `-varfile` command line argument, the full path to the response file. If no response file is used, the variable value is `null`.

- **sys.preferredJre [Installer application state]**

  The home directory of the JRE that will be used by the installed launchers. This variable will only be set after the "Install files" action has run. It will be the same as `System.getProperty("java.home")` or the `sys.javaHome` installer variable unless a bundled JRE (shared or non-shared) has been installed. This variable is not available in the uninstaller or custom installer applications, use the `sys.javaHome` directory there.

- **sys.languageId [Installer application state]**

  The 2-letter ISO 639 code (see https://www.loc.gov/standards/iso639-2/php/code_list.php (1)) for the actual language of the installer. For fixed-language installers, this is the same as the compiler variable of the same name. For multi-language installers, the value is determined at runtime.

- **sys.installerApplicationMode [Installer application state]**

  A string that reports the type of the installer application: "installer" for the installer, "uninstaller" for the uninstaller and "custom" for custom installer applications.

- **sys.programGroupDisabled [Installer application state/Program group]**

  If the user has disabled program group creation on the "Standard program group" screen. This applies to both the Windows program group and the Linux/Unix launcher link directory selection. If no "Standard program group" screen is present, the variable value will be `null`.

- **sys.programGroupName [Installer application state/Program group]**

  The name of the program group that user has selected on the "Standard program group" screen. If no program group has been selected, the variable value will be `null`. Only set in Windows installers.

- **sys.programGroupDir [Installer application state/Program group]**

  The directory that has been selected as the program group. This is the full path to the actual location of the program group, not just the name of the program group. If no program group has been selected, the variable value will be `null`. Only set in Windows installers.

- **sys.programGroupAllUsers [Installer application state/Program group]**

  If the user has selected to create menu entries for all users on the "Standard program group" screen. If no "Standard program group" screen is present, the variable value will be `null`. Only set in Windows installers.

- **sys.symlinkDir [Installer application state/Program group]**

  The name of the directory for launcher links that user has selected on the "Standard program group" screen. If no program group has been selected, the variable value will be `null`. Only set in Linux/Unix installers.

- **sys.fileSeparator [Cross-platform variables]**

  The platform-dependent separator for directories in a file path. On Windows, this is a backslash ("\"), on Unix a forward slash ("/").

- **sys.pathlistSeparator [Cross-platform variables]**

  The platform-dependent separator for lists of directories. On Windows, this is a semicolon (";"), on Unix a colon (":").

- **sys.userHome [Cross-platform variables]**

  The user home directory, typically something like `C:\Users\$USER` on Windows or `/home/$USER` on Unix platforms.

- **sys.userName [Cross-platform variables]**

  The user account name.

---

(1) https://www.loc.gov/standards/iso639-2/php/code_list.php

- **sys.workingDir [Cross-platform variables]**

  The working directory. For the installer, this is the temporary directory that the installer was extracted to.

- **sys.tempDir [Cross-platform variables]**

  The temporary directory of the operating system. On all supported platforms, this is the value of the `TEMP` environment variable.

- **sys.javaHome [Cross-platform variables]**

  The Java home directory of the currently used JRE.

- **sys.javaVersion [Cross-platform variables]**

  The Java version of the currently used JRE.

- **sys.confirmedUpdateInstallation [Cross-platform variables]**

  If the user has confirmed an update installation on top of a previous installation. If a previous installation is detected, the "Welcome" screen asks the user whether to perform an update installation or choose another installation directory. The result of that question is saved to this variable. If the "Welcome screen is not shown, this variable is not set and `Context#getBooleanVariable(...)` returns false for this variable.

- **sys.desktopDir [Cross-platform variables]**

  The directory used to physically store file objects on the desktop. On Windows, a typical path is `C:\Users\[user name]\Desktop`. On macOS, this is the `~/Desktop` directory and on Unix the freedesktop.org setting for the `XDG_DESKTOP_DIR` directory is returned.

- **sys.docsDir [Cross-platform variables]**

  The directory used to physically store a user's common repository of documents. On Windows, a typical path is `C:\Users\[user name]\Documents`. On macOS, this is the `~/Documents` directory and on Unix the freedesktop.org setting for the `XDG_DOCUMENTS_DIR` directory is returned.

- **sys.downloadsDir [Cross-platform variables]**

  The directory used to physically store a user's downloads. On Windows, a typical path is `C:\Users\[user name]\Downloads`. On macOS, this is the `~/Downloads` directory and on Unix the freedesktop.org setting for the `XDG_DOWNLOAD_DIR` directory is returned.

- **sys.appdataDir [Platform-specific variables]**

  The directory that serves as a common repository for application-specific data. On Windows, a typical path is `C:\Users\[user name]\AppData\Roaming`. On macOS, this is the `~/Library/Application Support` directory. On Unix, the value of the `XDG_DATA_HOME` environment variable or if not defined `~/.local/share` is returned.

- **sys.localAppdataDir [Platform-specific variables]**

  The user-specific directory that serves local applications to store computed data. On Windows, a typical path is `C:\Users\[user name]\AppData\Local`. On macOS, this is the `~/Library/Caches` directory. On Unix, the value of the `XDG_CACHE_HOME` environment variable or if not defined `~/.cache` is returned.

- **sys.windowsDir [Platform-specific variables]**

  The Windows installation directory, typically `C:\Windows`.

- **sys.system32Dir [Platform-specific variables]**

  The system32 directory of your Windows installation, typically `C:\Windows\system32`.

- **sys.commonDir [Platform-specific variables]**

  The common files directory of your Windows installation, typically `C:\Program Files\Common Files`.

- **sys.programDataDir [Platform-specific variables]**

  The directory where applications can save data that is not specific to particular users. A typical path is `C:\ProgramData`.

- **sys.startMenuDir [Platform-specific variables]**

  The directory containing Start menu items. A typical path is `C:\Users\[user name]\AppData\Roaming\Microsoft\Windows\Start Menu`.

- **sys.programsDir [Platform-specific variables]**

  The directory that contains the user's program groups. The groups are themselves file system directories. A typical path is `C:\Users\[user name]\AppData\Roaming\Microsoft\Windows\Start Menu\Programs`.

- **sys.startupDir [Platform-specific variables]**

  The directory that corresponds to the user's Startup program group. The system starts these programs whenever any user logs onto Windows. A typical path is `C:\Users\[user name]\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup`.

- **sys.sendToDir [Platform-specific variables]**

  The directory that contains Send To menu items. A typical path is `C:\Users\[user name]\AppData\Roaming\Microsoft\Windows\SendTo`.

- **sys.templatesDir [Platform-specific variables]**

  The directory that serves as a common repository for document templates. A typical path is `C:\Users\[user name]\AppData\Roaming\Microsoft\Windows\Templates`.

- **sys.favoritesDir [Platform-specific variables]**

  The directory that serves as a common repository for the user's favorite items. A typical path is `C:\Users\[user name]\Favorites`.

- **sys.programGroupDir [Platform-specific variables]**

  The directory of the program group that will be or was created by the "Create standard program group" action. If this action is not present, the value will be `null`. The value of this variable can change in the installer as the user selects a program group on the "Create program group" screen.

- **sys.fontsDir [Platform-specific variables]**

  The folder that contains fonts. A typical path is `C:\Windows\Fonts`. On macOS, the value is `/Library/Fonts`.

- **sys.programFilesDir [Platform-specific variables]**

  The directory where programs are installed, typically something like `C:\Program Files`. On macOS, the value is `/Applications`.

- **sys.date [Cross-platform variables]**

  The current date in the format `YYYYMMDD` (e.g. "20090910"). The value is set when the installer is started and will not change for the current process.

- **sys.time [Cross-platform variables]**

  The current time in the format `HHMMSS` (e.g. "153012") where HH is the hour in 24-hour format, MM is the minute and SS is the second. The value is set when the installer is started and will not change for the current process.

- **sys.timestamp [Cross-platform variables]**

  The current time as the Unix epoch. This is a long value with the milliseconds since January 1st, 1970 (UTC). The value is set when the installer is started and will not change for the current process.

Launcher variables

Launcher variables are written as

```
${launcher:variableName}
```

The value of a launcher variable is a string that is not known at compile time. In contrast to installer variables, they are replaced by the launcher and not by Java code, so the replaced value is seen by the JVM at startup. Launcher variables can only be used in the "VM parameters" and "Arguments" text fields on the "Java invocation" step of the launcher wizard [p. 36].

No user-defined launcher variables exist, the available system launcher variables are:

- **sys.launcherDirectory**

  The directory in which your launcher has been installed at runtime.

- **sys.jvmHome**

  The home directory of the JVM that your launcher is running with. This is useful to put JAR files from the JRE into your boot classpath. The "home directory" is the directory that contains the "bin" directory of the JRE.

- **sys.tempDir**

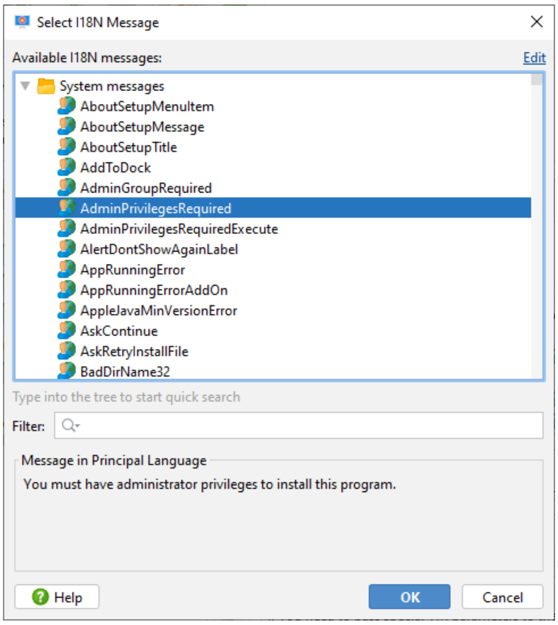
  The temporary directory for the current user.

I18N messages

I18N messages are written as

```
${i18n:keyName}
```

The value of an I18N message depends on the language that is selected for the installer. You can use this facility to localize messages in your installers if they support multiple languages [p. 79]. To do that, you supply key-value pairs in the custom localization file. The variable selection dialog for I18N messages shows all system messages as well as all messages in the custom localization file for the principal language of your project.



You must have administrator privileges to install this program.

All standard messages displayed by install4j can be referenced with this syntax as well. You can locate the key name in one of the `message_*.utf8` files in the `$INSTALL4J_HOME/resource/messages` directory and use it anywhere in your project. The standard messages can be overwritten by your custom localization files.

Default values for missing variables

For the text field syntax of installer and compiler variables there is a mechanism to supply a default value in case the variable is not defined: After the variable name you add the delimiter `?:` and insert the default value before the closing curly bracket.

For example:

```
${installer:myVariable?:defaultValue}
```

will resolve to `defaultValue` if the installer variable "myVariable" is not defined. The default value can be another variable, also of a different type. For example:

```
${installer:updatesUrl?:${compiler:sys.updatesUrl}}
```

If the installer variable "updatesUrl" is not defined, the compiler variable "sys.updatesUrl" is inserted. This is the default value of the "Update descriptor URL" property of the "Check for update" action.

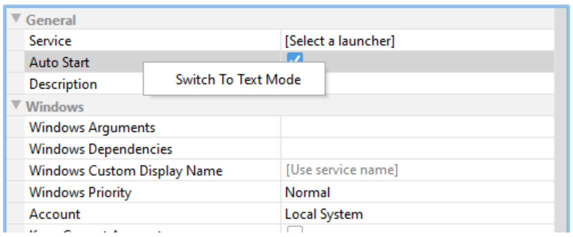
The chain of default values can be arbitrarily long:

```
${installer:one?:${installer:two?:${installer:three?:${installer:four?:some plain
text}}}}
```

This will resolve to the first defined installer variable out of "one", "two", "three", "four" or to `some plain text` if none of them are defined.

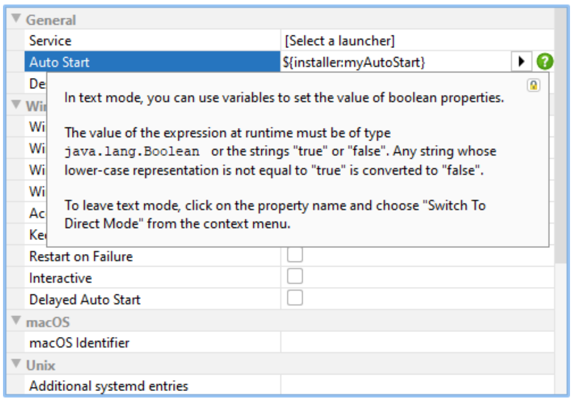
Binding variables to non-text properties

Many bean properties do not take text input, for example boolean, integer or enum properties, so that the variable syntax `${installer:myVariable}` for text fields is not applicable. For these properties, you can select "Switch to text mode" in the context menu and enter a variable expression that resolves to the required type. Conversions from string values are important because compiler variables can only hold string values, unlike installer variables that can hold arbitrary types.



General	
Service	[Select a launcher]
Auto Start	☑
Description	
Windows	
Windows Arguments	
Windows Dependencies	
Windows Custom Display Name	[Use service name]
Windows Priority	Normal
Account	Local System

The help icon in the property editor tells you what the property type is and also informs about the supported conversions from other primitive types or strings. For example, "true" or "false" string values are supported for boolean properties as well, which is what you would use with a compiler variable. For enum properties, the name of the enum or the ordinal as a number or as a string will be resolved to the actual enum value. Also, numeric values will be parsed from strings.



General	
Service	[Select a launcher]
Auto Start	${installer:myAutoStart}
Restart on Failure	☐
Interactive	☐
Delayed Auto Start	☐
macOS	
macOS Identifier	
Unix	
Additional systemd entries	

In text mode, you can use variables to set the value of boolean properties.

The value of the expression at runtime must be of type `java.lang.Boolean` or the strings "true" or "false". Any string whose lower-case representation is not equal to "true" is converted to "false".

To leave text mode, click on the property name and choose "Switch To Direct Mode" from the context menu.

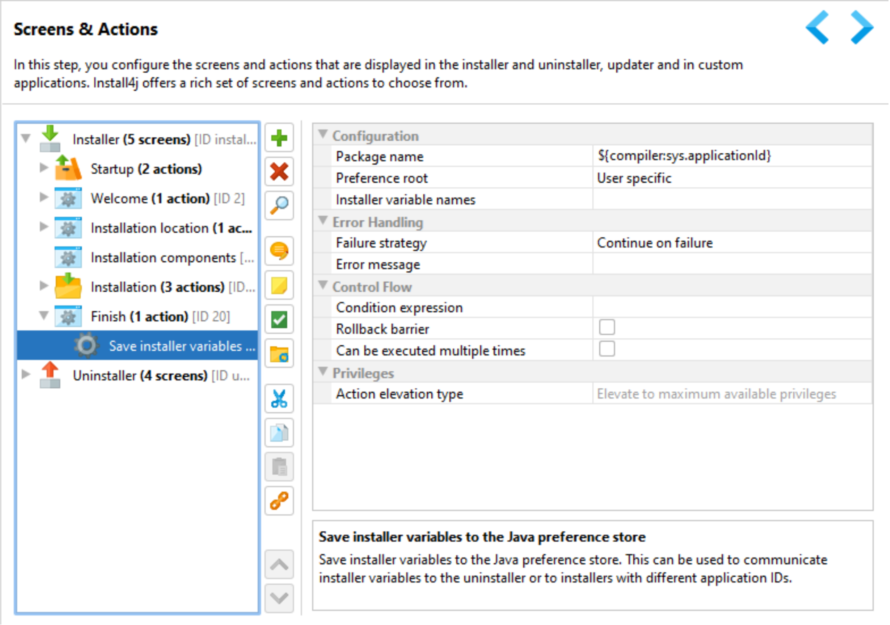
If you develop a custom bean and want to support that functionality as well, you have to enable it in the property descriptor and insert a call into the property getter as explained in the Javadoc for AbstractBean.

Using variables in your own applications

Frequently there is a need in the installed applications to access user input that was made in the installer. The launcher API [p. 216] provides the helper class `com.install4j.api.launcher.Variables` to access the values of installer variables.

There are two ways that installer variables can be persisted in the installer: First, installer variables are saved to the default response file [p. 202] `.install4j/response.varfile` that is created when the installer exits or if a "Create response file" action is executed. Only response file variables are saved to that file. Secondly, selected installer variables can be saved to the Java preference store. `com.install4j.api.launcher.Variables` offers methods to load variables from both sources.

Saving to the Java preference store is interesting if you want to modify those variable values in your applications and save back the modified values. The Java preference store is available on a per-user basis so that it is possible to modify settings even if the user does not have write permissions for the installation directory. `com.install4j.api.launcher.Variables` has methods for loading and saving the entire map of installer variables that was saved in the installer. Also, it is possible to specify an arbitrary package to which the installer variables are saved, so that settings can be shared between different installers.



Screens & Actions

In this step, you configure the screens and actions that are displayed in the installer and uninstaller, updater and in custom applications. Install4j offers a rich set of screens and actions to choose from.

Installer (5 screens) [ID instal...
Startup (2 actions)
Welcome (1 action) [ID 2]
Installation location (1 ac...
Installation components [...
Installation (3 actions) [ID...
Finish (1 action) [ID 20]
Save installer variables ...
Uninstaller (4 screens) [ID u...

Configuration	
Package name	${compiler:sys.applicationId}
Preference root	User specific
Installer variable names	
Error Handling	
Failure strategy	Continue on failure
Error message	
Control Flow	
Condition expression	
Rollback barrier	☐
Can be executed multiple times	☐
Privileges	
Action elevation type	Elevate to maximum available privileges

Save installer variables to the Java preference store

Save installer variables to the Java preference store. This can be used to communicate installer variables to the uninstaller or to installers with different application IDs.

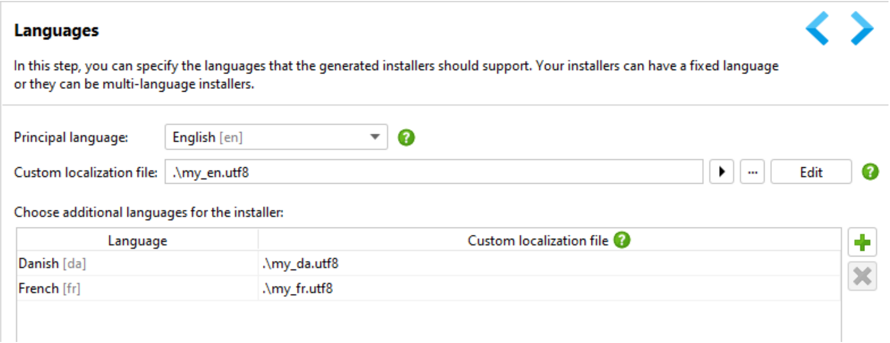
Finally, it is useful to access compiler variables in your own applications. For example, the version number configured in the install4j IDE can be accessed in your own application through `com.install4j.api.launcher.Variables`.

A.13 Localization

On the "General Settings->Languages" step, you configure the languages that are supported by your project. The following languages are available:

- Arabic [ar]
- Chinese (Simplified) [zh_CN]
- Chinese (Traditional) [zh_TW]
- Croatian [hr]
- Czech [cs]
- Danish [da]
- Dutch [nl]
- English [en]
- Finnish [fi]
- French [fr]
- German [de]
- Greek [el]
- Hebrew [he]
- Hungarian [hu]
- Italian [it]
- Japanese [ja]
- Korean [ko]
- Norwegian [no]
- Polish [pl]
- Portuguese [pt]
- Portuguese (Brazilian) [pt_BR]
- Romanian [ro]
- Russian [ru]
- Spanish [es]
- Swedish [sv]
- Turkish [tr]

By default, only one language is shipped with the installer. This is called the **principal language**. By adding additional languages, you can build multi-language installers. If none of the configured languages match the locale at runtime, the principal language is used.



Languages

In this step, you can specify the languages that the generated installers should support. Your installers can have a fixed language or they can be multi-language installers.

Principal language: English [en]

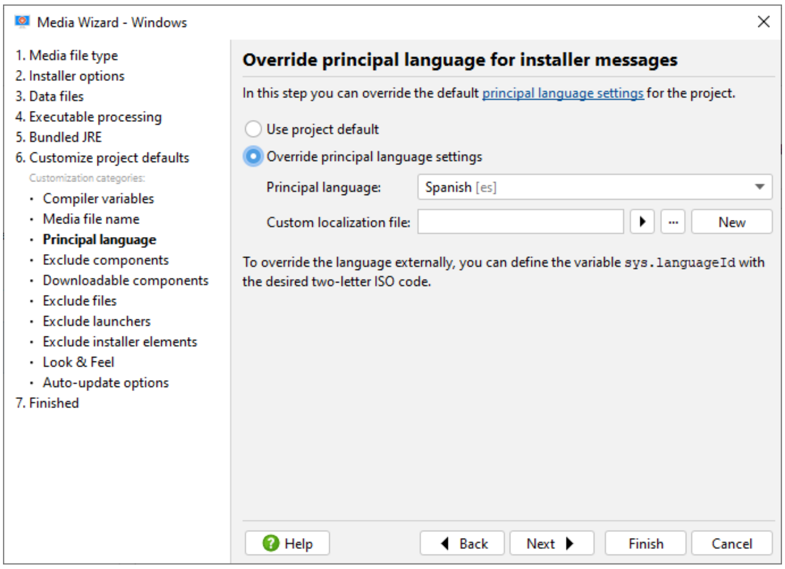
Custom localization file: .\my_en.utf8

Choose additional languages for the installer:

Language	Custom localization file
Danish [da]	.\my_da.utf8
French [fr]	.\my_fr.utf8

For multi-language installers, a language selection dialog is shown when the installer is started. By selecting the `Skip language selection dialog` check box you can choose to show the language selection only if the installer cannot find a match between a supported language and the auto-detected locale.

The principal language setting can be overridden for each media file on the "Customize project defaults->Principal language" step of the media wizard. In this way, you can build multiple fixed-language installers, each with a different principal language.



Override principal language for installer messages

In this step you can override the default principal language settings for the project.

○ Use project default

● Override principal language settings

Principal language: Spanish [es]

Custom localization file:

To override the language externally, you can define the variable `sys.languageId` with the desired two-letter ISO code.

Localization mechanism

In projects, localized messages are obtained in one of two ways;

- **with i18n messages**

  The i18n variable system [p. 63] gives access to all messages with the syntax

  ```
  ${i18n:messageKey}
  ```

  To select a message, use the ▶ variable selector button next to text fields and properties. For messages with one or more parameters of the form `{0}` to `{n}`, the variable selector will insert placeholder values like in

```
${i18n:DiskSpaceWarning("arg 0", "arg 1")}
```

- **with the API**

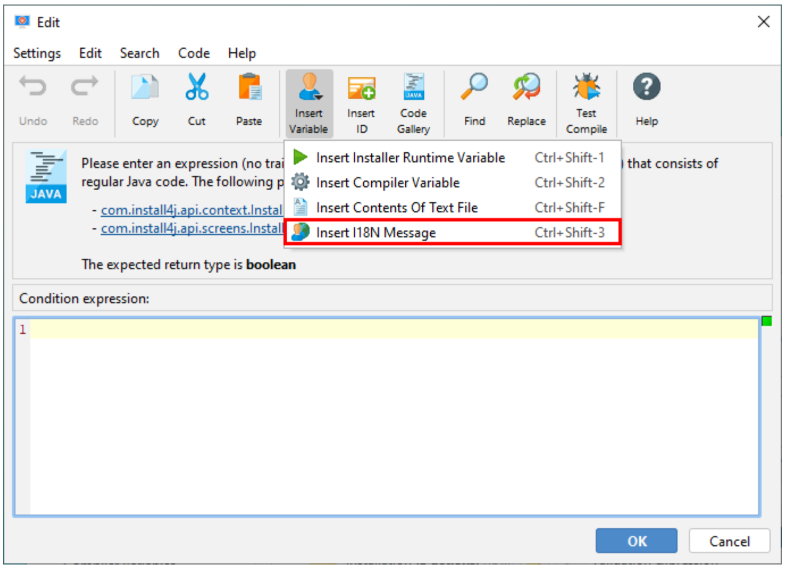
  In scripts and in your custom code you can call

  ```
  context.getMessage("messageKey")
  ```

  For messages with arguments, you pass the arguments with the vararg syntax:

  ```
  context.getMessage("DiskSpaceWarning", 10000, 100)
  ```

  The "Insert variable" tool bar button in script editors allows you to insert these calls with the correct syntax for selected message keys.



Please enter an expression (no trai … that consists of
regular Java code. The following p
- com.install4j.api.context.Instal
- com.install4j.api.screens.Install

The expected return type is **boolean**

Condition expression:

Edit
Settings
Edit
Search
Code
Help
Undo
Redo
Copy
Cut
Paste
Insert Variable
Insert ID
Code Gallery
Find
Replace
Test Compile
Help
JAVA
Insert Installer Runtime Variable Ctrl+Shift-1
Insert Compiler Variable Ctrl+Shift-2
Insert Contents Of Text File Ctrl+Shift-F
Insert I18N Message Ctrl+Shift-3
OK
Cancel

Custom localization

In addition to the standard messages that are displayed in the generated installer and uninstaller, you will have your own messages that need to be localized in the same way. To configure these messages, create a custom localization file for the principal language. A custom localization file is a text file with key-message pairs in the format of

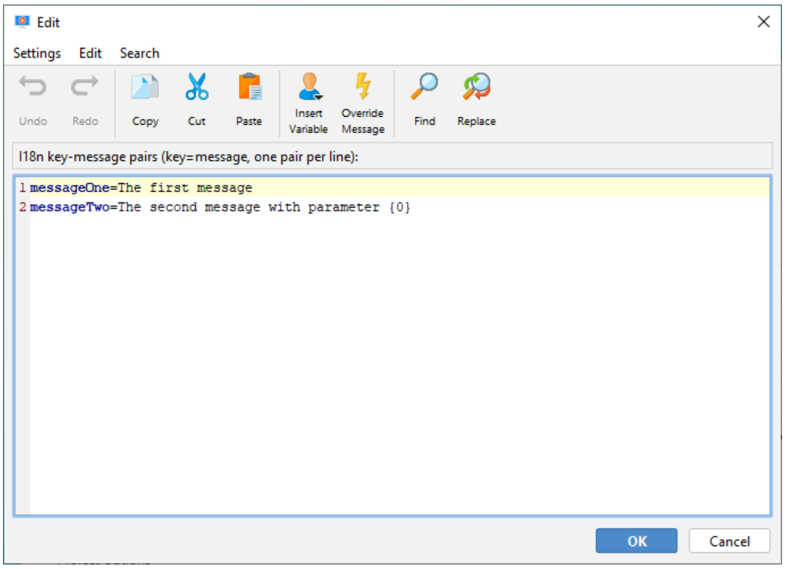
- **a Java properties file**

  A Java properties file has a `.properties` file extension and must use ISO 8859-1 encoding. All other characters must be represented as Unicode escape sequences, like `\u0823`.

- **a properties file with UTF-8 encoding**

  A properties file with UTF-8 encoding has an `.utf8` file extension and has the advantage that you do not have to use escape sequences. However, it might not be supported by some localization tools.

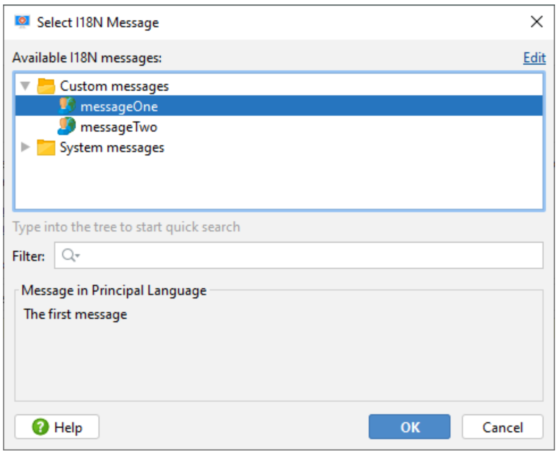
You can create and edit custom localization files externally or directly in the install4j IDE with the built-in editor:



I18n key-message pairs (key=message, one pair per line):

```
messageOne=The first message
messageTwo=The second message with parameter {0}
```

For each additional language, add a corresponding custom localization file that contains the same keys. If a message is missing for an additional language, the message for the principal language is used. The variable selection dialog for i18n messages will show all keys in the custom localization file for the principal language.



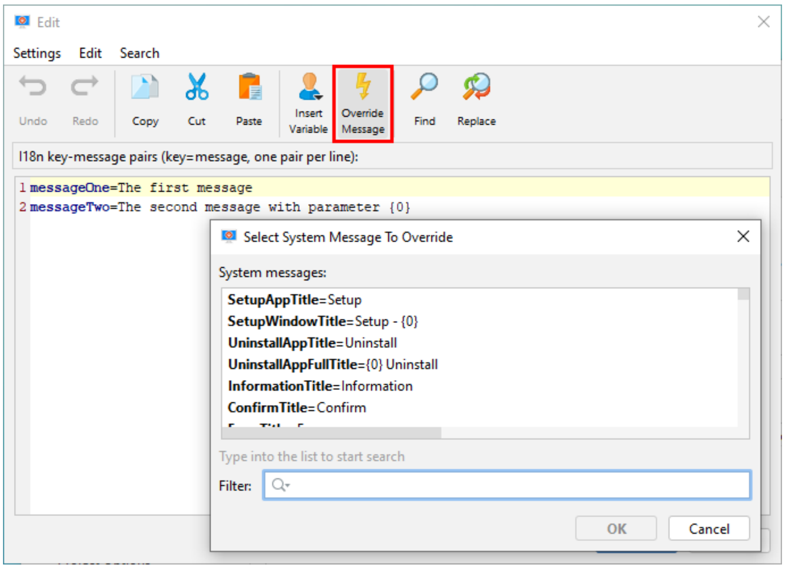
Message in Principal Language

The first message

If any standard message in the installer is not appropriate for your purpose, you can override it by looking up the corresponding keys in the appropriate message file with the path

```
<install4j installation directory>/resource/messages/messages_*.utf8
```

and defining the same key in your custom localization file. The built-in editor has an "Override message" tool bar button that helps you to find the message of interest and inserts the key-value pair in the editor.



I18n key-message pairs (key=message, one pair per line):

```
messageOne=The first message
messageTwo=The second message with parameter {0}
```

System messages:

SetupAppTitle=Setup
SetupWindowTitle=Setup - {0}
UninstallAppTitle=Uninstall
UninstallAppFullTitle={0} Uninstall
InformationTitle=Information
ConfirmTitle=Confirm

Type into the list to start search

Filter:

Parameters in i18n messages

If required, you can use parameters for your messages by using the usual `{n}` syntax in the value and listing the parameters with a function-like syntax after the key name. For example, if your key name is `myKey` and your message value is

```
Create {0} entries of type {1}
```

you can use a variable

```
${i18n:myKey("5", "foo")}
```

in order to fill the parameters, so that the actual message becomes

```
Create 5 entries of type foo
```

However, in the context of localizing an installer this is rarely necessary. Should you need to include a literal variable expression `{n}` in the message, you have to quote it like `'{'n'}'`.

Another way of adding parameters to i18n messages is to use compiler or installer variables. Compiler variables are replaced at build time and installer variables are replaced at runtime. For example:

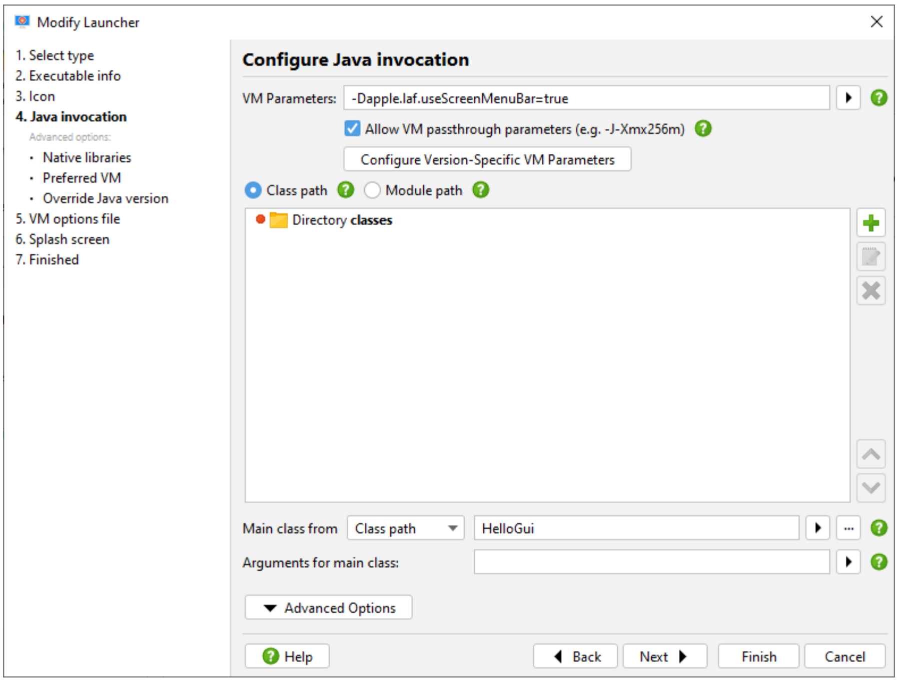
```
messageWithCompilerVariable=Title for ${compiler:sys.fullName}
messageWithInstallerVariable=Installing to ${installer:sys.installationDir}
```

A.14 VM Parameters

VM parameters can be passed to generated launchers [p. 36] in a variety of ways: You can specify fixed VM parameters, pass them on the command line or add them to a text file where the user or your application can edit them.

Fixed VM parameters

Fixed VM parameters can be configured in the launcher wizard [p. 36] where you can use compiler variables [p. 63] to handle platform-specific changes or launcher variables [p. 63] to use runtime-dependent paths.



Configure Java invocation

VM Parameters: -Dapple.laf.useScreenMenuBar=true

☑ Allow VM passthrough parameters (e.g. -J-Xmx256m)

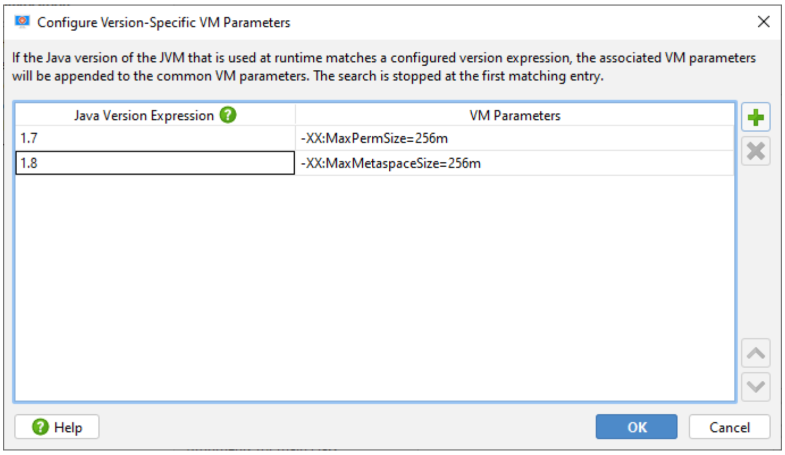
● Class path ○ Module path

Directory **classes**

Main class from: Class path — HelloGui

Arguments for main class:

install4j has the ability to add specific VM parameters depending on the Java version. To set this up, click on the *Configure version specific VM parameters* button. In the dialog, add rows for each range of Java versions that should receive specific VM parameters. If the Java version of the JVM that is used at runtime matches a configured version expression, the associated VM parameters will be appended to the common VM parameters. The search is stopped at the first matching entry. The syntax for the Java version expressions is explained by the help icon on the table header.



If the Java version of the JVM that is used at runtime matches a configured version expression, the associated VM parameters will be appended to the common VM parameters. The search is stopped at the first matching entry.

Java Version Expression	VM Parameters
1.7	-XX:MaxPermSize=256m
1.8	-XX:MaxMetaspaceSize=256m

Passing VM parameters on the command line

When executing a generated launcher, arguments are passed to the main class, so you cannot pass an argument like `-Xmx800m` and expect it to be interpreted as a VM parameter. To tell the launcher that you want to use a specific command line argument as a VM parameter, you have to prefix it with `-J`, as in

```
-J-Xmx800m
```

If this behavior is not desirable, you can deactivate it on the "Java invocation" step of the launcher wizard.

*.vmoptions files

A common requirement is the capability to adjust the VM parameters of launchers after the installation has been completed or to determine the VM parameters at installation time depending on the environment like the target platform or some user selection in the installer.

For this purpose, a parameter file in the same directory as the executable is read and its contents are added to the list of fixed VM parameters. The name of this parameter file is the same as the executable file with the extension `.vmoptions`.

For example, if your executable is named `hello.exe`, the name of the VM parameter file is `hello.vmoptions`. For GUI launchers on macOS, an additional `.vmoptions` file inside the application bundle with the relative path `Contents/vmoptions.txt` is read.

In the `.vmoptions` file, each line is interpreted as a single VM parameter and the last line must be followed by a line feed. install4j adapts your `.vmoptions` files during the compilation phase so that the line endings are suitable for all platforms. For example, the contents of the VM parameter file could be:

```
-Xmx256m
-Xms32m
```

The `.vmoptions` files allow the installer as well as expert users to modify the VM parameters for your generated launchers.

It is possible to include other `.vmoptions` files from a `.vmoptions` file with the syntax

```
-include-options [path to other .vmoptions file]
```

Recursive includes are supported. You can also add this option to the fixed VM parameters of a launcher. In that way, you do not have to create `.vmoptions` files for all your launchers, but you can have a single `.vmoptions` file for all of them.

This allows you to to centralize the user-editable VM options for multiple launchers and to have `.vmoptions` files in a location that can be edited by the user if the installation directory is not writable. You can use environment variables to find a suitable directory, for example

```
-include-options ${APPDATA}\My Application\my.vmoptions
```

on Windows and

```
-include-options ${HOME}/.myApp/my.vmoptions
```

on Unix. If you have to decide at runtime where the included `.vmoptions` file is located, use an installer variable:

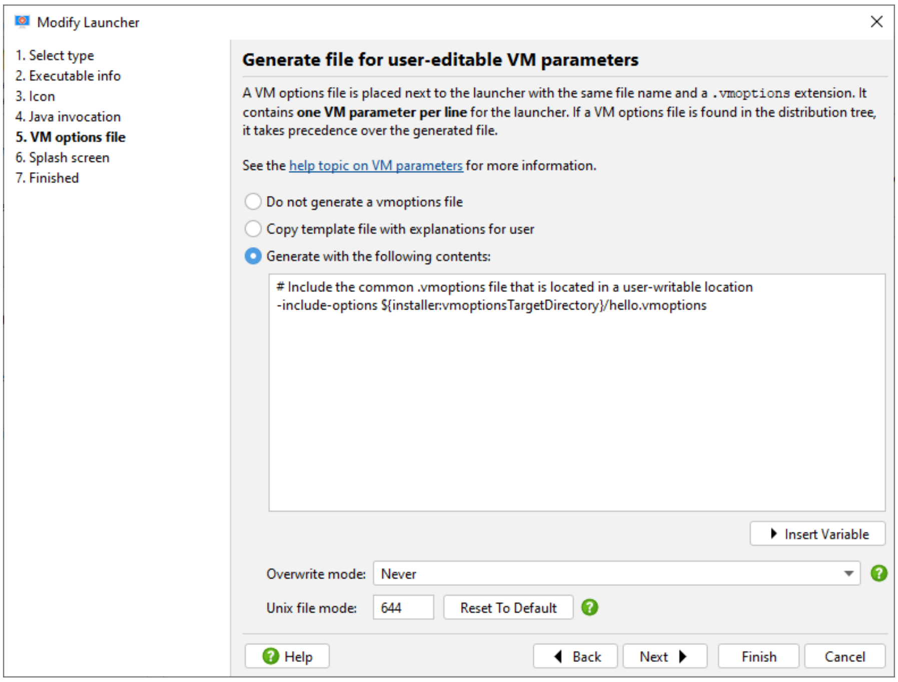
```
-include-options ${installer:vmOptionsTargetDirectory}/my.vmoptions
```

and add a "Replace installer variables in a text file" action to replace it after you have set the the `vmOptionsTargetDirectory` installer variable to a suitable path with a "Set a variable" action.

In addition to the VM parameters you can also modify the classpath in the `.vmoptions` files with the following options:

- **-classpath [classpath]**
  Replace the classpath of the generated launcher.
- **-classpath/a [classpath]**
  Append to the classpath of the generated launcher.
- **-classpath/p [classpath]**
  Prepend to the classpath of the generated launcher.

Instead of adding your own `.vmoptions` to the distribution tree, you can set up an initial `.vmoptions` file on the "VM options file" step of the launcher wizard, either with a template or with your own pre-defined content. Overwrite mode and file rights can also be configured in this step.



Generate file for user-editable VM parameters

A VM options file is placed next to the launcher with the same file name and a `.vmoptions` extension. It contains **one VM parameter per line** for the launcher. If a VM options file is found in the distribution tree, it takes precedence over the generated file.

See the help topic on VM parameters for more information.

○ Do not generate a vmoptions file
○ Copy template file with explanations for user
● Generate with the following contents:

```
# Include the common .vmoptions file that is located in a user-writable location
-include-options ${installer:vmoptionsTargetDirectory}/hello.vmoptions
```

Overwrite mode: Never

Unix file mode: 644

Environment variables

You can use environment variables in the fixed VM parameters and in the `.vmoptions` file with the syntax `${variableName}` replacing `variableName` with the name of the environment variable.

This environment variable syntax also works in the arguments text field and the classpath configuration.

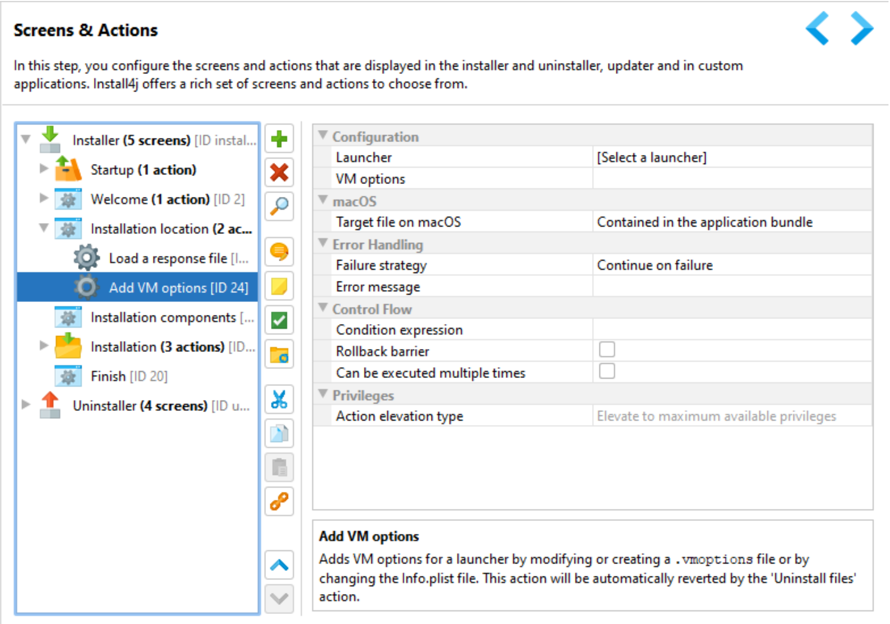
"Add VM options" action

With the "Add VM options" action [p. 169], you can handle VM parameter additions to the `.vmoptions` file in the installer. The action creates a `.vmoptions` file if necessary or adds your options if it already exists.

A number of VM parameters can only occur once, so the action replaces the following parameters if they already exist:

- -Xmx
- -Xms
- -Xss
- -Xloggc
- -Xbootclasspath
- -verbose
- -ea / -enableassertions
- -da / -disableassertions
- -splash

as well as the install4j-specific classpath modification options that can be used in `.vmoptions` files.


Installer (5 screens) [ID instal...
Startup (1 action)
Welcome (1 action) [ID 2]
Installation location (2 ac...
Load a response file [I...
Add VM options [ID 24]
Installation components [...
Installation (3 actions) [ID...
Finish [ID 20]
Uninstaller (4 screens) [ID u...

Screens & Actions

In this step, you configure the screens and actions that are displayed in the installer and uninstaller, updater and in custom applications. Install4j offers a rich set of screens and actions to choose from.

Configuration	
Launcher	[Select a launcher]
VM options	
macOS	
Target file on macOS	Contained in the application bundle
Error Handling	
Failure strategy	Continue on failure
Error message	
Control Flow	
Condition expression	
Rollback barrier	☐
Can be executed multiple times	☐
Privileges	
Action elevation type	Elevate to maximum available privileges

Add VM options

Adds VM options for a launcher by modifying or creating a `.vmoptions` file or by changing the Info.plist file. This action will be automatically reverted by the 'Uninstall files' action.

To set an `-Xmx` value to a fraction the total memory of the target system, you can use a "Set a variable action" that calculates the numeric part of the `-Xmx` value using the utility method `SystemInfo.getPhysicalMemory()`. In the second step you use that variable in the "VM options" property of the "Add VM options" action. For example, if you want to set the maximum heap size to 50% of the total memory, you do the following after the "Install files" action:

1. Add a "Set a variable" action with variable name "xmx" and a script of

   ```
   "-Xmx" + Math.round(SystemInfo.getPhysicalMemory() * 0.5 / 1024 / 1024) + "m"
   ```

2. Add a "Add VM options" action with VM options

   ```
   ${installer:xmx}
   ```

A.15 JRE Bundles

When deploying a Java application, it is recommended to bundle a JRE. While a JRE with the required version may be available in a controlled environment, it is generally far less error-prone to ship a JRE with each media file. Any JRE bundle that is installed by install4j is private to your application and will not interfere with other applications.

install4j offers two ways to create JRE bundles. You can either let install4j download JDK archives from well-known OpenJDK providers and create JRE bundles from them on the fly, or you can create JRE bundles yourself from installed JREs.

How JRE bundles work at runtime

install4j automatically adjusts the JRE search sequence [p. 36] of all generated launchers and includes the bundled JRE as the first choice. A bundled JRE is used automatically by the installer, the uninstaller, custom installer applications and the generated launchers.

A bundled JRE will always be distributed inside the installation root directory [p. 14], on Windows and Linux/Unix in the directory

```
<installation directory>/jre
```

and on macOS in

```
<content directory>/.install4j/jre.bundle
```

The content directory is available from the installer runtime variable `sys.contentDir` and resolves to the installation directory for folder media file types and `Contents/Resources/app` for archive media file types. The actual location of the JRE installation directory is available from the installer runtime variable `sys.preferredJre` after the "Install files" action has run.

When you update your application and include a new JRE bundle, the old JRE bundle will be deleted prior to the installation, so that any files left over from the old JRE cannot interfere with the new JRE. With the "Update bundled JRE" property of the "Install files" action you can disable updates of JRE bundles.

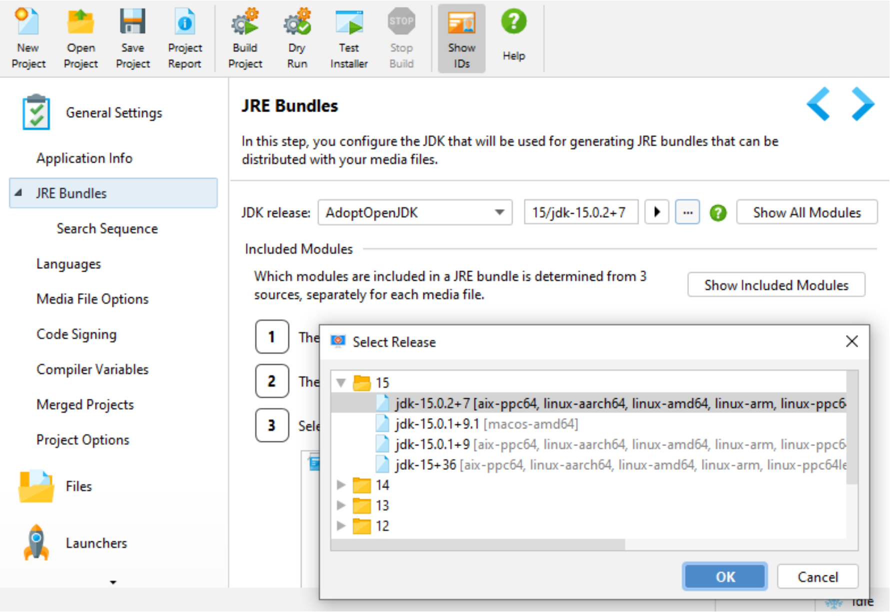
Generated JRE bundles

On the "General Settings->JRE Bundles" step, you can use the release chooser dialog to select a release from which you would like to create the JRE bundles. The available platforms are listed next to each release. The standard platform IDs are

- `windows-amd64` for 64-bit Windows
- `windows-x86` for 32-bit Windows
- `macos-amd64` for macOS on x64
- `macos-aarch64` for macOS on ARM
- `linux-amd64` for 64-bit Linux
- `linux-x86` for 32-bit Linux

Other platforms may be provided by the JDK providers and are usable in the Linux/Unix media files.

By default, AdoptOpenJDK (1) is set as the JDK provider and is recommended for general purpose usage. For JavaFX applications, the Liberica (2) and the Zulu (3) providers are convenient, because JavaFX is already included and you don't have to work with separately downloaded JMOD files. Liberica also offers an especially wide range of Linux architectures. For Swing desktop applications, the JetBrains Runtime (4) is the best choice because it contains a lot of fixes that are not included in the upstream OpenJDK. Finally, Amazon Corretto (5) is an OpenJDK distribution that focuses on including additional fixes and patches from the main branch and other sources into older releases.



JRE Bundles

In this step, you configure the JDK that will be used for generating JRE bundles that can be distributed with your media files.

JDK release: AdoptOpenJDK 15/jdk-15.0.2+7

Included Modules

Which modules are included in a JRE bundle is determined from 3 sources, separately for each media file.

Select Release

- 15
  - jdk-15.0.2+7 [aix-ppc64, linux-aarch64, linux-amd64, linux-arm, linux-ppc6
  - jdk-15.0.1+9.1 [macos-amd64]
  - jdk-15.0.1+9 [aix-ppc64, linux-aarch64, linux-amd64, linux-arm, linux-ppc6
  - jdk-15+36 [aix-ppc64, linux-aarch64, linux-amd64, linux-arm, linux-ppc64le
- 14
- 13
- 12

Selecting a release folder node in the chooser dialog rather than a node for a specific release will insert a key ending in `/latest`. At compile time, the latest release that includes the required platform will be taken.

To add new JDK providers, an SPI is provided in `resource/jdk-provider.jar`. The associated Javadoc in the archive `resource/jdk-provider-javadoc.jar` has more information.

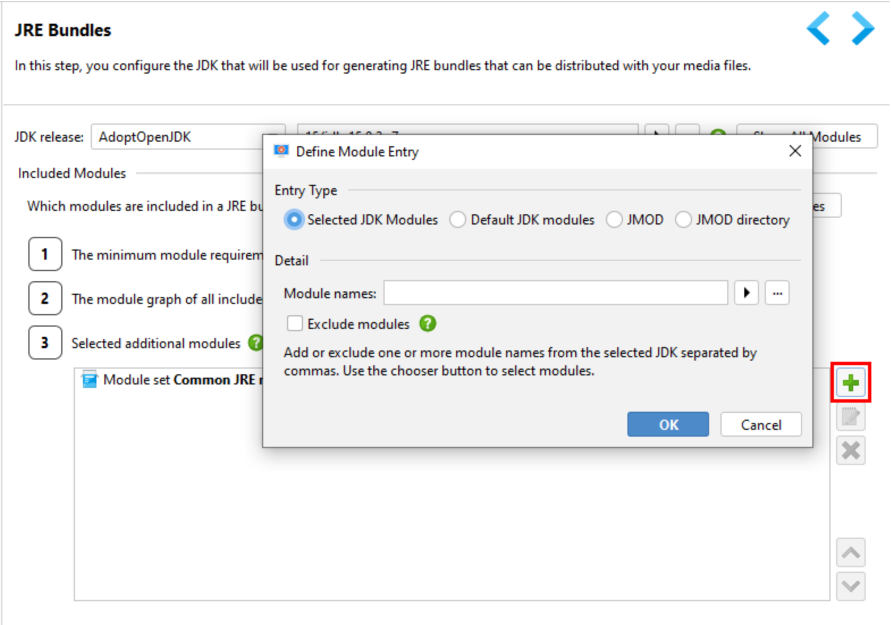
Downloaded JDK bundles contain all kinds of modules that you do not need in your distribution. On the other hand, you may have a set of JMODs that have to be linked into the JRE bundle, such as JavaFX (6). With your configuration in the module selector you can include a base set of modules, single named modules and additional JMODs. By default, a "JRE" with commonly used modules is linked, but the module sets "Minimum" and "All" are also available.

install4j always adds modules that are required by the install4j runtime. This includes the `java.desktop` module which is required even if you only want to create console installers or archives. In addition, install4j scans the module requirements of your generated launchers [p. 36] and adds them automatically. With the *Show included modules* button, you can show the actual list

---

(1) https://adoptopenjdk.net/
(2) https://bell-sw.com/
(3) https://www.azul.com/downloads/zulu-community/?package=jdk
(4) https://confluence.jetbrains.com/display/JBR/JetBrains+Runtime
(5) https://aws.amazon.com/de/corretto/
(6) https://openjfx.io/

of modules that will be added to the JRE bundle. In Java 7 and Java 8 there is no module system, so the entire JRE is bundled for those versions.



JRE Bundles

In this step, you configure the JDK that will be used for generating JRE bundles that can be distributed with your media files.

JDK release: AdoptOpenJDK

Define Module Entry

Entry Type: ● Selected JDK Modules ○ Default JDK modules ○ JMOD ○ JMOD directory

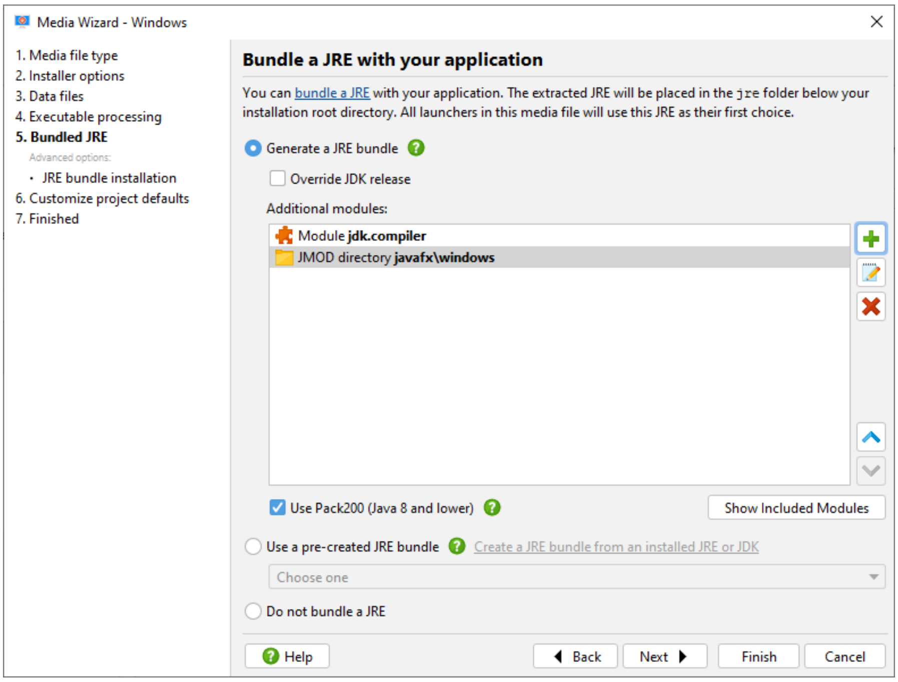
☐ Exclude modules

Add or exclude one or more module names from the selected JDK separated by commas. Use the chooser button to select modules.

Included Modules
Which modules are included in a JRE bu
1 The minimum module requirem
2 The module graph of all include
3 Selected additional modules
Module set Common JRE
Detail
Module names:
OK
Cancel
Modules

In the "Bundled JRE" step of the media wizard, the "Generate a JRE bundle" option is selected by default. You can, set it to "Do not bundle a JRE" in order to create media files without JRE bundles. Furthermore, you can customize the common JRE bundle configuration.

In addition to overriding the JDK provider and the release, you can specify additional modules and JMOD files that should be included for the current media file. The *Show included modules* button on this step uses the JDK bundle for the target platform unlike the corresponding button on the "General Settings->JRE Bundles" step which uses the JDK bundle for the current platform. This can lead to slight differences because JDKs contains platform-specific modules.



Bundle a JRE with your application

You can bundle a JRE with your application. The extracted JRE will be placed in the `jre` folder below your installation root directory. All launchers in this media file will use this JRE as their first choice.

● Generate a JRE bundle

☐ Override JDK release

Additional modules:

- Module **jdk.compiler**
- JMOD directory **javafx\windows**

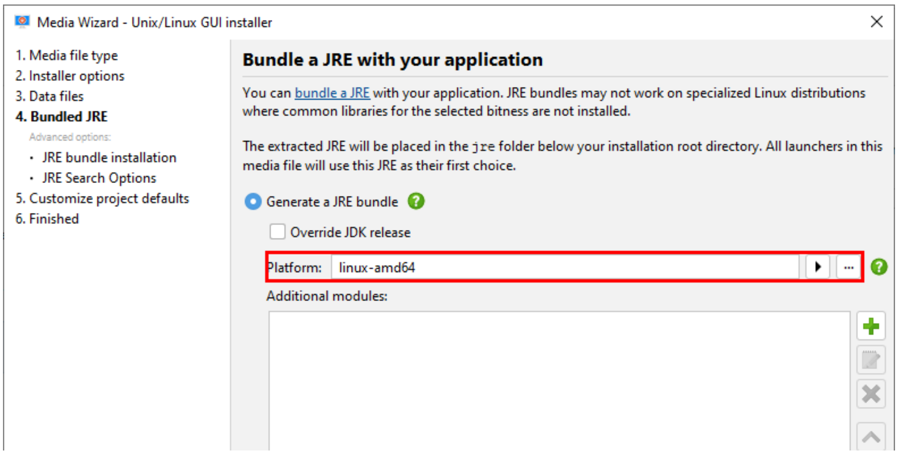
☑ Use Pack200 (Java 8 and lower)

○ Use a pre-created JRE bundle — Create a JRE bundle from an installed JRE or JDK

Choose one

○ Do not bundle a JRE

For Unix/Linux media files, the actual platform must be defined on the "Bundled JRE" step of the media wizard. By default, it is set to `linux-amd64` which stands for 64-bit Linux. The chooser button displays a dialog with all platforms that are available for the selected release.



Bundle a JRE with your application

You can bundle a JRE with your application. JRE bundles may not work on specialized Linux distributions where common libraries for the selected bitness are not installed.

The extracted JRE will be placed in the `jre` folder below your installation root directory. All launchers in this media file will use this JRE as their first choice.

● Generate a JRE bundle

☐ Override JDK release

Platform: linux-amd64

Additional modules:

If Java 8 is bundled, you can optionally deactivate the Pack200 compression for JAR files in the JRE. In archives, for example, these JAR files are decompressed the first time when a generated launcher is executed, adding a possibly undesired lag. That is why Pack200 compression is not selected by default for archive media files. Pack200 compression is unavailable for macOS single bundle archives where the signature requirements forbid the modification of any included files.

install4j will cache both downloaded JDK bundles as well as generated JRE bundles in the JRE cache directory

```
%LOCALAPPDATA%\install4j\v<version>\cached_jres
```

on Windows.

```
~/Library/Caches/install4j/v<version>/cached_jres
```

on macOS, and

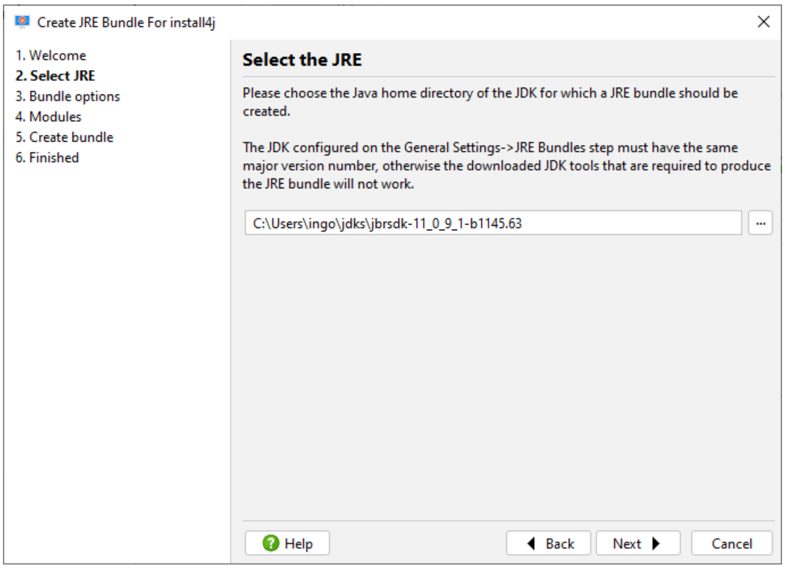
```
.caches/install4j/v<version>/cached_jres
```

on Linux and Unix where the root directory can be modified with the environment variable `XDG_CACHE_HOME`.

You can move the contents of this directory including the subdirectories "original" and "generated" to another machine to avoid downloads and speed up compilation. You can also delete this directory to force install4j to re-download all JDK bundles and generate new JRE bundles.

Pre-created JRE bundles

You can create a JRE bundle from any installed JRE on your file system. install4j offers the "Create a JRE bundle" wizard in the "Project" menu to make this task as simple as possible.



Select the JRE

Please choose the Java home directory of the JDK for which a JRE bundle should be created.

The JDK configured on the General Settings->JRE Bundles step must have the same major version number, otherwise the downloaded JDK tools that are required to produce the JRE bundle will not work.

C:\Users\ingo\jdks\jbrsdk-11_0_9_1-b1145.63

If you wish to automate the process, a command line tool [p. 224] for building JRE bundles is available with corresponding tasks in the Gradle, Maven or ant integrations.

Packaging your own JRE can be useful if you want to use JDK providers not supported by install4j (such as the official Oracle JDKs), or if you want to use runtime images that were created by jlink (7). The JRE bundle wizard only works for the platform you are running on. That means, to create a JRE bundle for Windows, you have to run install4j on Windows, to create a bundle for Linux, you have to run install4j on Linux.

All JREs are saved with a `tar.gz` extension to the root of the pre-created JRE directory which is

```
%LOCALAPPDATA%\install4j\v<version>\jres
```

on Windows.

---

(7) https://docs.oracle.com/en/java/javase/11/tools/jlink.html

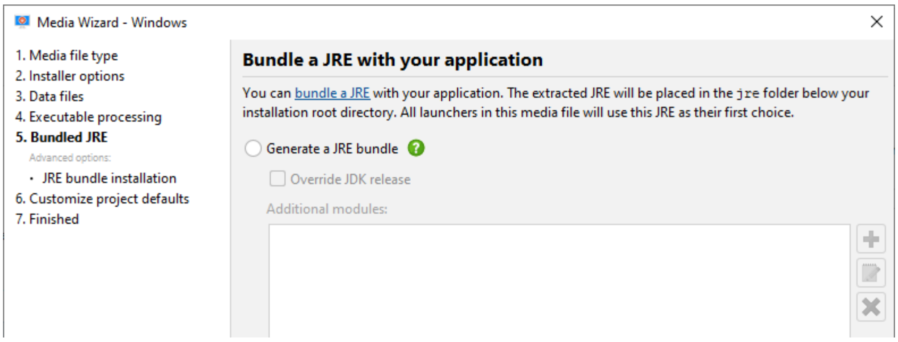
```
~/Library/Application Support/install4j/v<version>/jres
```

on macOS, and

```
.local/share/install4j/v<version>/jres
```

on Linux and Unix where the root directory can be modified with the environment variable `XDG_DATA_HOME`.

Pre-created JRE bundles can be selected in the "Bundled JRE" step of the media wizards



Bundle a JRE with your application

You can bundle a JRE with your application. The extracted JRE will be placed in the `jre` folder below your installation root directory. All launchers in this media file will use this JRE as their first choice.

○ Generate a JRE bundle

☐ Override JDK release

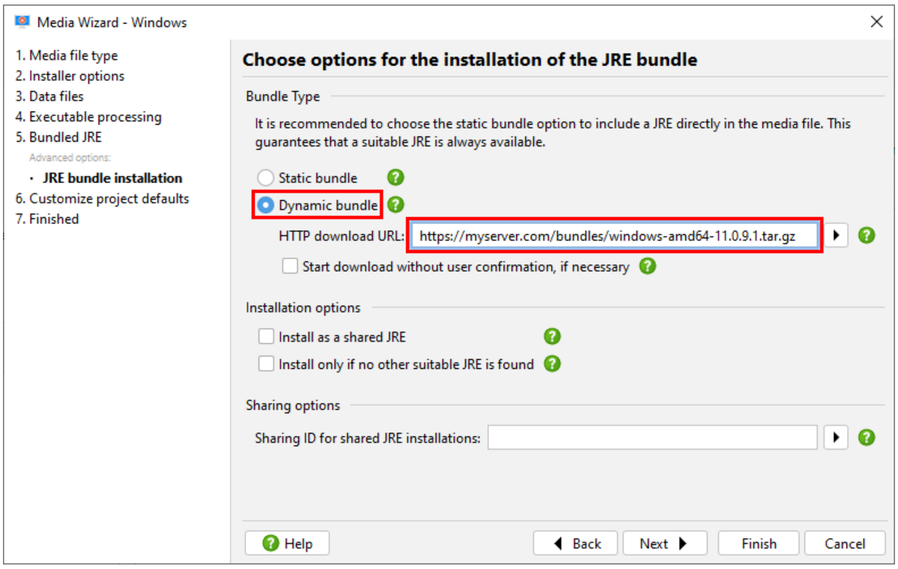
Additional modules:

If you would like to put your JRE bundles into a different directory, such as a directory in a version-controlled location, you can copy the `.tar.gz` file to that directory with the *Copy Bundle File* button and choose "Manual entry" in the JRE bundle drop-down to enter the path to the bundle file.

Dynamically downloaded JRE bundles

By default, JRE bundles are statically bundled and are always distributed along with your application. A dynamic bundle is downloaded on demand. If the user already has a suitable JRE installed, that JRE will be used. If there is no such JRE available on the target machine, the installer will download the dynamically bundled JRE from the URL that you have specified on the "Bundled JRE" step of the media wizard.

To enable the download on demand, you have to make the `.tar.gz` JRE bundle archive file available on a web server so that the configured `HTTP download URL` will point to that bundle archive. The URL has to be of the form `https://www.myserver.com/somewhere/windows-x86-11.tar.gz`. The build output displays the location of the JRE bundle file.



Choose options for the installation of the JRE bundle

Bundle Type

It is recommended to choose the static bundle option to include a JRE directly in the media file. This guarantees that a suitable JRE is always available.

○ Static bundle

● Dynamic bundle

HTTP download URL: https://myserver.com/bundles/windows-amd64-11.0.9.1.tar.gz

☐ Start download without user confirmation, if necessary

Installation options

☐ Install as a shared JRE

☐ Install only if no other suitable JRE is found

Sharing options

Sharing ID for shared JRE installations:

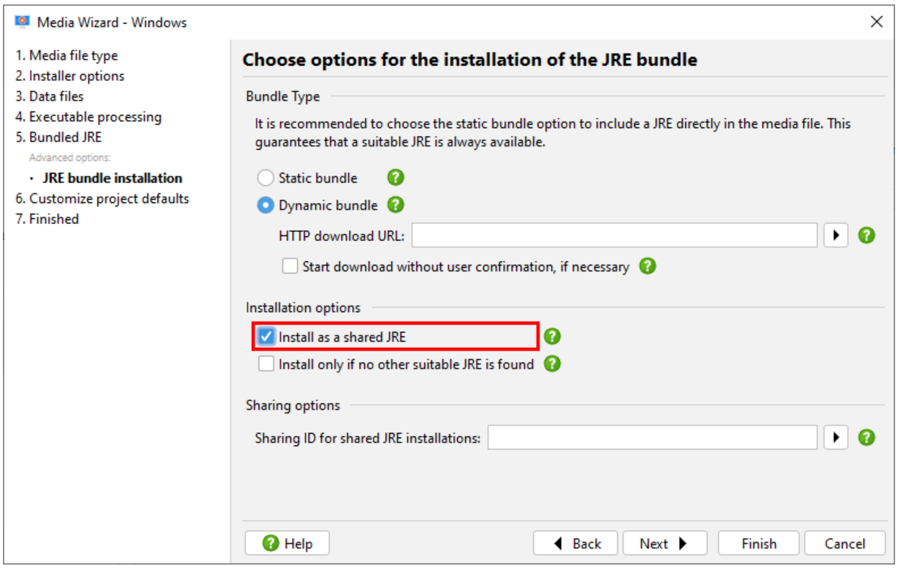
If the installer determines that there is no suitable JRE present, it will ask the user whether the JRE should be downloaded. If the `Start download without user confirmation, if necessary` check box has been selected, that confirmation is skipped and the download starts immediately.

If the download fails or is aborted by the user, the download URL will be displayed together with instructions on where to place the downloaded bundle archive.

You can override the default JRE search in a Windows installer executable by passing the argument `-manual` to the installer executable. The installer will then report that no JRE could be found and offer you to locate one in your file system. If you have set up a dynamic JRE bundle, it will also offer to start the download. This is a good way to test if your download URL is correct.

Shared JRE bundles

On Windows, Linux and Unix, it is possible to install JRE bundles as "shared", meaning that other installers generated by install4j will be aware of these bundles. A shared JRE bundle will not be uninstalled when the application that has installed the bundle is uninstalled itself.



Choose options for the installation of the JRE bundle

Bundle Type

It is recommended to choose the static bundle option to include a JRE directly in the media file. This guarantees that a suitable JRE is always available.

○ Static bundle

● Dynamic bundle

HTTP download URL:

☐ Start download without user confirmation, if necessary

Installation options

☑ Install as a shared JRE

☐ Install only if no other suitable JRE is found

Sharing options

Sharing ID for shared JRE installations:

Note that installers generated by install4j will never install a JRE on the system path or make Windows registry changes. The term "shared installation" only applies to applications distributed with install4j. Other applications will not be able to use such a JRE.

Both the installer that installs the shared JRE as well as the installers that want to use the shared JRE have to set the "Sharing ID" to the same string. This ensures that there is no sharing between installers that have different requirements for the JRE, such as the included modules.

If you dynamically bundle a shared JRE for multiple installers, the bundle will only be downloaded the first time when a user installs one of your installers. Subsequent installations of other installers will find the shared JRE installation.

JRE bundle format

In special cases you might want to create or modify a JRE bundle programmatically, without using the install4j IDE or the command line tools. This can be done with the standard GNU tools `tar` and `gzip`. A JRE bundle for install4j is simply a file with the naming scheme:

```
[operating system]-[architecture]-[JRE version].tar.gz
```

For windows bundles, the operating system name must be "windows", for macOS "macos", and for Linux and Unix any name can be used. The `.tar.gz` file contains the JRE `bin` and `lib` folders as top-level entries. The steps to create a bundle are outlined below:

```
cd jre
tar cvf minix-x86-11.tar *
gzip minix-x86-11.tar
cp minix-x86-11.tar.gz $HOME/.local/share/.install4j/v<version>/jres
```

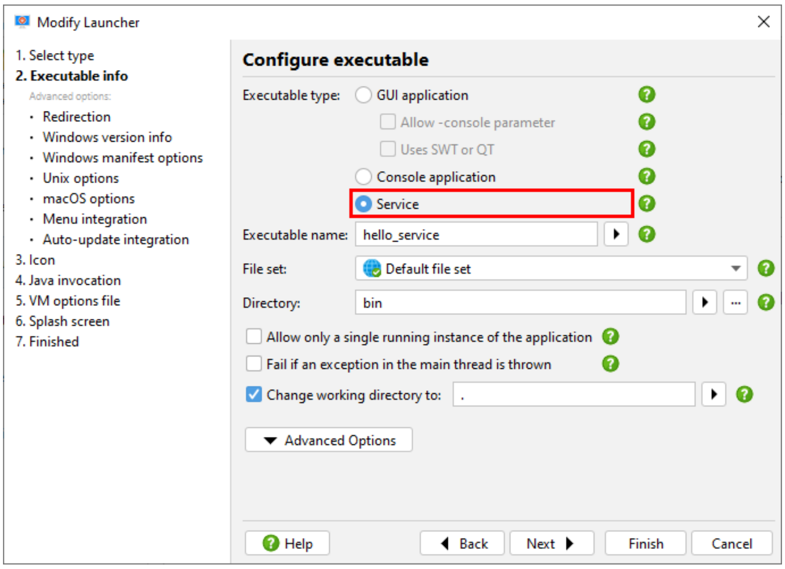
First you change into the top-level directory of the JRE, then you tar all files and directories and gzip the tar archive.

A.16 Services

Many applications have a component that has to run in the background without user interaction. On Windows, this is called a "service", on Unix a "daemon", in install4j the term "service" is used exclusively. install4j can generate service launchers for your application on all supported platforms. On Windows, managing services is a particularly demanding area and so other service executables that have not been generated by install4j are supported as well.

Generated service launchers

A service launcher will be generated if the selected executable type in the "Executable" step of the launcher wizard is set to "Service".



Configure executable

Executable type:
○ GUI application
☐ Allow -console parameter
☐ Uses SWT or QT
○ Console application
● Service

Executable name: hello_service

File set: Default file set

Directory: bin

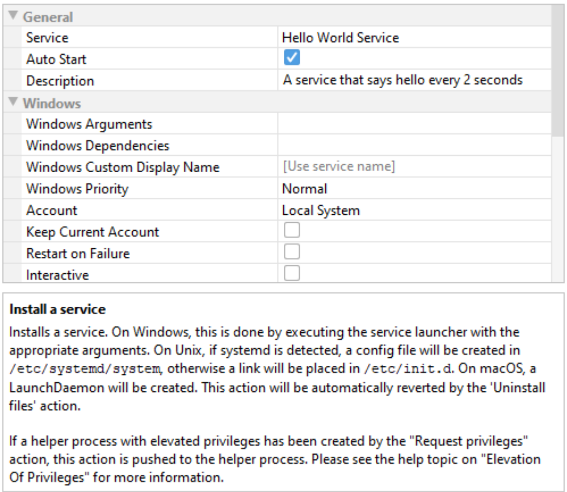
☐ Allow only a single running instance of the application
☐ Fail if an exception in the main thread is thrown
☑ Change working directory to: .

There are no special requirements and interfaces that have to be used by your code. When the service is started, the `main` method of the configured main class will be called just like for GUI or console launchers. Also, there is no special "shutdown" interface that is notified when the service is stopped. To perform cleanup, use the `Runtime.addShutdownHook()` method to register a thread that will be executed just before the JVM is terminated.

If you define a service launcher, it will not run automatically after the installation. A generated service launcher has to be installed and started explicitly. To do that, you have to add the following actions to the installer:

- **Install a service**

  This action registers a service with the system, so that it can be executed automatically when the computer is started. By default, the name of the installed service is the launcher name that is configured in the launcher section of the install4j IDE. In order to change the service name you have to rename the launcher.



General	
Service	Hello World Service
Auto Start	☑
Description	A service that says hello every 2 seconds
Windows	
Windows Arguments	
Windows Dependencies	
Windows Custom Display Name	[Use service name]
Windows Priority	Normal
Account	Local System
Keep Current Account	☐
Restart on Failure	☐
Interactive	☐

Install a service

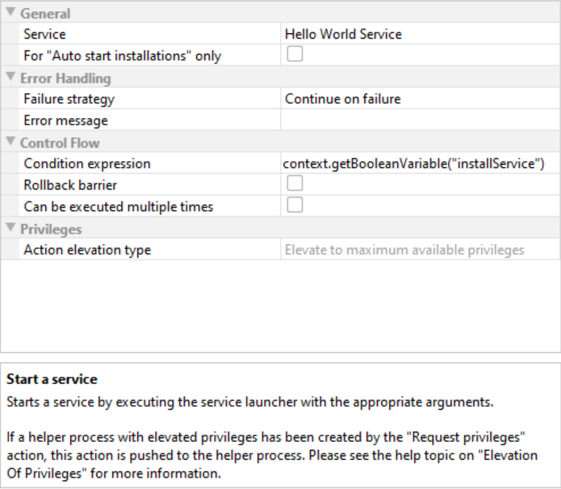
Installs a service. On Windows, this is done by executing the service launcher with the appropriate arguments. On Unix, if systemd is detected, a config file will be created in `/etc/systemd/system`, otherwise a link will be placed in `/etc/init.d`. On macOS, a LaunchDaemon will be created. This action will be automatically reverted by the 'Uninstall files' action.

If a helper process with elevated privileges has been created by the "Request privileges" action, this action is pushed to the helper process. Please see the help topic on "Elevation Of Privileges" for more information.

On Windows, if you require a user-configurable service name or if you wish to install the service multiple times, use the method for external service launchers as described below.

- **Start a service**

  Installing a service does not start it immediately and you have to add this separate action to actually run the service.



General	
Service	Hello World Service
For "Auto start installations" only	☐
Error Handling	
Failure strategy	Continue on failure
Error message	
Control Flow	
Condition expression	context.getBooleanVariable("installService")
Rollback barrier	☐
Can be executed multiple times	☐
Privileges	
Action elevation type	Elevate to maximum available privileges

Start a service

Starts a service by executing the service launcher with the appropriate arguments.

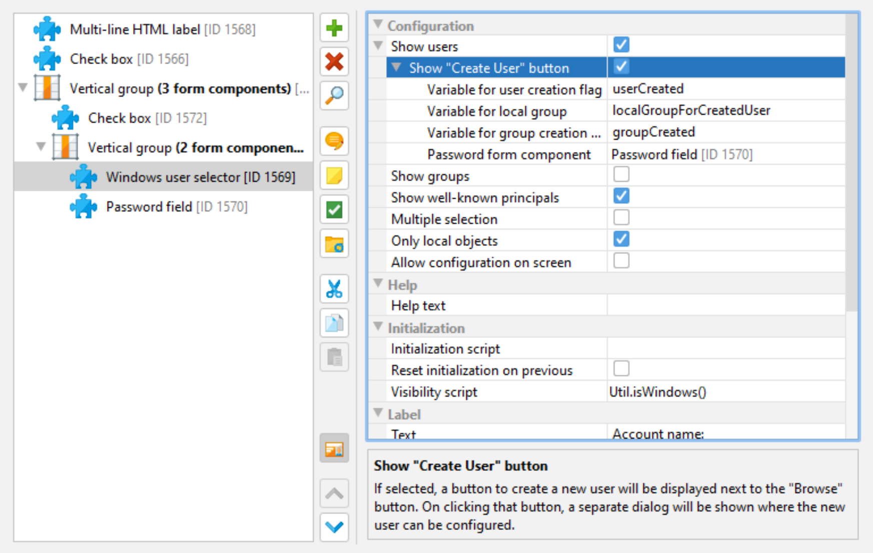
If a helper process with elevated privileges has been created by the "Request privileges" action, this action is pushed to the helper process. Please see the help topic on "Elevation Of Privileges" for more information.

When the "Install Files" action runs and a previous installation is being updated, any running services that are associated with the same executables are stopped.

Windows user accounts

On Windows, you can configure the user account that is used for running the service. There are a few well-known user accounts, like "Local System" (the default) or "Local Service" that you can choose directly in the configuration of this action.

In some cases, you might want to create a separate user to run a service. install4j offers API support for creating new user accounts with the `com.install4j.api.windows.WinUser` class. If you would like to query the user for details on the user account, it is possible to do that without using the API. On a configurable form, add a "Windows user selector" component and select the "Show 'Create User' button" property.


Multi-line HTML label [ID 1568]
Check box [ID 1566]
Vertical group (3 form components) [...
Check box [ID 1572]
Vertical group (2 form componen...
Windows user selector [ID 1569]
Password field [ID 1570]

Configuration	
Show users	☑
Show "Create User" button	☑
Variable for user creation flag	userCreated
Variable for local group	localGroupForCreatedUser
Variable for group creation ...	groupCreated
Password form component	Password field [ID 1570]
Show groups	☐
Show well-known principals	☑
Multiple selection	☐
Only local objects	☑
Allow configuration on screen	☐
Help	
Help text	
Initialization	
Initialization script	
Reset initialization on previous	☐
Visibility script	Util.isWindows()
Label	
Text	Account name:

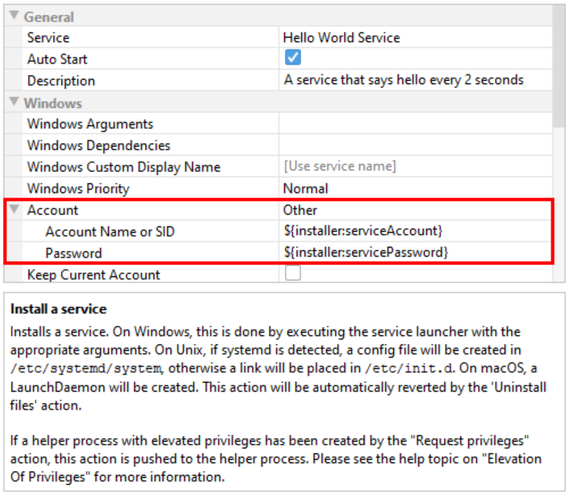
Show "Create User" button

If selected, a button to create a new user will be displayed next to the "Browse" button. On clicking that button, a separate dialog will be shown where the new user can be configured.

The SID of the created or selected user is saved to the configured variable, say "serviceUser".

You also have to query the user for the password of the account. For that purpose, add a "Password field" form component, set its variable to "servicePassword" and choose that form component in the "Password form component" property of the user selector form component.

In the "Install a service" action, you can then choose `Other` in the "Account" property and enter `${installer:serviceUser}` in the nested "Account name or SID" property as well as `${installer:servicePassword}` in the nested "Password" property.



General	
Service	Hello World Service
Auto Start	☑
Description	A service that says hello every 2 seconds
Windows	
Windows Arguments	
Windows Dependencies	
Windows Custom Display Name	[Use service name]
Windows Priority	Normal
Account	Other
Account Name or SID	${installer:serviceAccount}
Password	${installer:servicePassword}
Keep Current Account	☐

Install a service

Installs a service. On Windows, this is done by executing the service launcher with the appropriate arguments. On Unix, if systemd is detected, a config file will be created in `/etc/systemd/system`, otherwise a link will be placed in `/etc/init.d`. On macOS, a LaunchDaemon will be created. This action will be automatically reverted by the 'Uninstall files' action.

If a helper process with elevated privileges has been created by the "Request privileges" action, this action is pushed to the helper process. Please see the help topic on "Elevation Of Privileges" for more information.

Command line options of generated service launchers

Under some circumstances, services must be able to be installed and started manually from the command line. While this is required functionality on Unix, service executables on Windows usually offer no command line functionality. Instead, it is expected that there is a special program that installs an uninstalls the service.

This task is handled by the "Install a service" and "Uninstall a service" actions in install4j. In addition, you can start and stop services in the Windows service manager. install4j includes the "Start a service" and "Stop a service" actions to do this programatically in the installer.

To improve usability, install4j adds Unix-like arguments to the generated service launchers on Windows as well. For Unix and Windows service executables, the usual

```
my_service start   | my_service.exe /start
my_service stop    | my_service.exe /stop
my_service status  | my_service.exe /status
my_service restart | my_service.exe /restart
```

options for daemon start scripts are supported. The stop command waits for the service to shut down. The exit code of the status command is 0 when the service was running, 3 when it was not running and and 1 when the state cannot be determined, for example when it is not installed on Windows.

For debugging purposes, you may want to run the executable on the command line without starting it as a background service. This can be done with the `run` parameter.

```
my_service run | my_service.exe /run
```

In that case, all output will be printed on the terminal. If you want to keep the redirection settings, use the `run-redirect` parameter instead.

To install or uninstall a service on Windows from the command line, call

```
my_service.exe /install
my_service.exe /uninstall
```

In this way, your service is always started when Windows is booted. To prevent the automatic startup of your service, call

```
my_service.exe /install-demand
```

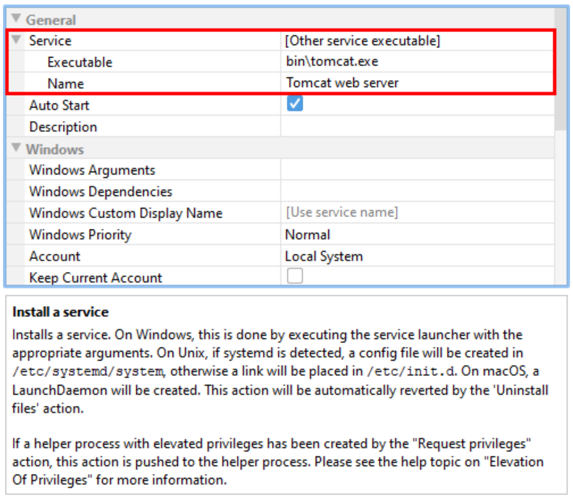
instead. As a second parameter after the `/install` parameter, you can optionally pass a service name. In that way you can

- install a service with a different service name than the default name.
- Use the same service executable to start multiple services with different names. To distinguish several running service instances at runtime, you can query the system property `exe4j.launchName` for the service name. Note that you also have to pass the same service name as the second parameter if you use the `/start`, `/restart`, `/status` `/stop` and `/uninstall` parameters.

On Windows, all command line switches also work with a prefixed dash instead of a slash, like `-uninstall` or with two prefixed dashes, like `--uninstall`.

External service launchers on Windows

When deploying third-party software, you may want to install and start services that were not generated by install4j. Both the "Install a service" action as well as the "Start a service" action provide a way to select other service executables. If you choose `[Other service executable]` in the drop-down list of the "Service" property, two new nested properties are shown: In the "Executable" property you set the path of the external service executable and the "Name" property allows you to specify the name of the installed service.



General	
Service	[Other service executable]
Executable	bin\tomcat.exe
Name	Tomcat web server
Auto Start	☑
Description	
Windows	
Windows Arguments	
Windows Dependencies	
Windows Custom Display Name	[Use service name]
Windows Priority	Normal
Account	Local System
Keep Current Account	☐

Install a service

Installs a service. On Windows, this is done by executing the service launcher with the appropriate arguments. On Unix, if systemd is detected, a config file will be created in `/etc/systemd/system`, otherwise a link will be placed in `/etc/init.d`. On macOS, a LaunchDaemon will be created. This action will be automatically reverted by the 'Uninstall files' action.

If a helper process with elevated privileges has been created by the "Request privileges" action, this action is pushed to the helper process. Please see the help topic on "Elevation Of Privileges" for more information.

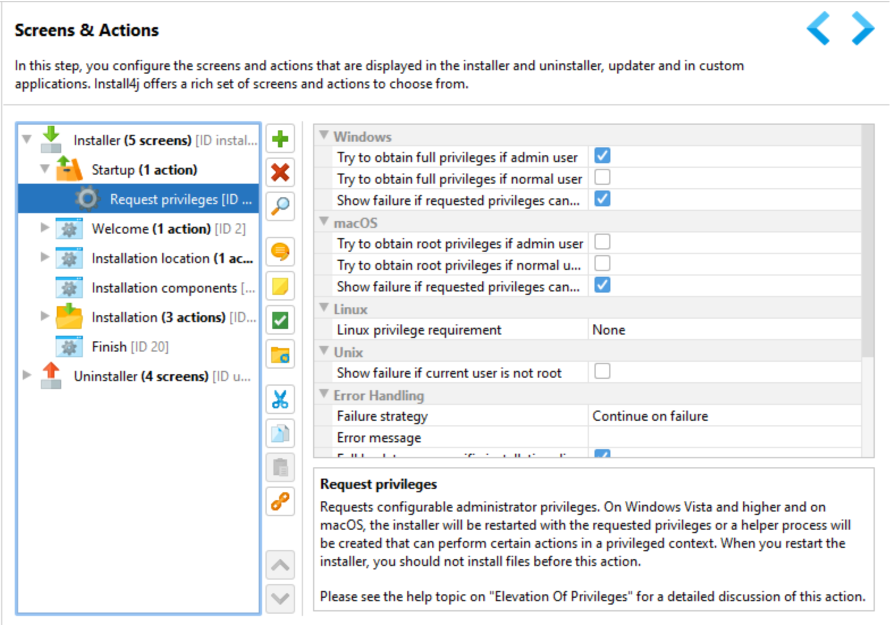
Note that this action does not provide "service wrapper" functionality for regular executables. The selected executable has to be a service executable, otherwise the action will not work.

A.17 Elevation Of Privileges

Most operating systems have the concept of normal users and administrators. While regular applications can run with limited privileges, installers often need full administrator privileges because they make modifications to the system that are not granted to limited users.

The required privileges depend on two factors: The operating system and the type of operations that are performed by the installer. The "Request privileges" action that is present in the "Startup" sequence of the default template for installers takes care of elevating the privileges to the required level and optionally terminating the installer with an error message if the required privileged cannot be obtained.

Due to the differences of the different operating systems, this configuration is made separately for Windows, macOS and Unix.



Screens & Actions

In this step, you configure the screens and actions that are displayed in the installer and uninstaller, updater and in custom applications. Install4j offers a rich set of screens and actions to choose from.

- Installer **(5 screens)** [ID instal...
  - Startup **(1 action)**
    - Request privileges [ID ...
  - Welcome **(1 action)** [ID 2]
  - Installation location **(1 ac...**
  - Installation components [...
  - Installation **(3 actions)** [ID...
  - Finish [ID 20]
- Uninstaller **(4 screens)** [ID u...

Windows	
Try to obtain full privileges if admin user	☑
Try to obtain full privileges if normal user	☐
Show failure if requested privileges can...	☑
macOS	
Try to obtain root privileges if admin user	☐
Try to obtain root privileges if normal u...	☐
Show failure if requested privileges can...	☑
Linux	
Linux privilege requirement	None
Unix	
Show failure if current user is not root	☐
Error Handling	
Failure strategy	Continue on failure
Error message	

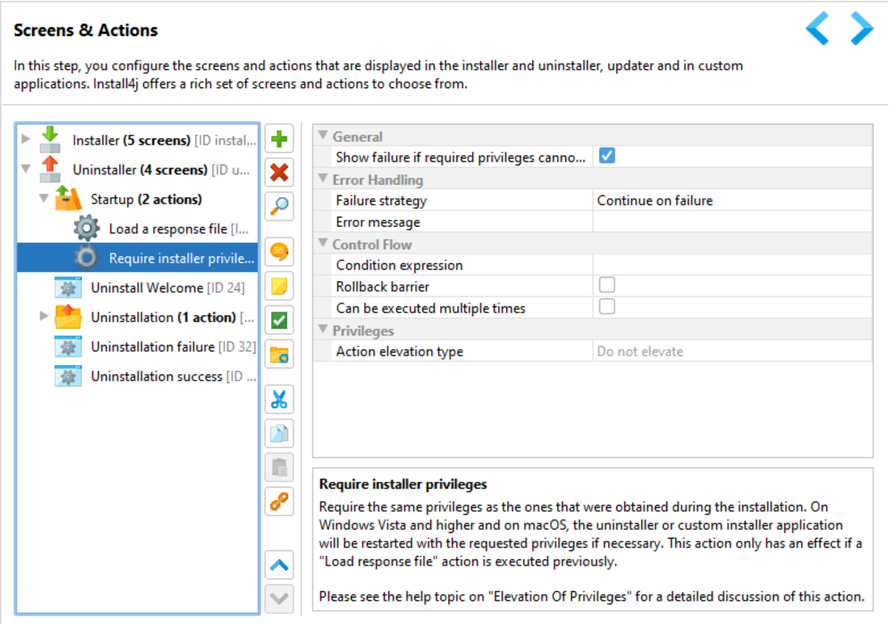
Request privileges

Requests configurable administrator privileges. On Windows Vista and higher and on macOS, the installer will be restarted with the requested privileges or a helper process will be created that can perform certain actions in a privileged context. When you restart the installer, you should not install files before this action.

Please see the help topic on "Elevation Of Privileges" for a detailed discussion of this action.

If the action fails, you can choose to not display an error message and switch to an installation directory in the user home directory with the "Fall back to user specific installation directory" property. Use `Util.hasFullAdminRights()` in condition expressions of actions that only work with elevated privileged in this case.

For the installer and the uninstaller, the privileges should be the same. This is why the default template for the uninstaller has a "Request installer privileges" action that will request the same privileges that were obtained in the installer.



Screens & Actions

In this step, you configure the screens and actions that are displayed in the installer and uninstaller, updater and in custom applications. Install4j offers a rich set of screens and actions to choose from.

- Installer **(5 screens)** [ID instal...
- Uninstaller **(4 screens)** [ID u...
  - Startup **(2 actions)**
    - Load a response file [I...
    - Require installer privile...
  - Uninstall Welcome [ID 24]
  - Uninstallation **(1 action)** [...
  - Uninstallation failure [ID 32]
  - Uninstallation success [ID ...

General	
Show failure if required privileges canno...	☑
Error Handling	
Failure strategy	Continue on failure
Error message	
Control Flow	
Condition expression	
Rollback barrier	☐
Can be executed multiple times	☐
Privileges	
Action elevation type	Do not elevate

Require installer privileges

Require the same privileges as the ones that were obtained during the installation. On Windows Vista and higher and on macOS, the uninstaller or custom installer application will be restarted with the requested privileges if necessary. This action only has an effect if a "Load response file" action is executed previously.

Please see the help topic on "Elevation Of Privileges" for a detailed discussion of this action.

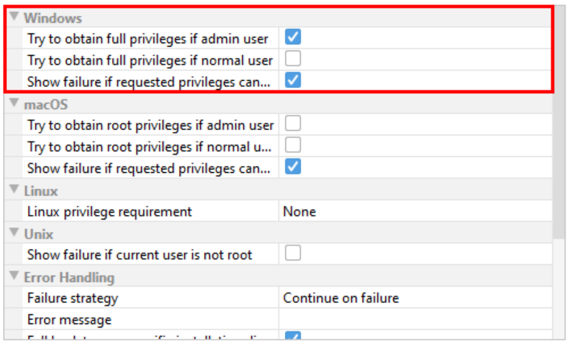
If you have more complex requirements, you can have multiple "Request privileges" actions with appropriate condition expressions, each with a link in the uninstaller.

Windows privileges

On Windows, "User Account Control" (UAC) (1) limits privileges for all users by default. An application can request full privileges, with different effects for normal users and admin users:

- A **normal user** cannot be elevated to full privileges, so the user has to enter credentials for a different administrator account. A normal user is not likely to have these credentials, so by default the "Request privileges" action does not try to obtain full privileges for normal users.

  Under some circumstances, for example if you want to manage services in your installer, you absolutely require full privileges. In this case, you can select the "Try to obtain full privileges if normal user" property in the Windows category.
- An **admin user** can be elevated. A UAC prompt will be shown in this case and the user simply has to agree in order to elevate privileges for the installer. Given that it is not possible to write to the program files directory without elevated privileges, this elevation is performed by default. With the "Try to obtain full privileges if admin user" property you can configure this behavior according to your own needs.



Windows	
Try to obtain full privileges if admin user	☑
Try to obtain full privileges if normal user	☐
Show failure if requested privileges can...	☑
macOS	
Try to obtain root privileges if admin user	☐
Try to obtain root privileges if normal u...	☐
Show failure if requested privileges can...	☑
Linux	
Linux privilege requirement	None
Unix	
Show failure if current user is not root	☐
Error Handling	
Failure strategy	Continue on failure
Error message	

(1) http://en.wikipedia.org/wiki/User_Account_Control

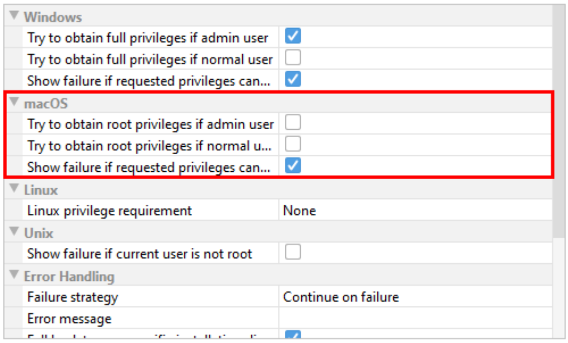
By default, the installer will fail if the requested privileges cannot be obtained. You can deselect the "Show failure if requested privileges cannot be obtained" property in the Windows category to continue and let the user install into the user home directory or another writable directory.

When you insert a service action and the elevation properties are not selected, you will be asked whether the necessary changes should be made automatically.

macOS privileges

Similar to Windows, macOS limits privileges for all users by default and normal users and admin users behave differently with respect to privilege elevation:

- A **normal user** cannot be elevated to full privileges, so the user has to enter the root password. A normal user is not likely to have the root password, so the "Request privileges" action does not try to obtain full privileges for normal users by default.
- To elevate an **admin user**, an authentication dialog will be shown and users have to enter their own password. Contrary to Windows, admin users can always write to the `/Applications` directory, even without full privileges. That is why on macOS no elevation of privileges is requested by default.



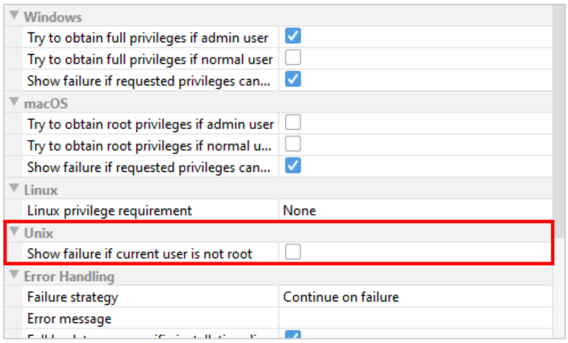
Windows	
Try to obtain full privileges if admin user	☑
Try to obtain full privileges if normal user	☐
Show failure if requested privileges can...	☑
macOS	
Try to obtain root privileges if admin user	☐
Try to obtain root privileges if normal u...	☐
Show failure if requested privileges can...	☑
Linux	
Linux privilege requirement	None
Unix	
Show failure if current user is not root	☐
Error Handling	
Failure strategy	Continue on failure
Error message	

Like on Windows, the installer will fail by default if the requested privileges cannot be obtained. In the default setting this has no effect, because privileges are never requested.

Service installations require full privileges, so the "Try to obtain full privileges if admin user" and the "Try to obtain full privileges if normal user" properties in the macOS category should be selected in that case. Again, the necessary changes will be suggested when service actions are inserted into the project.

Unix privileges

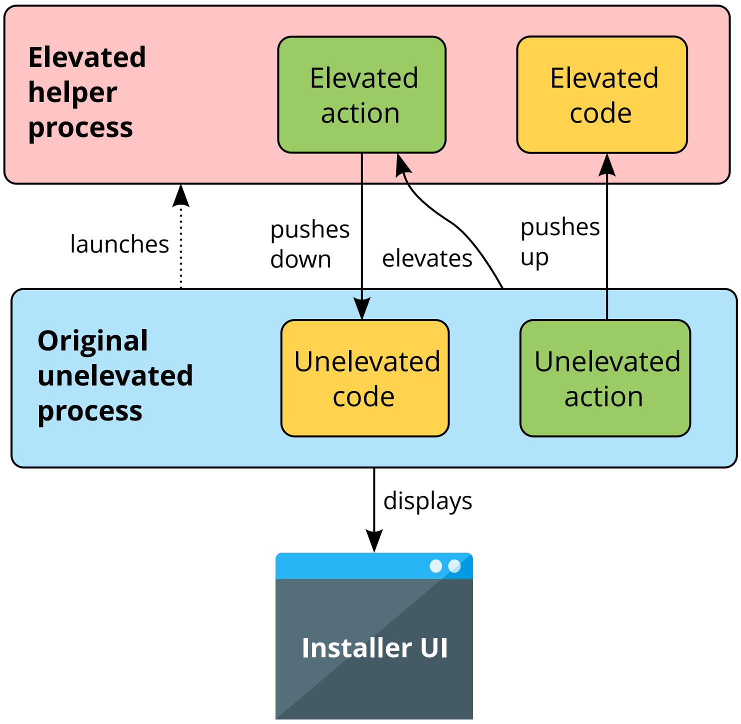
install4j does not support elevation of privileges on Linux and Unix. Partly this is due to the different incompatible systems of elevation, most notably "su" and "sudo" which cannot be easily detected. If full privileges are required, the user has to elevate the installer manually, either with "su" or with "sudo" or with the corresponding GUI tools. In this case, the "Show failure if current user is not root" has to be selected, so that an error message is shown if the installer is not started as root.



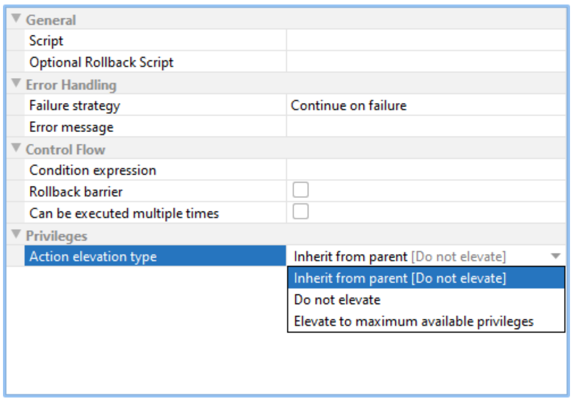
Windows
Try to obtain full privileges if admin user ☑
Try to obtain full privileges if normal user ☐
Show failure if requested privileges can... ☑
macOS
Try to obtain root privileges if admin user ☐
Try to obtain root privileges if normal u... ☐
Show failure if requested privileges can... ☑
Linux
Linux privilege requirement None
Unix
Show failure if current user is not root ☐
Error Handling
Failure strategy Continue on failure
Error message

Elevation mechanism

install4j does not elevate the entire process, but it starts an elevated helper process with full privileges.



All actions have an "Action elevation type" property that can be set to "Inherit from parent", "Do not elevate" or "Elevate to maximum available privileges". The root element in the hierarchy or beans is always an installer application whose "Action elevation type" property is set to "Do not elevate" by default.



General	
Script	
Optional Rollback Script	
Error Handling	
Failure strategy	Continue on failure
Error message	
Control Flow	
Condition expression	
Rollback barrier	☐
Can be executed multiple times	☐
Privileges	
Action elevation type	Inherit from parent [Do not elevate]

Inherit from parent [Do not elevate]
Do not elevate
Elevate to maximum available privileges

An action whose "Action elevation type" property results as "Elevate to maximum available privileges" will run in the elevated helper process. Such an action has full access to all installer variables as long as the contents of the variables are serializable.

Actions can have a preferred elevation type that is set automatically when you add the action. Actions that need to be elevated include

- the "Install files" and "Uninstall files" actions
- service actions
- actions that add rights on Windows
- actions that write files
- the "Run executable or batch file" action

Actions that are explicitly not elevated by default include

- the "Show URL" action
- the "Show file" action
- the "Execute launcher" action
- actions that should run as the original user, such as registry actions
- actions that interact with the GUI of the installer application

Elevated code can only interact with the GUI in a limited way. All methods in the `com.install4j.api.Util` class for displaying message dialogs or option dialogs are supported. You cannot call `context.getWizardContext()` or directly display a GUI using the Java Swing API. Also, calling methods in the `com.api.install4j.context.Context` that change screens is not supported. Most importantly, because an elevated action runs in a different process, you cannot access any static state in custom code. The only means to modify state from elevated actions are installer variables.

For your own scripts or your custom code, the API offers a way to push a piece of code to the elevated helper process or to the original process if they exist. This is done by wrapping the code in a `com.install4j.api.context.RemoteCallable` and calling `context.runElevated(...)` for the elevated helper process and `context.runUnelevated(...)` for the original process with the `RemoteCallable`:

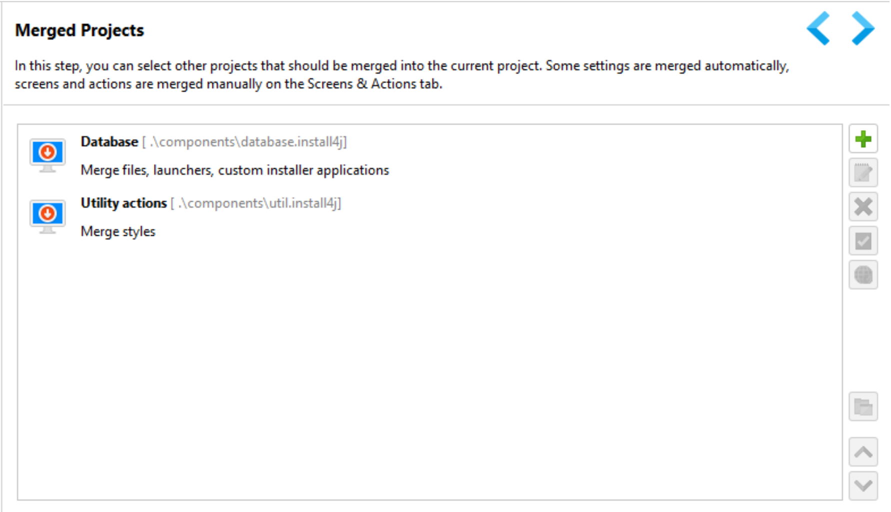
```
context.runElevated(new RemoteCallable() {
    public Serializable execute() {
        // do something in the elevated helper process
        return null;
    }
}, true);

context.runUnelevated(new RemoteCallable() {
    public Serializable execute() {
        // do something in the original process
        return null;
    }
});
```

The `RemoteCallable` must be serializable, so its fields can be transferred to the other process. Its `execute()` method that contains the code returns a `Serializable` so you can return a result to the calling process.

A.18 Merged Projects

There are two basic motivations for merged projects: First, there are large projects where a monolithic project file is inconvenient because multiple developers work on the same installer. Secondly, if you have multiple products that share certain components, it is undesirable to duplicate configuration for their installers.

The "merged projects" feature is a solution for both of these problems. You can create project files that are separate installers by themselves, such as a "database installer" and reuse them in multiple projects by adding them on the on the "General Settings->Merged Projects" step. On the other hand, you can also create project files that do not install anything by themselves, but just contain a collection of "Run script" actions that are useful in several of your installers.



Merged Projects

In this step, you can select other projects that should be merged into the current project. Some settings are merged automatically, screens and actions are merged manually on the Screens & Actions tab.

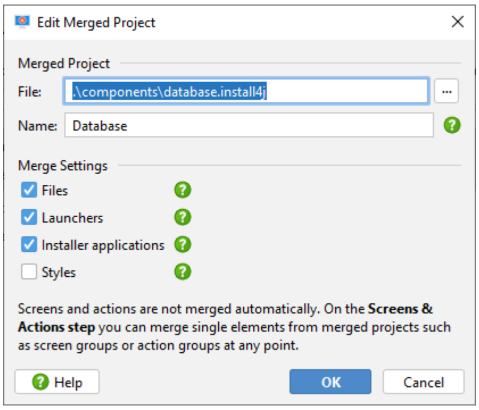
Database [.\components\database.install4j]
Merge files, launchers, custom installer applications

Utility actions [.\components\util.install4j]
Merge styles

Merged projects in install4j are not sub-projects that will retain their structure at runtime. Merging inserts selected elements into the main project before the main project is compiled.

Merge options

By default, files, launchers and custom installer applications are inserted. The corresponding merged elements are only added at compile-time and will not be visible in the main project. You can change merge options for each merged project individually.



Edit Merged Project

Merged Project

File: .\components\database.install4j

Name: Database

Merge Settings

☑ Files
☑ Launchers
☑ Installer applications
☐ Styles

Screens and actions are not merged automatically. On the **Screens & Actions step** you can merge single elements from merged projects such as screen groups or action groups at any point.

Merging works across an arbitrary number of levels and is performed in a bottom-to-top fashion: If the main project A includes a merged project B which in turn includes a merged project C, then C is first merged into B and the result is merged into A.

All selections are transitive for nested merged projects. For example, if the merged project contains another merged project for which merging of files is enabled, those files are only merged if file merging is enabled in the main project.

Merging of files

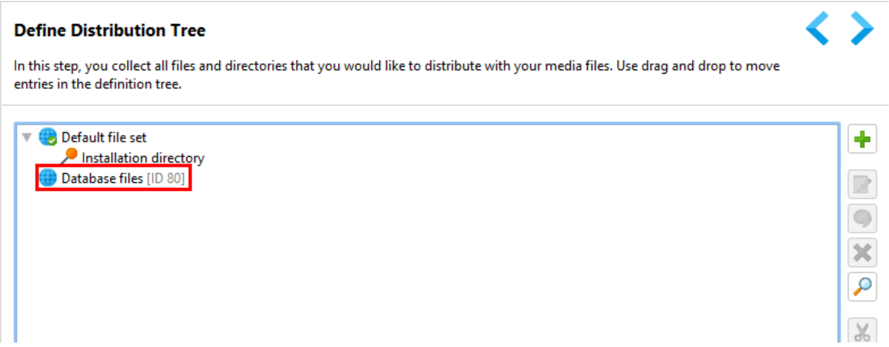
If you have enabled file merging for a merged project, files are merged automatically according to the following rules:

- All files from the default file set of the merged project are merged into the default file set of the main project.
- Roots are merged if the main project has roots with the same name, otherwise they are discarded.
- Files in each file set of the merged project are only merged if the main project has a file set with the same name.

The contained files in the merged project are not displayed in the main project. When defining installation components in the main project, you will only be able to select the entire file set. This means that the file sets in the merged project have to be as granular as required for your main project.

If there are files with the same relative paths, the main project has the highest precedence and the most deeply nested merged project has the lowest precedence. For merged projects on the same level, a project with a lower position in the list has a higher precedence than a project with a higher position.

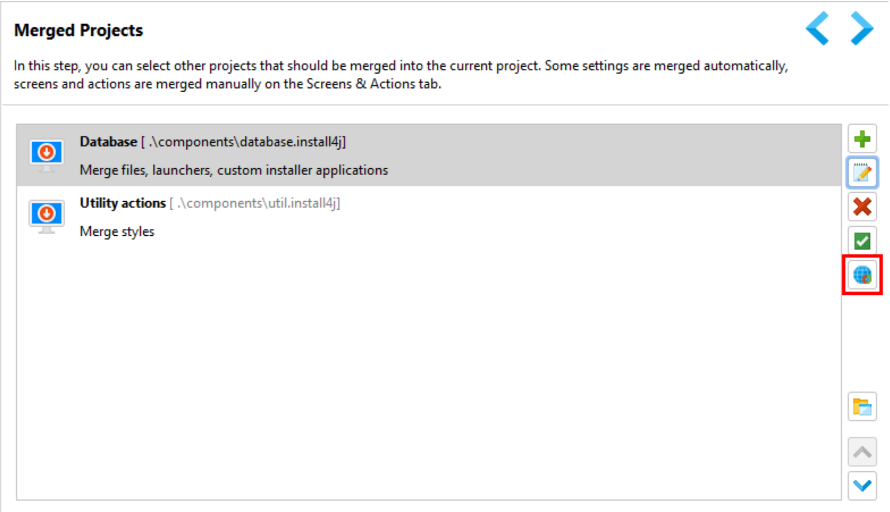
There is no merging of installation components. Installation components can only be defined in the main project. However, with the appropriate definition of file sets in merged projects you can easily contribute files to installation components in the main project. For example, if your merged project installs your database, and you want to ask the user whether to install the database, define a file set named "Database files" in the merged project and add all files to that file set. In your main project, you also add a file set named "Database files".



Define Distribution Tree

In this step, you collect all files and directories that you would like to distribute with your media files. Use drag and drop to move entries in the definition tree.

When adding the merged project, you will be asked whether to add that file set automatically to the main project. If file sets change later on, there is an action to repeat this synchronization. After invoking the action, the new file sets are displayed in the definition of the distribution tree [p. 14].



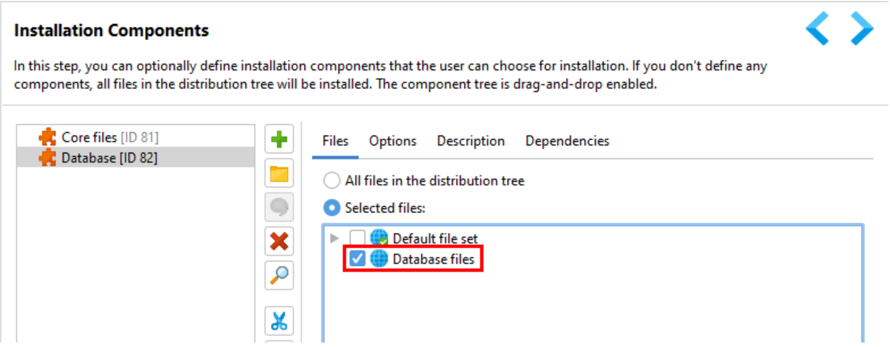
Merged Projects

In this step, you can select other projects that should be merged into the current project. Some settings are merged automatically, screens and actions are merged manually on the Screens & Actions tab.

Database [.\components\database.install4j]
Merge files, launchers, custom installer applications

Utility actions [.\components\util.install4j]
Merge styles

In your installation component for the database, choose the file set "Database files". It will not contain any files in the IDE, but during compilation, the files from the merged project will be added to it.



Installation Components

In this step, you can optionally define installation components that the user can choose for installation. If you don't define any components, all files in the distribution tree will be installed. The component tree is drag-and-drop enabled.

Core files [ID 81]
Database [ID 82]

○ All files in the distribution tree
● Selected files:

☐ Default file set
☑ Database files

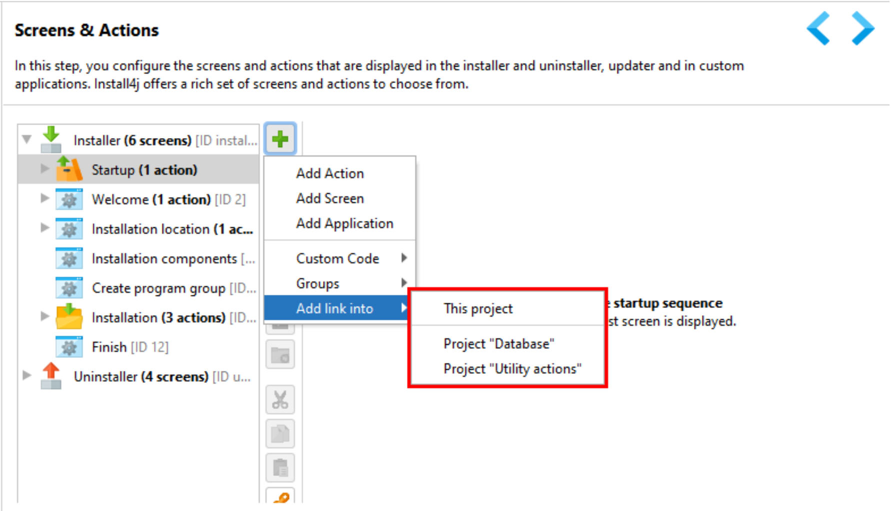
Merging of launchers and custom installer applications

All launchers and custom installer applications are merged if you have enabled the corresponding option for a merged project. It is not an error if there are collisions of launchers or custom installer applications with the same relative paths and the rules of precedence are the same as for the merging of files. However, it is recommend not to hide launchers in this way because this can lead to unexpected problems at runtime.

Both launchers and custom installer applications can be attributed to a particular file set. In that case, they are only merged if the file set also exists in the main project. The attribution to a particular installation component in the main project is done in the same way as for files.

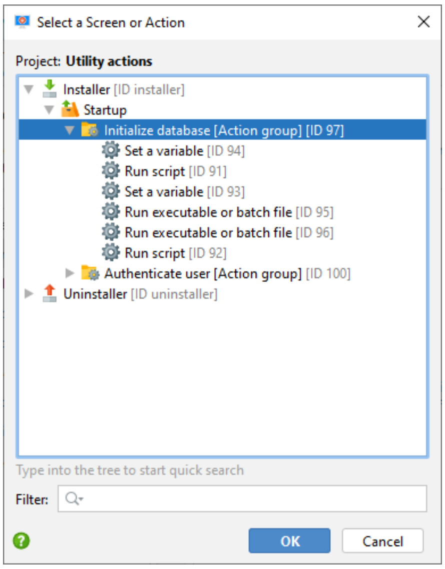
Merging of screens and actions

Screens and actions are not merged automatically, but through a selective placement of links on the "Installer->Screens & Actions" step [p. 148]. If merged projects are configured, the "Add link into" menu contains an entry for each merged project.

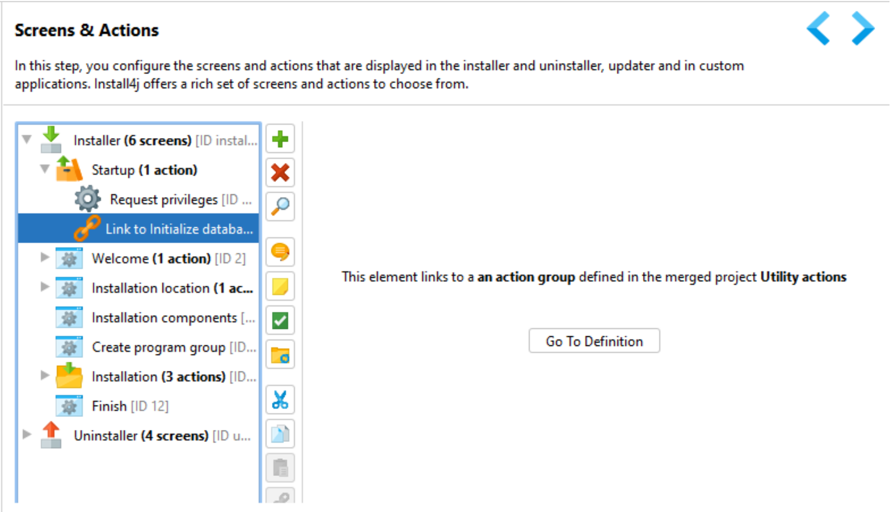

Screens & Actions
Installer (6 screens) [ID instal...
Startup (1 action)
Welcome (1 action) [ID 2]
Installation location (1 ac...
Installation components [...
Create program group [ID...
Installation (3 actions) [ID...
Finish [ID 12]
Uninstaller (4 screens) [ID u...
Add Action
Add Screen
Add Application
Custom Code
Groups
Add link into
This project
Project "Database"
Project "Utility actions"
startup sequence
st screen is displayed.

In this step, you configure the screens and actions that are displayed in the installer and uninstaller, updater and in custom applications. Install4j offers a rich set of screens and actions to choose from.

You can add multiple links to single screens and actions, but for more complex tasks it is advisable to create groups for related beans and add a link to a single group.


Select a Screen or Action
Project: Utility actions
Installer [ID installer]
Startup
Initialize database [Action group] [ID 97]
Set a variable [ID 94]
Run script [ID 91]
Set a variable [ID 93]
Run executable or batch file [ID 95]
Run executable or batch file [ID 96]
Run script [ID 92]
Authenticate user [Action group] [ID 100]
Uninstaller [ID uninstaller]
Type into the tree to start quick search
Filter:
OK
Cancel

When adding links, the install4j IDE, shows special nodes that do not show any structure but just a button that opens the target of the link in a different window. At compile-time, the target elements are inlined. This means that at runtime, it appears as if all merged elements were defined directly in the main project.



Screens & Actions

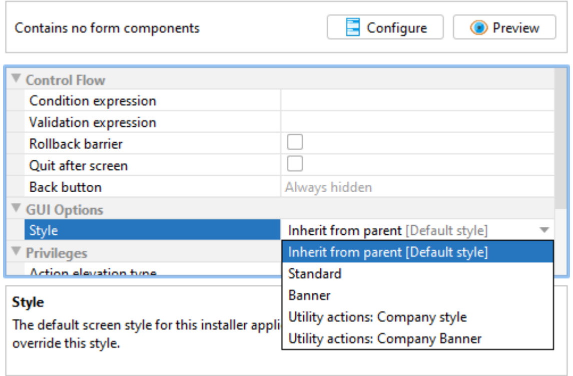
In this step, you configure the screens and actions that are displayed in the installer and uninstaller, updater and in custom applications. Install4j offers a rich set of screens and actions to choose from.

Installer (6 screens) [ID instal...
Startup (1 action)
Request privileges [ID ...
Link to Initialize databa...
Welcome (1 action) [ID 2]
Installation location (1 ac...
Installation components [...
Create program group [ID...
Installation (3 actions) [ID...
Finish [ID 12]
Uninstaller (4 screens) [ID u...
Go To Definition

This element links to a **an action group** defined in the merged project **Utility actions**

Merging of styles

If style merging is enabled, all styles from the main project are made available for installer applications, screen groups and screens. This allows you to centrally manage a set of styles and re-use it in multiple projects.



Contains no form components

Control Flow	
Condition expression	
Validation expression	
Rollback barrier	☐
Quit after screen	☐
Back button	Always hidden
GUI Options	
Style	Inherit from parent [Default style]
Privileges	
Action elevation type	

Inherit from parent [Default style]
Standard
Banner
Utility actions: Company style
Utility actions: Company Banner

Style

The default screen style for this installer appli
override this style.

See the help topic on styles [p. 55] for more information on how merged styles can be used in the project.

Flat merging considerations

As a result of flat merging, there are no intermediary artifacts for merged projects and the result of the compilation is a single monolithic installer. This has the advantage of being easy and flexible, but collisions can occur unless concerns are properly separated between the main project and its merged projects.

In particular, all elements in the final result share the same namespace for compiler and installer variables. All custom localization files are merged, so that localization in merged projects is not impacted unless there is a collision in the message keys. Such problems can be avoided if unique prefixes are used for compiler variables and installer variables as well as custom localization keys. For example in project A, all variables could be prefixed with "a." and in project B with "b.".

One area where such collisions are not possible is for IDs of any entity in a project, such as launchers, file sets, actions, screens or form components. When a project is merged, install4j prefixes all IDs with the application ID of that project.

For example, if the application ID of a merged project is "1406-2150-6354-3051" and a launcher has the ID "2265", the ID is changed to "1406-2150-6354-3051:2265" after merging. This ensures that all IDs remain unique no matter how many projects are merged. Beans (screens, actions and form components) in the merged project are passed a special context that automatically prefixes all unqualified IDs with this application ID. For example, if you have a script in your merged project that calls

```
context.getLauncherById("2265")
```

this will succeed, even though the ID is now actually "1406-2150-6354-3051:2265". If you want to access that same launcher configuration from a script in the main project, you would have to call

```
context.getLauncherById("1406-2150-6354-3051:2265")
```

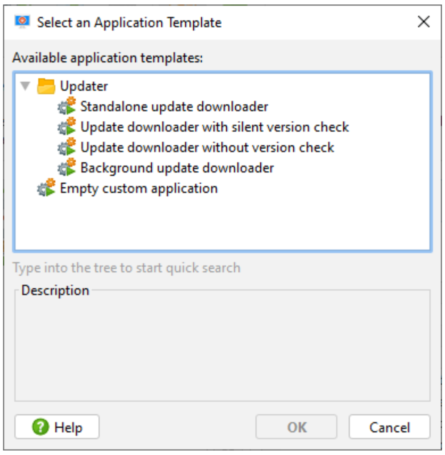
Generally, it is recommended to organize merged projects so that they are relatively self-contained and only interact with their main project through common installer variables. In that way, the main project can continue to work if the merged project is removed and the merged project can work as a standalone installer.

A.19 Auto-Update Functionality

install4j can help you to include auto-updating functionality to your application. Auto-updating includes two tasks: First, there must be a way to check if there is a newer version available for download. This check can be initiated by the user in various ways or the check can be triggered automatically by your application. Secondly, there must be a way to download and execute an appropriate installer for the new version.

install4j creates a special file named `updates.xml` in the media output directory when you build the project. This file describes the media files of the current version. If you want to use install4j's auto-update functionality, you have to upload this file to a web server. The file is then downloaded by deployed installations and delivers information about the current version.

Downloading and installing the new version is done with a custom installer application [p. 154]. install4j offers several templates for update downloaders that correspond to the update strategies outlined below.


Select an Application Template
Available application templates:
Updater
Standalone update downloader
Update downloader with silent version check
Update downloader without version check
Background update downloader
Empty custom application
Type into the tree to start quick search
Description
Help
OK
Cancel

Quick start

To get basic auto-update functionality for a GUI application, you can follow these instructions:

1. Upload the file `updates.xml` together with your media files to a directory on your web server.
2. Go to the "Installer->Auto-Update Options" step and enter the download URL for the `updates.xml` file. This must be the full URL for the file, like `https://www.server.com/download/updates.xml` and not just for the containing directory.
3. Go to the "Installer->Screens & Actions" step, click on the add button, choose *Add application* from the popup menu and select the "Update downloader with silent version check" template.
4. For the added update downloader application, enter the "Executable name" property, for example `automaticUpdater`.
5. Activate the "Launcher integration" tab for the new update downloader application and select the "Start automatically when launcher is executed" check box. Leave all other settings at their default.
6. In your installer, add a "Configurable form" and add an "Update schedule selector" form component to it.

In the installer, the user will get the possibility to choose the frequency of the update checks. When the user executes a launcher after you publish an update, the update downloader will be shown after the application window is displayed and tell the user about the new version. If the user accepts, the new installer is downloaded and installed.

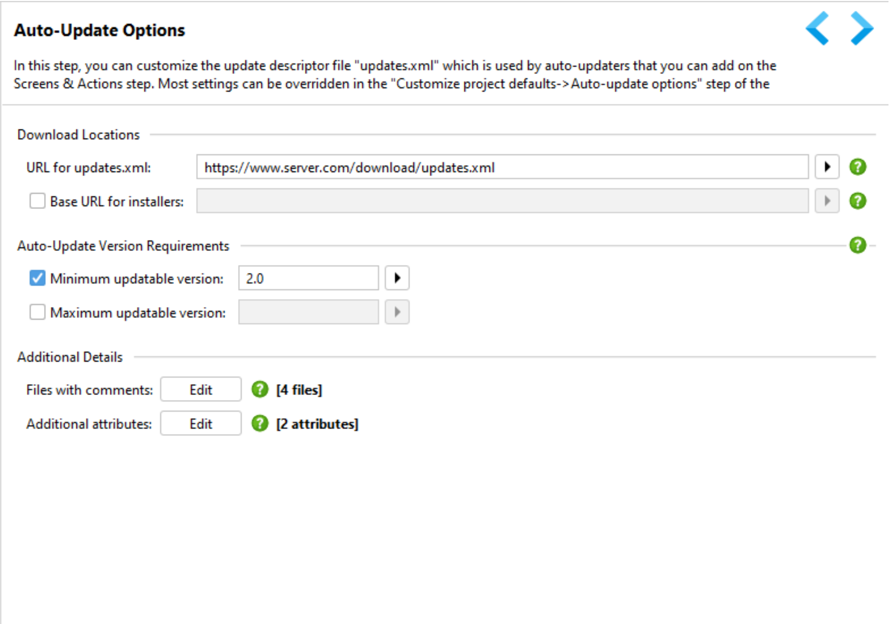
Of course your ideas for auto-update might be different. Maybe you do not have a GUI application and you want to perform unattended updates, or you want to notify your users about updates directly in your application. That is why the auto-update functionality has to be extremely flexible, with the unavoidable downside that its configuration is somewhat involved and there are a couple of concepts that you have to understand in order to be successful. The bulk of this flexibility comes from the fact that the update downloader is not a monolithic entity, but is composed of standard form components and actions that can be adjusted according to your particular requirements.

The updates.xml file

The `updates.xml` file is created in the media output directory [p. 126] each time you build the project. For advanced use cases, you can modify this file before uploading it to the web server. The file looks like the sample below:

```
<?xml version="1.0" encoding="UTF-8"?>
<updateDescriptor baseUrl="">
  <entry targetMediaFileId="8" updatableVersionMin="" updatableVersionMax=""
fileName="hello_windows_4_0.exe"
         newVersion="4.0" newMediaFileId="8" fileSize="2014720" bundledJre=""
myCustomAttribute="showWarning">
    <comment language="en">Hello world</comment>
    <comment language="de">Hallo Welt</comment>
    <comment language="it">Ciao mondo</comment>
  </entry>
  <entry targetMediaFileId="9" updatableVersionMin="" updatableVersionMax=""
fileName="hello_linux_4_0.rpm"
         newVersion="4.0" newMediaFileId="9" fileSize="817758" bundledJre="">
    <comment />
  </entry>
  <entry targetMediaFileId="10" updatableVersionMin="" updatableVersionMax=""
fileName="hello_macos_4_0.dmg"
         newVersion="4.0" newMediaFileId="10" fileSize="1359872" bundledJre="">
    <comment />
  </entry>
</updateDescriptor>
```

Its contents are derived from your input on the "Installer->Auto-Update Options" step where you define global options and common options that are replicated on all media file entries.



Auto-Update Options

In this step, you can customize the update descriptor file "updates.xml" which is used by auto-updaters that you can add on the Screens & Actions step. Most settings can be overridden in the "Customize project defaults->Auto-update options" step of the

Download Locations

URL for updates.xml: https://www.server.com/download/updates.xml

☐ Base URL for installers:

Auto-Update Version Requirements

☑ Minimum updatable version: 2.0

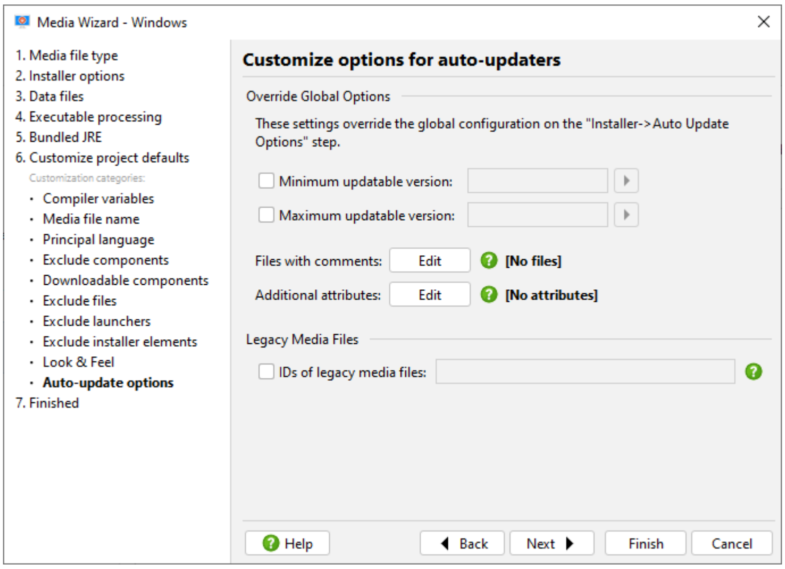
☐ Maximum updatable version:

Additional Details

Files with comments: Edit **[4 files]**

Additional attributes: Edit **[2 attributes]**

On the "Customize project defaults->Auto-update options" step of the media wizard you can override settings with specific values for the each media file.



Media Wizard - Windows

1. Media file type
2. Installer options
3. Data files
4. Executable processing
5. Bundled JRE
6. Customize project defaults
   Customization categories:
   - Compiler variables
   - Media file name
   - Principal language
   - Exclude components
   - Downloadable components
   - Exclude files
   - Exclude launchers
   - Exclude installer elements
   - Look & Feel
   - **Auto-update options**
7. Finished

Customize options for auto-updaters

Override Global Options

These settings override the global configuration on the "Installer->Auto Update Options" step.

☐ Minimum updatable version:

☐ Maximum updatable version:

Files with comments: Edit **[No files]**

Additional attributes: Edit **[No attributes]**

Legacy Media Files

☐ IDs of legacy media files:

The root of the `updates.xml` file is the `updateDescriptor` element. It has a `baseUrl` attribute that can be used to specify an alternate download URL for the installers and contains the value of the "Base URL for installers" setting on the "Installer->Auto-Update Options" step. By default, it is empty which means that the installers must be located in the same directory as the `updates.xml` file.

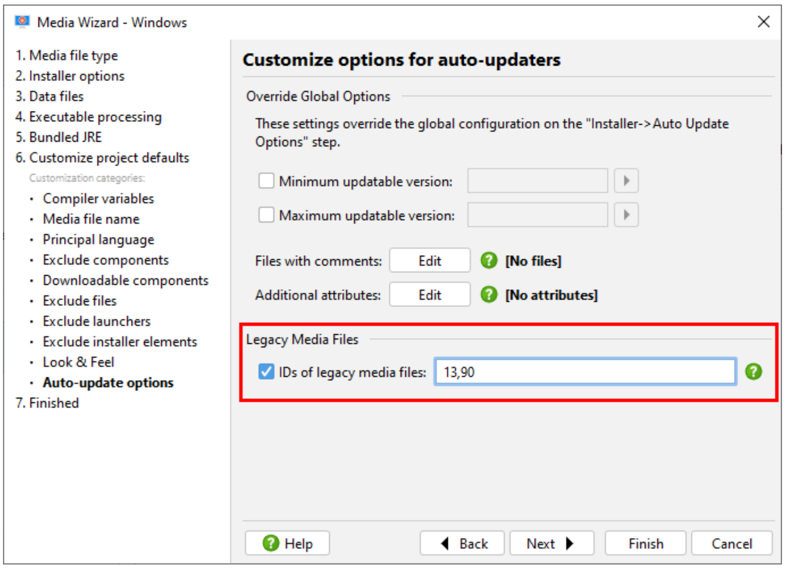
The `updateDescriptor` element contains one or more `entry` elements that correspond to the media files that were created by the build.

When install4j determines whether an entry in the update descriptor is a match for the current installation, it looks at three attributes of the `entry` element: Most importantly, the `targetMediaFileId` attribute has to match the media file ID of the current installation. You can show media file IDs by toggling the "Show IDs" tool bar button

Another criterion is the installed version of the application. Depending on that version, you might want to offer different updates. The `updatableVersionMin` and the `updatableVersionMax` attributes can set lower and upper limits for the installed versions that should download the associated entry in the update descriptor. By default, these attributes are empty, so no version restrictions apply. On the "Installer->Auto-Update Options" step, these versions can be set for all media files.

Attributes that describe the update installer include `fileName` which is necessary to construct the download URL, and `fileSize` which contains the size of the file in bytes. `newVersion` contains the available version while `newMediaFileId` is the media file ID of the update installer which is the same as `targetMediaFileId`. Lastly, `bundledJre` contains the original file name of the JRE bundle without the `.tar.gz` extension or the empty string if no JRE is bundled in the installer.

If you discontinue a media file, you can migrate users of that media file to a different media file with the legacy media file setting on the "Customize project defaults->Auto-update options" step of the media wizard. For each specified legacy ID, the entry for the current media file is duplicated. For more complex scenarios, you can modify the `updates.xml` file yourself and add additional entry elements as required.



Customize options for auto-updaters

Override Global Options

These settings override the global configuration on the "Installer->Auto Update Options" step.

☐ Minimum updatable version:

☐ Maximum updatable version:

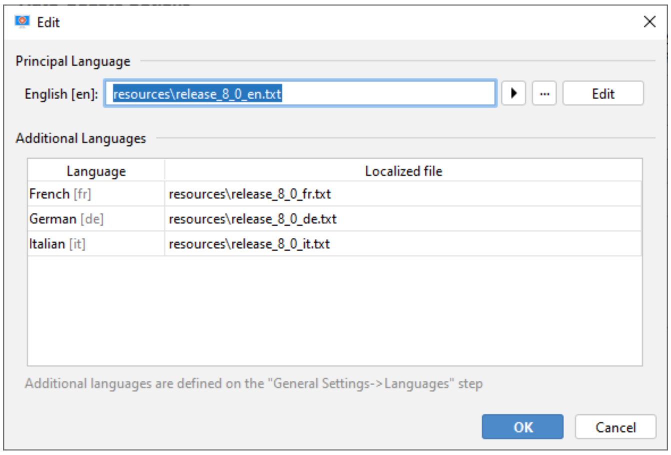
Files with comments: Edit **[No files]**

Additional attributes: Edit **[No attributes]**

Legacy Media Files

☑ IDs of legacy media files: 13,90

In addition to the above attributes, the nested `comment` elements can contain a localized description that should be displayed to the user. You can populate these elements for all media files by configuring the "Files with comments" setting in the "Installer->Auto-Update Options" step. The main use case for this feature is to display release notes in the update downloader.

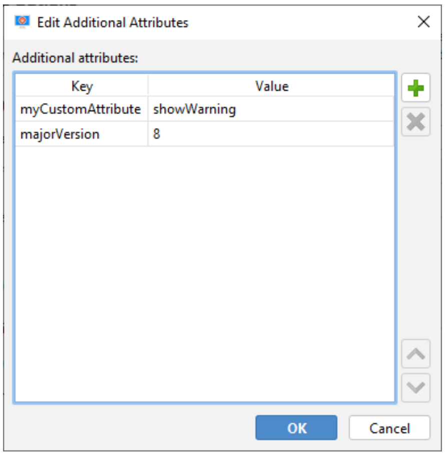


English [en]: resources\release_8_0_en.txt

Language	Localized file
French [fr]	resources\release_8_0_fr.txt
German [de]	resources\release_8_0_de.txt
Italian [it]	resources\release_8_0_it.txt

Additional languages are defined on the "General Settings->Languages" step

Finally, you can add any number of arbitrary attributes to the `entry` element. This is configured with the "Additional attributes" setting in the "Installer->Auto-Update Options" step. Additional attributes are useful for custom logic to select a suitable update installer in the update downloader.



Key	Value
myCustomAttribute	showWarning
majorVersion	8

The update descriptor API and up-to-date checks

The install4j runtime API [p. 212] contains the `com.install4j.api.update.UpdateChecker` utility class that can download the `updates.xml` file and translate it to an instance of `com.install4j.api.update.UpdateDescriptor`. From there, you can get a suitable `com.install4j.api.update.UpdateDescriptorEntry` with a single method call. See the Javadoc for more detailed information.

This API is primarily intended for use in your application. The "hello" sample project shows how to use it in a complex example, see the source file `hello/gui/HelloGui.java` in your install4j installation and look for the `checkForUpdateWithApi` method.

In a custom installer application, you would rather use a "Check for update" action that performs the same actions as `UpdateChecker` and saves the downloaded `UpdateDescriptor` to an installer variable. All update downloader templates included with install4j execute the "Check for update" action at some point. Its URL is set to `${installer:updatesUrl?:${compiler:sys.updatesUrl}}` by default. If you start the update downloader with the argument `-VupdatesUrl=<URL>`, it will define the installer variable "updatesUrl" and that value will be used as the URL. Otherwise it falls back to the compiler variable "sys.updatesUrl" that contains the URL for `updates.xml` that you have entered on the "Installer->Auto-Update Options" step.

Instances of `UpdateDescriptorEntry` expose all attributes of the corresponding `entry` element in the `updates.xml` file. They also provide access to any additional attributes that were added to the `entry` element so you can implement custom logic to find a suitable update. The most important method of the `UpdateDescriptorEntry` class is the `getUrl()`method that constructs the full URL from which the update installer can be downloaded. If no `baseUrl`has been specified on the `updateDescriptor` root element, the URL starts with the parent directory from which the `updates.xml` file has been downloaded.

Strategy 1: Standalone update downloader

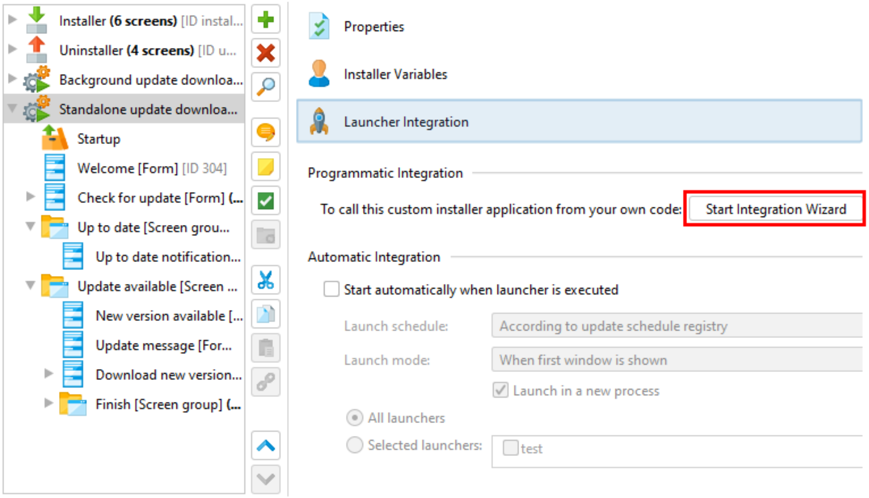
The easiest way to provide auto-update functionality to your users is to create a self-contained update downloader application. This is done by adding an application on the "Installer->Screens & Actions" step [p. 148] and choosing the "Standalone update downloader" application template. Such an auto-updater can by invoked manually by the user. On Windows, it can also be added to the start menu. No changes in in your application code are required so far.

If you have a GUI application, you could provide integration with the update downloader by offering a "Check for update" menu item or similar that invokes the update downloader. One problem in this scenario is that if the updater downloads and executes the update installer, your application will still be running and the user will receive a corresponding warning message in the installer. The solution to this problem is to use the `com.install4j.api.launcher.ApplicationLauncher` class to launch the update downloader. With this utility class, you can launch the update installer by passing its ID as an argument. IDs of installer applications can be shown by toggling the "Show IDs" tool bar button.

If you launch an installer application such as an update downloader that way, the "Shut down calling launcher" action will be able to close your application. To react to the shutdown and perform cleanup operations, you can pass a callback to the `ApplicationLauncher.launchApplication(...)` call. After you are notified through the call back, your application will be terminated with a call to `System.exit()`. For example, for an update downloader with ID 123:

```
import java.io.IOException;
import com.install4j.api.launcher.ApplicationLauncher;

try {
    ApplicationLauncher.launchApplication("123", null, false, new
ApplicationLauncher.Callback() {
            public void exited(int exitValue) {
                //TODO update check complete, no update available
            }

            public void prepareShutdown() {
              //TODO update installer will be executed, perform cleanup before process
 is terminated
            }
        }
    );
} catch (IOException e) {
    e.printStackTrace();
    //TODO handle invocation failure
}
```

To easily get such a code snippet for invoking the update downloader, select the update downloader application and click on the *Start Integration Wizard* button on the right.


Installer (6 screens) [ID instal...
Uninstaller (4 screens) [ID u...
Background update downloa...
Standalone update downloa...
Startup
Welcome [Form] [ID 304]
Check for update [Form] (...
Up to date [Screen grou...
Up to date notification...
Update available [Screen ...
New version available [...
Update message [For...
Download new version...
Finish [Screen group] (...
Properties
Installer Variables
Launcher Integration
Programmatic Integration
Start Integration Wizard
Automatic Integration

To call this custom installer application from your own code:

☐ Start automatically when launcher is executed

Launch schedule: According to update schedule registry

Launch mode: When first window is shown

☑ Launch in a new process

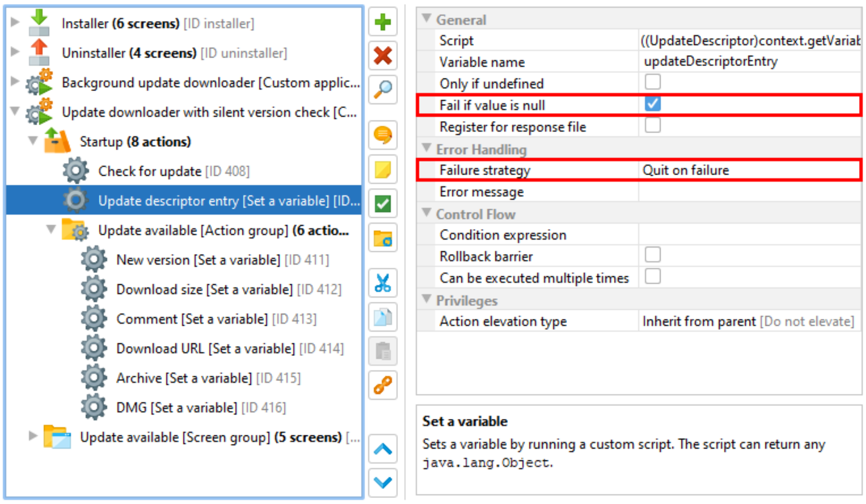
● All launchers

○ Selected launchers: ☐ test

Strategy 2: Update downloader with silent version check

In this scenario, you invoke the update downloader like in strategy 1, but rather than offering a "Check for update" menu item, you do so on a regular schedule. For example, you automatically check for updates every week or each time the user starts the application. In that case, the standalone update downloader template is not suitable because you only want to give the user feedback if a new version is actually available. However, the standalone update downloader always starts with a "Welcome" screen, verbosely checks for updates and informs the user that no new version is available. This behavior is only appropriate if the user explicitly requested an update check.

The "Update downloader with silent version check" application template is intended for automatic update checks. It looks for an update in the startup sequence and terminates the update downloader if no new version is available. This means that if there is no new version available, your users will not see that a check has taken place. Only if a new version is available will the update downloader display its window and inform the user of the possibility to download the update installer.


Installer (6 screens) [ID installer]
Uninstaller (4 screens) [ID uninstaller]
Background update downloader [Custom applic...
Update downloader with silent version check [C...
Startup (8 actions)
Check for update [ID 408]
Update descriptor entry [Set a variable] [ID...
Update available [Action group] (6 actio...
New version [Set a variable] [ID 411]
Download size [Set a variable] [ID 412]
Comment [Set a variable] [ID 413]
Download URL [Set a variable] [ID 414]
Archive [Set a variable] [ID 415]
DMG [Set a variable] [ID 416]
Update available [Screen group] (5 screens) [...

General	
Script	((UpdateDescriptor)context.getVaria
Variable name	updateDescriptorEntry
Only if undefined	☐
Fail if value is null	☑
Register for response file	☐
Error Handling	
Failure strategy	Quit on failure
Error message	
Control Flow	
Condition expression	
Rollback barrier	☐
Can be executed multiple times	☐
Privileges	
Action elevation type	Inherit from parent [Do not elevate]

Set a variable

Sets a variable by running a custom script. The script can return any `java.lang.Object`.

For such an automatic check you will likely want to invoke the update downloader in a blocking fashion. If you call `ApplicationLauncher.launchApplication(...)` with the `blocking` argument set to `true`, the method will not return until the update installer has exited. If the user decides to run the installer on the "Finish" screen, your application will terminate as explained in strategy 1.

Strategy 3: Update downloader without version check

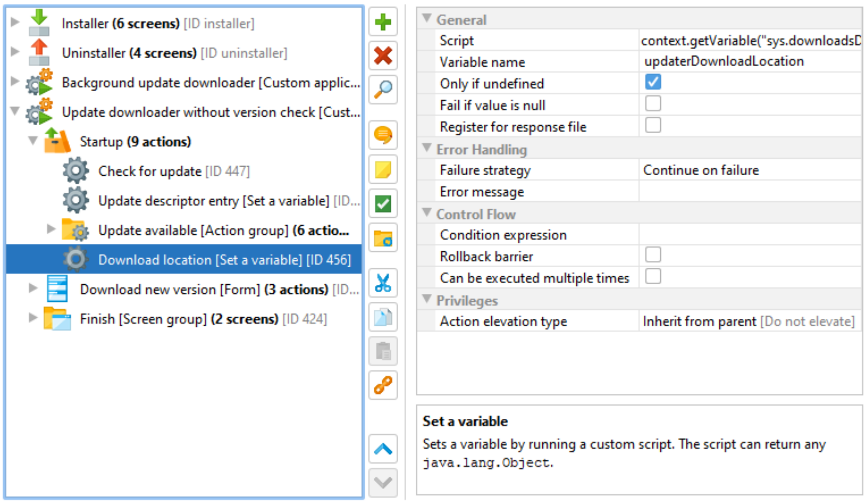
If you want to take the integration one step further and display the availability of a new version in your application yourself, you can use the `com.install4j.api.update.UpdateChecker` class in your code:

```
import com.install4j.api.launcher.Variables;
import com.install4j.api.update.*;

String updateUrl = Variables.getCompilerVariable("sys.updatesUrl");
UpdateDescriptor updateDescriptor = UpdateChecker.getUpdateDescriptor(updateUrl,
ApplicationDisplayMode.GUI);
if (updateDescriptor.getPossibleUpdateEntry() != null) {
    // TODO an update is available, execute update downloader
}
```

In this way, you can display your own notification that announces the new version and lets the user decide whether to download it or not. The "hello" sample project shows how this is done.

If the user decides to download, the "Update downloader with silent version check" template is not suitable because it informs the user about the new version once more. Instead, you should use the "Update downloader without version check" application template. It immediately starts downloading the new version and then proceeds to the "Finish" screen where the user can decide to start the downloaded installer.

This template does not offer the user a directory selection for the downloaded installer, but downloads to the user-specific download directory by default. You can change this default directory by passing the argument `-DupdaterDownloadLocation=[directory]` to the `ApplicationLauncher.launchApplication(...)` call. Again, the update downloader will terminate your application if the user starts the installer as explained for strategy 1.



Installer **(6 screens)** [ID installer]
Uninstaller **(4 screens)** [ID uninstaller]
Background update downloader [Custom applic...
Update downloader without version check [Cust...
  Startup **(9 actions)**
    Check for update [ID 447]
    Update descriptor entry [Set a variable] [ID...
    Update available [Action group] **(6 actio...**
    Download location [Set a variable] [ID 456]
  Download new version [Form] **(3 actions)** [ID...
  Finish [Screen group] **(2 screens)** [ID 424]

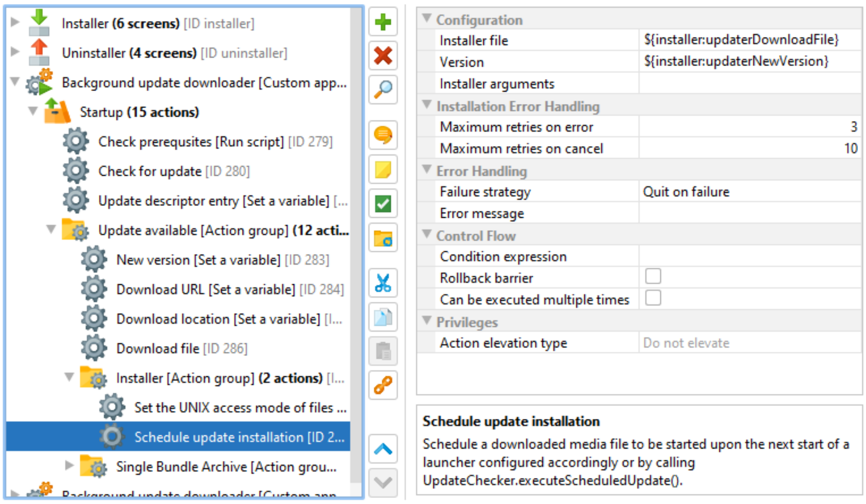
General	
Script	context.getVariable("sys.downloadsD
Variable name	updaterDownloadLocation
Only if undefined	☑
Fail if value is null	☐
Register for response file	☐
Error Handling	
Failure strategy	Continue on failure
Error message	
Control Flow	
Condition expression	
Rollback barrier	☐
Can be executed multiple times	☐
Privileges	
Action elevation type	Inherit from parent [Do not elevate]

Set a variable

Sets a variable by running a custom script. The script can return any `java.lang.Object`.

Strategy 4: Background update downloader

A background update downloader has no UI, and automatically downloads an update installer if available. It will not execute the downloaded update installer because that would disrupt the work of the user. Instead, it executes a "Schedule update installation" action to register the downloaded update installer for later execution.


Installer (6 screens) [ID installer]
Uninstaller (4 screens) [ID uninstaller]
Background update downloader [Custom app...
Startup (15 actions)
Check prerequsites [Run script] [ID 279]
Check for update [ID 280]
Update descriptor entry [Set a variable] [...
Update available [Action group] (12 acti...
New version [Set a variable] [ID 283]
Download URL [Set a variable] [ID 284]
Download location [Set a variable] [I...
Download file [ID 286]
Installer [Action group] (2 actions) [I...
Set the UNIX access mode of files ...
Schedule update installation [ID 2...
Single Bundle Archive [Action grou...

Configuration	
Installer file	${installer:updaterDownloadFile}
Version	${installer:updaterNewVersion}
Installer arguments	
Installation Error Handling	
Maximum retries on error	3
Maximum retries on cancel	10
Error Handling	
Failure strategy	Quit on failure
Error message	
Control Flow	
Condition expression	
Rollback barrier	☐
Can be executed multiple times	☐
Privileges	
Action elevation type	Do not elevate

Schedule update installation

Schedule a downloaded media file to be started upon the next start of a launcher configured accordingly or by calling UpdateChecker.executeScheduledUpdate().

There are two options to execute an update installer that is scheduled for execution:

- **Programmatic invocation**

  By calling

  ```
  com.install4j.api.update.UpdateChecker.executeScheduledUpdate(...);
  ```

  you can execute the downloaded update installer programatically, usually after checking the result of

  ```
  com.install4j.api.update.UpdateChecker.isUpdateScheduled()
  ```

  to determine whether such a download has been completed. You can do that while the launcher is running or at startup. Notifying the user about this event or letting the user defer the installation is handled by your own code. For GUI and server launchers, this is the only option.

  The "HelloGui" class the in the "hello" sample contains a complete demonstration of how to use the API to check for updates programatically and uses a background update downloader to download and install updates.

- **Automatic invocation**

  For GUI launchers, you can edit the launcher, go to the "Executable info->Auto-update integration" step and select the `Execute downloaded update installers at startup` check box. When the GUI installer is started and a downloaded update installer has been scheduled for installation, the update installer will be executed. See the help topic on launchers [p. 36] for more information.

Update schedule registry

A common requirement is to check for an update on a regular schedule. install4j comes with a standard implementation of an update schedule registry that frees you of the task to implement one yourself. This update schedule registry is fully integrated with the launcher integration that starts update downloaders when launchers are executed, but it is also available in the API.

The `com.install4j.api.update.UpdateScheduleRegistry` class is intended to be used in your application. You configure a particular `UpdateSchedule` by calling

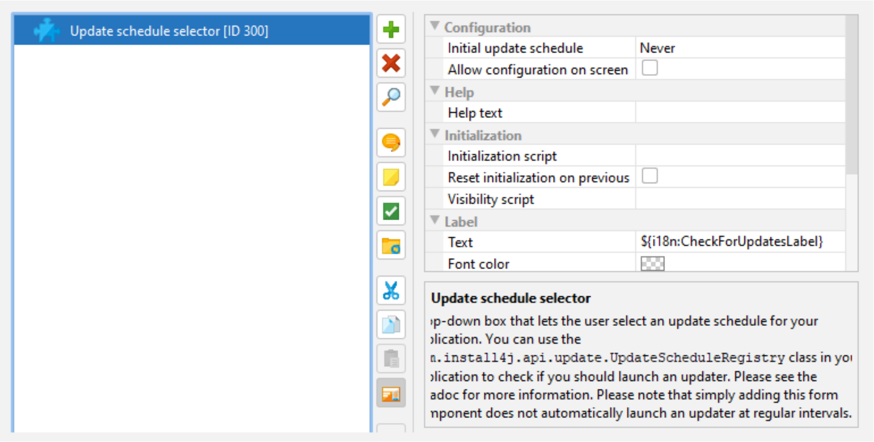
```
import com.install4j.api.update.*;

UpdateScheduleRegistry.setUpdateSchedule(UpdateSchedule.DAILY);
```

and call

```
boolean shouldCheckForUpdate = UpdateScheduleRegistry.checkAndReset();
```

each time your application is started. If you get a positive response, you can start a suitable update downloader with the `ApplicationLauncher` class as explained above.

To facilitate the configuration of the update schedule in your installer, install4j offers a special "Update schedule selector" form component whose initial value is set to the current setting (if any) and automatically updates the setting for the installed application when the user clicks "Next".



Configuration	
Initial update schedule	Never
Allow configuration on screen	☐
Help	
Help text	
Initialization	
Initialization script	
Reset initialization on previous	☐
Visibility script	
Label	
Text	${i18n:CheckForUpdatesLabel}
Font color	▨

Update schedule selector

ɪp-down box that lets the user select an update schedule for your ɔlication. You can use the n.install4j.api.update.UpdateScheduleRegistry class in yo ɔlication to check if you should launch an updater. Please see the adoc for more information. Please note that simply adding this form nponent does not automatically launch an updater at regular intervals.

A.20 Version Numbers

Version numbers in install4j should be a sequence of version components separated by dots:

```
A.B.C...
```

where A, B, C are composed of alphanumeric characters without dots, for example `1`, `112`, `5-rc-9` or `release`.

Version comparisons in the auto-update API

The auto-update [p. 114] API in the `com.install4j.api.update` package has to determine whether a new version is greater than an installed version or not. Usually, the `getPossibleUpdateEntry()` method of the update descriptor is called to make that comparison:

```
UpdateDescriptor updateDescriptor = ...;
if (updateDescriptor.getPossibleUpdateEntry() != null) {
    //TODO an update is available
}
```

In its implementation, it calls

```
UpdateDescriptorEntry updateDescriptorEntry = ...;
String installedVersion = context.getVersion();
if (updateDescriptorEntry.checkVersionCompatible(installedVersion)) {
  // TODO This entry has a version that is newer than the installed version
}
```

The `checkVersionCompatible` method checks if the supplied version

- is greater or equal than the minimum updatable version in the update descriptor entry (if defined)
- is less or equal than the maximum updatable version in the update descriptor entry (if defined)
- is less than the version of the new media file

Internally, it calls

```
if (UpdateChecker.isVersionGreaterThan(newVersion, installedVersion)) {
    // TODO newVersion is indeed greater than installedVersion
}
```

to compare the version strings of the installed version with the new version in the update descriptor entry.

Comparision algorithm for versions

Let us call the two versions that should be compared A and B. A has N_A components while B has N_B components. Components are determined by splitting the version string with a `java.util.StringTokenizer` and a single dot as a delimiter. The components are denoted as `A(0) ...` `A(`N_A`-1)` and `B(0) ... B(`N_B`-1)`.

The following rules apply when comparing these two versions:

1. Before the comparison, the following replacements are performed for both versions in this order:

   - When going from left to right, a boundary between digits and non-digits creates a new component, for example `2.3a` becomes `2.3.a`. Boundaries between non-digits and digits are left intact, for example `2.3.a4`. This means that non-numeric characters only appear as leading characters for each component.
   - After dots, any "-" and "_" characters are discarded.
   - All characters are converted to lower-case, for example `1.0-HEAD` becomes `1.0.head`.

2. The version that has less components is filled up with components of value `0`, so that both versions have the same number of components `N = max(`N_A`, `N_B`)`.
3. The versions are compared from left to right, component by component. The version comparison is finished for the first `K = 0 ... N-1` for which the components are not equal:

   ```
   B(K) > A(K) => B > A
   B(K) < A(K) => B < A
   ```

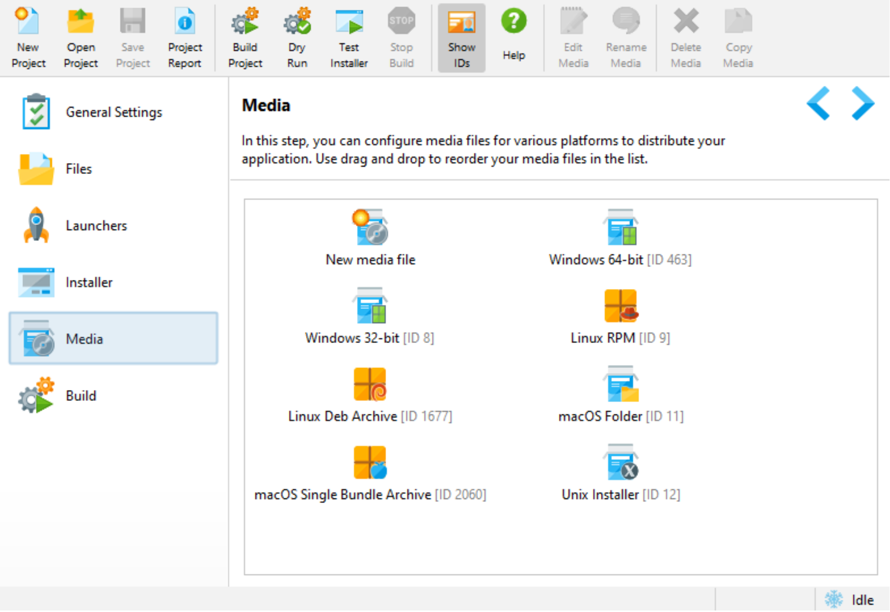
4. Components that have leading non-numeric characters are considered as less than components with leading numeric characters. For example `2.3-pre < 2.3`, because `2.3-pre` is converted to `2.3.pre` and `2.3` is converted to `2.3.0`.
5. If both components have a non-numeric part, version comparison is decided by their lexicographic comparison, as performed by `String.compareTo(...)`. For example, `2.Z3 > 2.X4`. If the non-numeric parts are equal, the numeric parts are compared where missing numeric parts are set to `0`.
6. Otherwise the components are both numeric and can be compared numerically.

Some examples from the unit test for the version comparison method are:

```
1 < 2
1.1 < 2
1.1 < 1.2
1.1.0 < 1.1.1
9.0 < 10
1.6.0_4 < 1.6.0_22
1.6.0 < 1.6.0_22
1.6.0_4 < 1.6.1
1.0beta1 < 1.0beta2
1.0.beta1 < 1.0.beta2
A10 < A11
2.0 beta 1 < 2.0
2.0 beta 1 < 2.0.0
2.0 beta 1 < 2.0 beta 2
1.0rc1 < 1.0
1.0-rc1 < 1.0
1.0.rc1 < 1.0
1.0alpha < 1.0rc1
1.0alpha < 1.0alpha1
1.0alpha9 < 1.0alpha10
1.0alpha100 < 1.0.rc100
1.0.alpha < 1.0-rc
z < 1
DEVELOP-HEAD130714193704 < DEVELOP-HEAD130714193705
```

A.21 Media Files

Media files are the final output of install4j: single artifacts that are used to distribute your application to your users. The creation of a media file has platform-dependent options, so for each platform, you have to define a separate media file. It also makes sense to define several media files for one platform in case you wish to distribute different subsets of your distribution tree, or if you distribute your application with and without a bundled JRE.

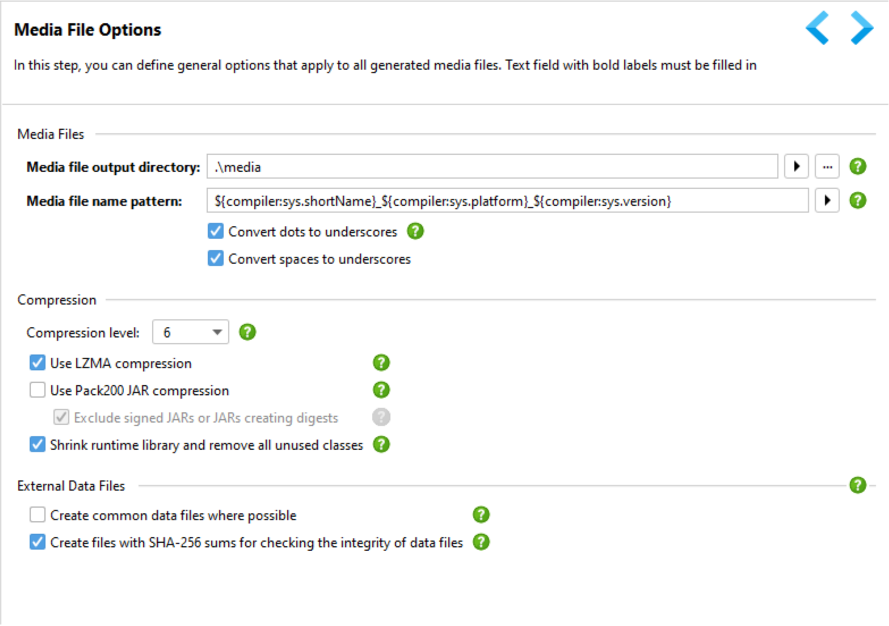


Media

In this step, you can configure media files for various platforms to distribute your application. Use drag and drop to reorder your media files in the list.

New media file
Windows 64-bit [ID 463]
Windows 32-bit [ID 8]
Linux RPM [ID 9]
Linux Deb Archive [ID 1677]
macOS Folder [ID 11]
macOS Single Bundle Archive [ID 2060]
Unix Installer [ID 12]

Common options for all media files, such as the destination directory, a pattern for naming the output file and compression options are defined on the "General Settings->Media File Options" step.



Media File Options

In this step, you can define general options that apply to all generated media files. Text field with bold labels must be filled in

Media Files

Media file output directory: .\media

Media file name pattern: ${compiler:sys.shortName}_${compiler:sys.platform}_${compiler:sys.version}

☑ Convert dots to underscores

☑ Convert spaces to underscores

Compression

Compression level: 6

☑ Use LZMA compression

☐ Use Pack200 JAR compression

☑ Exclude signed JARs or JARs creating digests

☑ Shrink runtime library and remove all unused classes

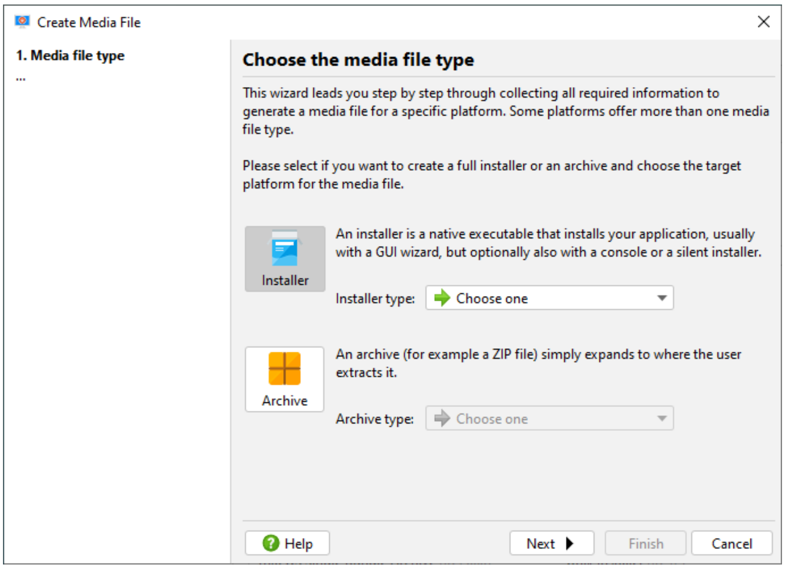
External Data Files

☐ Create common data files where possible

☑ Create files with SHA-256 sums for checking the integrity of data files

Media files have names and IDs. The name is available elsewhere by using the `sys.mediaName` compiler variable but is otherwise not used by the compiler. IDs of media files can be used for selecting media files when building the project from the command line [p. 220]. You can show IDs by toggling the "Show IDs" tool bar button.

There are two fundamentally different types of media files: installers and archives. Installers support the full functionality of install4j while archives are limited in several ways.



Choose the media file type

This wizard leads you step by step through collecting all required information to generate a media file for a specific platform. Some platforms offer more than one media file type.

Please select if you want to create a full installer or an archive and choose the target platform for the media file.

An installer is a native executable that installs your application, usually with a GUI wizard, but optionally also with a console or a silent installer.

Installer type: Choose one

An archive (for example a ZIP file) simply expands to where the user extracts it.

Archive type: Choose one

Installers

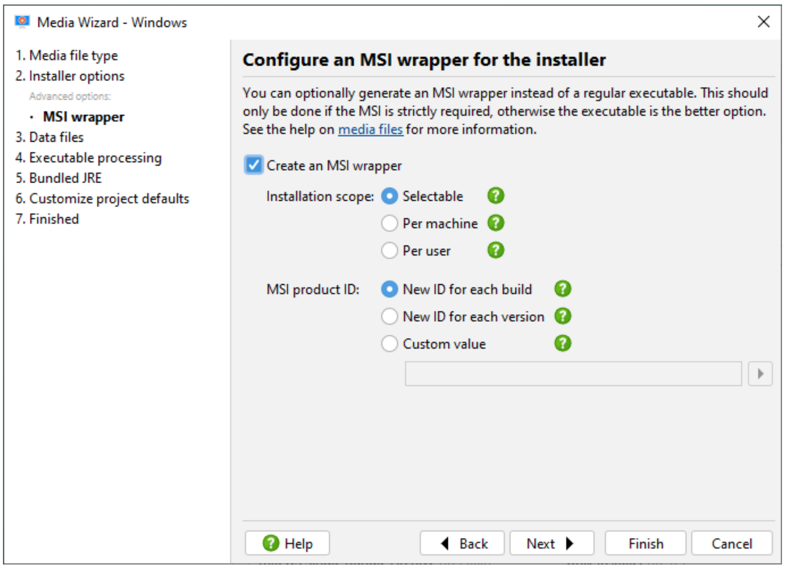
Installers install your application programmatically with the configured sequence of screens & actions [p. 24]. Optionally, an installer can be executed in unattended or in console mode [p. 195] and it can download a JRE [p. 89] if no suitable JRE is found on the target system.

The following installer media file types are available:

- **Windows**

  A media file for Windows is a native setup executable that installs your application with an installer wizard.

  Optionally, you can create an MSI wrapper instead of a regular executable. This is configured on the "MSI wrapper" advanced options step below the "Installation options". It is not recommended to use the MSI wrapper without having a specific requirement for it. The MSI wrapper adds a lot of extra process machinery and additional logic to bridge mismatches between the concepts of install4j and MSI. This results in additional overhead, increased temporary disk space requirements, reduced responsiveness and extra considerations for the non-GUI installer modes.



Configure an MSI wrapper for the installer

You can optionally generate an MSI wrapper instead of a regular executable. This should only be done if the MSI is strictly required, otherwise the executable is the better option. See the help on media files for more information.

☑ Create an MSI wrapper

Installation scope: ● Selectable ○ Per machine ○ Per user

MSI product ID: ● New ID for each build ○ New ID for each version ○ Custom value

MSI usually has a fixed setting for whether an installation will be performed per-machine or per-user. In install4j, this corresponds to whether the "Request privileges" action is performed or not. If you execute the "Request privileges" action conditionally, you have to use the default "Selectable" MSI installation scope. However, for unattended installations, the user then has to specify the command line options

```
ALLUSERS=1
```

and start the msi installation with elevated privileges for a per-machine installation, otherwise the installation will be per-user.

To avoid this extra parameter, install4j offers the "per-machine" MSI installation scope where this extra command line option is not required for unattended installations. It is your responsibility to ensure that the "Request privileges" action is always executed for the "per-machine" installation scope and is never executed for the "per-user" installation scope.

If the scope is not selectable, MSI will also prevent that an installation is repeated if it has already been performed. The identity of an installation is defined by the MSI product ID. If an installation with the same product ID is found, the MSI installer will show an error message and terminate. By default, install4j creates a unique MSI product ID for each build. You can also tell install4j to create a new product ID for each application version as configured on the "General Settings->Application Info" step, or to use a custom MSI product ID that you can change as required.

To change the installation directory, the variable `INSTALLDIR` can be specified on the command line. In addition, `PARAMETER` can be used to pass arbitrary command line parameters to the wrapped installer.

- **macOS single bundle [deprecated]**

  This media file type is deprecated because of signature requirements in modern macOS versions. Use the single bundle archive instead and configure a setup application in the media wizard that is run the first time the user starts the application.

  A single bundle media file for macOS is a DMG file that contains an installer wizard that is started by double clicking on it in the Finder. The wizard installs your application as a single application bundle for a selected GUI launcher. Command line launchers and service launchers are contained in the application bundle. If you wish to support multiple GUI launchers, choose the "macOS folder media wizard" instead.

All files in the distribution tree will be installed inside the application bundle under the relative path `Contents/Resources/app`. The full path of that directory is exposed by the installer variable `sys.contentDir` at runtime. To install files to other directories, add an installation root [p. 14] to the distribution directory, for example with the name

```
${installer:sys.installationDir}/My Application Documents
```

to create a folder "My Application Documents" next to the installed application bundle.

The main drawback of this media file type is that the installer application bundle is not signed. Signing an application bundle has to be done at compile-time. With an installer, the exact contents of the installation are not known at compile-time. The installer itself will be signed, but if you need app entitlements (1) that can be set on the "Executable info->macOS options" step of the launcher wizard, a signature of the installed application bundle is required. Use the single bundle archive in that case.

- **macOS folder**

  Like the single bundle installer, the folder media file for macOS is started by the user from the Finder after opening the DMG. The wizard installs your application as a folder that contains the entire distribution tree and multiple application bundles for each included GUI launcher.

- **Unix/Linux GUI installer**

  A Unix/Linux GUI installer media file is an executable shell script that extracts an installer and installs your application with an installer wizard.

Archives

Archives can be extracted by the user to arbitrary locations or are submitted to package managers for installation. No screens are shown and no actions are executed. If you define additional installation roots, the files in them are not installed. Also, no installation components are downloaded.

Apart from the "macOS single bundle" archive that produces the idiomatic deployment mode for GUI applications on macOS, archives are mainly intended as a fallback or for additional packages such as documentation bundles.

When a launcher is executed for the first time after an extraction, you can call a custom installer application to perform tasks that would otherwise have been part of the installer. With the `ApplicationLauncher.isNewArchiveInstallation()` method you can find out whether this is the case:

```
import com.install4j.api.launcher.*;

if (ApplicationLauncher.isNewArchiveInstallation()) {
    ApplicationLauncher.launchApplication("123", null, true, null);
}
```

where "123" is the ID of the custom installer application that should be run.

The following archive media file types are available:

---

(1) https://developer.apple.com/documentation/security

- **Windows archive**

  An archive media file for Windows is a ZIP-file that contains your application.

- **macOS single bundle archive**

  A single bundle media file for macOS is a DMG or .tgz archive that contains a single application bundle for a selected GUI launcher. Command line launchers and service launchers are contained in the application bundle. If you wish to support multiple GUI launchers, choose the "macOS folder archive" media file type instead.

  Just like for the single bundle installer, all files in the distribution tree are contained inside the application bundle under the relative path `Contents/Resources/app`.

  This is the preferred way to distribute a GUI application on macOS. The corresponding installer that installs a single application bundle is deprecated because of signature requirements of modern macOS versions. To make it easier to use the screen and action system in install4j for installations, the media wizard allows you to select a custom installer application that is executed the first time the user starts the application bundle.

- **macOS folder archive**

  A folder media file for macOS is a DMG or .tgz archive that contains the distribution tree and multiple application bundles for each included GUI launcher.

- **Unix/Linux archive**

  A Unix/Linux archive media file is a gzipped TAR archive that contains your application. Users will extract them with a command like

  ```
  tar xzf archive.tar.gz
  ```

- **Linux RPM**

  An RPM archive for Linux can be installed and uninstalled with the `rpm` command on Linux distributions that use the Redhat package management.

  A basic installation command looks like

  ```
  rpm -i archive.rpm
  ```

  You can configure custom installer applications to run in the post-installation phase and the pre-uninstallation phase. Alternatively, default actions for installed launchers can be performed without starting a JVM. These include the installation of services, creating links for non-service launchers in `/usr/local/bin` and integrating GUI launchers into the menu of the desktop environment. In addition, bash scripts for pre-install, post-install, pre-uninstall and post-uninstall phases can be configured.

- **Linux Deb**

  A Deb archive for Linux can be installed and uninstalled with the `dpkg` command on Linux distributions that use the Debian package management.

  A basic installation command looks like

  ```
  dpkg -i archive.deb
  ```

Deb media files have the same functionality for running custom installer applications as RPM media files.

Customizing project defaults

Many project configuration settings can be overridden for each media file. Settings in text fields can be overridden by using compiler variables [p. 63] and overriding them in the "Customize project defaults->Compiler variables" step of the media wizard.

It is also possible to override compiler variables for specific media files from the command line [p. 220] by prefixing the variable name with the media file ID and a colon, as in

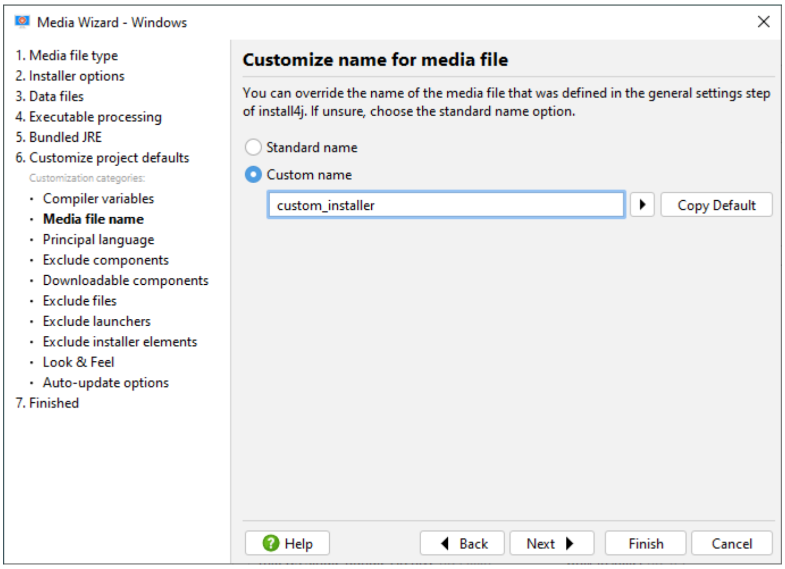
```
-D 123:key=value
```

if the media file ID is "123". As a special case, you can change the principal language on a per-media file basis by setting the compiler variable `sys.languageId` with the 2-letter ISO code (2) of the desired language, for example

```
-D 123:sys.languageId=fr
```

For some features where text fields are not used, special screens are available in the "Customize project defaults" category. They let you exclude files, launchers, installation components and installer elements. In addition, the principal language [p. 79] and auto-update options [p. 114] can be overridden for the media file.

Because it is often necessary to change the name of the media file from the global media file pattern configured on the "General Settings->Media File Options" step, a separate customization step is available in the media wizard. For example, you may want to produce two different variants for the same platform with and without some files. To avoid a name clash of the two media files, you have to adjust the name of one or both of the media files.



Customize name for media file

You can override the name of the media file that was defined in the general settings step of install4j. If unsure, choose the standard name option.

○ Standard name

● Custom name

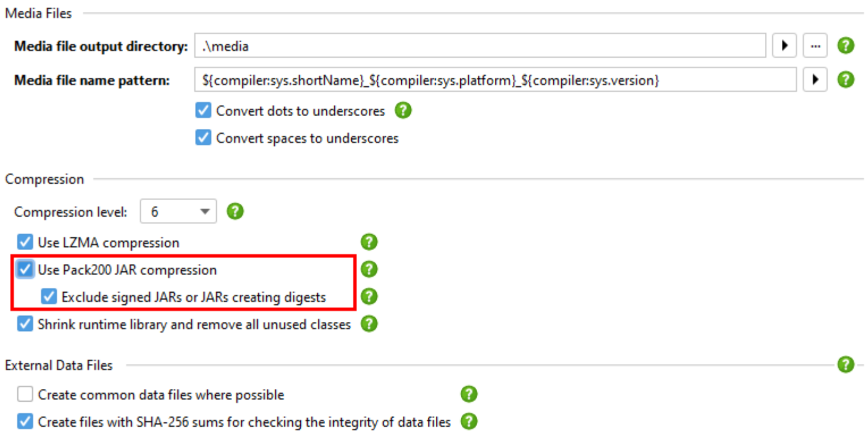
custom_installer

---

(2) https://www.w3.org/WAI/ER/IG/ert/iso639.htm

Pack200 JAR compression

Pack200 compression(3) is a compression algorithm that was designed for JAR files and achieves exceptional results, especially for large JAR files.



Media Files

Media file output directory: .\media

Media file name pattern: ${compiler:sys.shortName}_${compiler:sys.platform}_${compiler:sys.version}

☑ Convert dots to underscores

☑ Convert spaces to underscores

Compression

Compression level: 6

☑ Use LZMA compression

☑ Use Pack200 JAR compression

☑ Exclude signed JARs or JARs creating digests

☑ Shrink runtime library and remove all unused classes

External Data Files

☐ Create common data files where possible

☑ Create files with SHA-256 sums for checking the integrity of data files

If you have signed JAR files or JAR files that create a digest, apply the `$JDK_HOME/bin/pack200` executable in your build process with

```
pack200 --repack my.jar
```

before signing the JAR files. Pack200 rearranges JAR files but the reordering is idempotent, so the above pack/unpack sequence creates a stable JAR file.

While Pack200 compression can be quite slow, Pack200 decompression is relatively fast. Pack200 compression is only used for installers and not for archives.

To avoid problems with external JAR files, you can check the "Exclude signed JARs or JARs creating digests" option. If you would like to exclude selected JAR files only, you can place an empty `*.nopack` file next to it. For example, if the jar file is named `app.jar`, then a file `app.jar.nopack` in the same directory will disable Pack200 compression for that file.

To pass options(4) to the packer, create a file `*.packoptions` next to the file and add one option per line. Currently, only `-P` and `--pass-file` are supported.

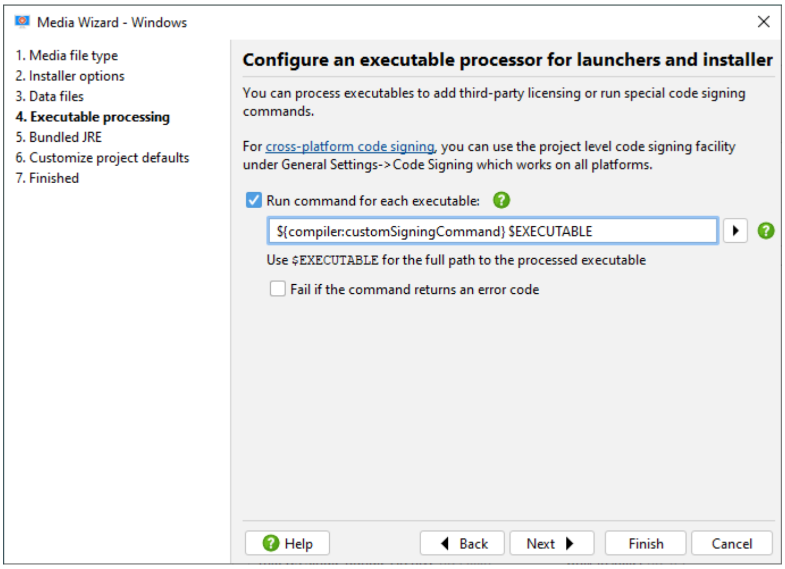
Executable post-processing

The install4j compiler can invoke a post-processor for each executable that is generated. This includes

- generated launchers
- the installer
- the uninstaller
- custom installer applications

(3) http://java.sun.com/j2se/1.5.0/docs/guide/deployment/deployment-guide/pack200.html

(4) http://docs.oracle.com/javase/8/docs/technotes/tools/windows/pack200.html



Configure an executable processor for launchers and installer

You can process executables to add third-party licensing or run special code signing commands.

For cross-platform code signing, you can use the project level code signing facility under General Settings->Code Signing which works on all platforms.

☑ Run command for each executable:

`${compiler:customSigningCommand} $EXECUTABLE`

Use `$EXECUTABLE` for the full path to the processed executable

☐ Fail if the command returns an error code

Typically, this feature is used for custom code signing with special requirements that are not supported by the integrated code signing [p. 138].

In the post processor text field you can use the `$EXECUTABLE` variable to reference the executable that is currently being post-processed. The working directory of the executed process is the directory where your project file is located, so you can use relative file names for key or certificate files. If the signing command cannot replace the executable directly, but rather needs a separate output file, use the `$OUTFILE` variable. It will receive a temporary output file name that will be moved back to the processed executable by install4j after the post processor has completed.

A.22 Data Files

Typically, installers are single files that contain all data that they can install when they are executed. There are three common situations where this is not the case:

- **DVD installers with large data files**

  If your application relies on large amounts of data, it is often distributed on a DVD. In that case, you typically ship a number of external data files that you do not wish to package inside the installer. The installer should start up quickly and the data files should not be extracted from the installer in order to save time. The user might decide to install only certain components, so some data files might not be needed at all. If they are included in the installer executable, all this data would have to be read from disk.

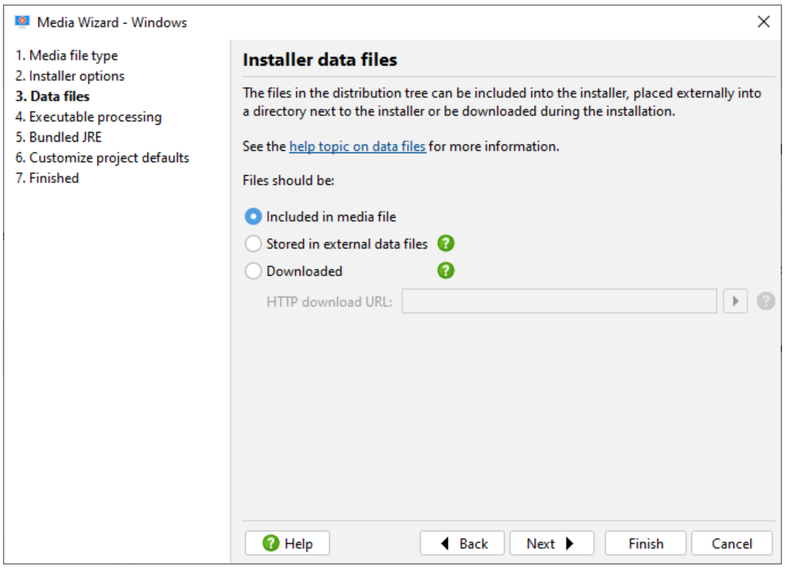
- **Installers with large optional components**

  Some applications have large optional components that are not relevant for the typical user. To reduce download size for the majority, the optional components should be downloadable on demand.

- **Net installers**

  Some applications are highly modular, so it is not feasible to build a set of installers for typical use cases. A net installer lets the user select the desired components and downloads them on demand. The download size of the net installer is small because no parts of the actual application are contained in the installer itself.

To accommodate the above use cases, install4j offers three different ways to handle the installer data files. The data file mode can be selected in the "Data files" step of the media wizard. By default, the "Included in media file" option is selected where all data files are included in the installer so you can ship it as a single download.



Installer data files

The files in the distribution tree can be included into the installer, placed externally into a directory next to the installer or be downloaded during the installation.

See the help topic on data files for more information.

Files should be:

● Included in media file
○ Stored in external data files
○ Downloaded

HTTP download URL:

External data files

This mode covers the "DVD installers with large data files" use case.

Next to your installer, a directory for the data files is created with the name of your installer and the extension `.dat`. For example, if your media file name is `hello_4_0`, resulting in a Windows installer executable `hello_4_0.exe`, the directory containing the external data files is named `hello_4_0.dat`. You have to ship this directory in the same relative location on your DVD.

The number of data files depends on the definition of your installation components. The data files are generated in such a way that

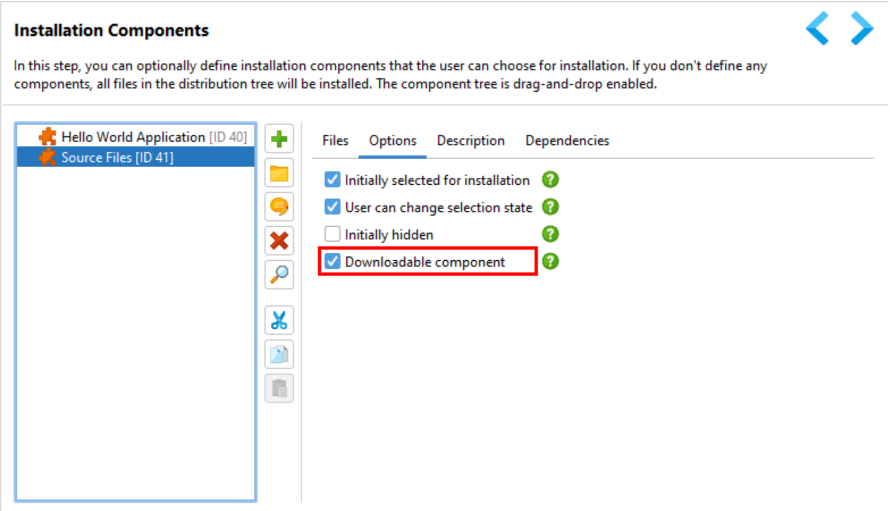
- the files for an installation component are contained in one or more data files
- there are no files in those data files that do not belong to this installation component

If components do not overlap, there's a one-to-one correspondence between data files and installation components.

Downloadable data files

This mode covers the "Installers with large optional components" and "Net installers" use cases. It can only be used if you define installation components [p. 20].

Data files are generated just like for the "External" mode, but only for installation components that have been marked as downloadable in the installation component definition [p. 20].



Installation Components

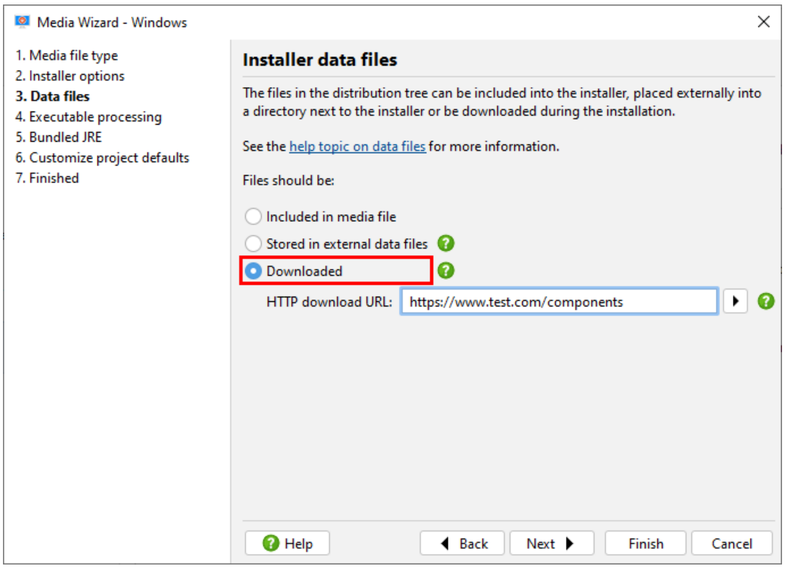
In this step, you can optionally define installation components that the user can choose for installation. If you don't define any components, all files in the distribution tree will be installed. The component tree is drag-and-drop enabled.

Hello World Application [ID 40]
Source Files [ID 41]

☑ Initially selected for installation
☑ User can change selection state
☐ Initially hidden
☑ Downloadable component

If no installation components are marked as "downloadable", this mode will behave like the "Included in media file" mode. For a "net installer", all installation components should be "downloadable".

For this mode, you have to enter a **HTTP download URL**, so the installer knows from where it should download the data files at runtime if the user requests downloadable components. The URL must begin with `http://` or `https://` and point to a directory where you place the data files that the compiler produces in the `.dat` folder next to the installer.



Installer data files

The files in the distribution tree can be included into the installer, placed externally into a directory next to the installer or be downloaded during the installation.

See the help topic on data files for more information.

Files should be:

○ Included in media file

○ Stored in external data files

● Downloaded

HTTP download URL: https://www.test.com/components

The build output will list the data files that belong to downloadable installation components with a message like

```
Important: Please make sure that the following files can be downloaded from

     https://www.test.com/components

     hello_windows-x64_8_0.41.dat
```

This means that the data file must be uploaded to the web server, so that the installer can download it from the URL

```
https://www.test.com/components/hello_windows-x64_8_0.41.dat
```

Any data files that you leave in the data file directory next to the installer will not be downloaded. This means that if you test your installer directory from the location where it was generated, the installer finds all data files in the data file directory and does not try to download them.

Naming and partitioning of data files

The naming of data files is stable and only depends on the name of the media file and the downloadable installation components.

For example, say your installer includes the following 7 files:

```
file_1.txt
file_2.txt
file_3.txt
file_12.txt
file_13.txt
file_23.txt
file_123.txt
```

and there are three installation components with IDs 1, 2 and 3 that include the following files:

```
Component 1:
  file_1.txt
  file_12.txt
  file_13.txt
  file_123.txt
Component 2:
  file_2.txt
  file_12.txt
  file_23.txt
  file_123.txt
Component 3:
  file_3.txt
  file_13.txt
  file_23.txt
  file_123.txt
```

Note that some files are in multiple components, and in the above scheme each component includes all files whose number contains the ID of the installation component.

If the media file is named `test`, the compiler then produces one data file per component named `test.X.dat` with the files that are included exclusively by the corresponding component:

```
test.1.dat
  file_1.txt
test.2.dat
  file_2.txt
test.3.dat
  file_3.txt
```

Next, data files named `test.X.Y.dat` for the files that are included in exactly two components are generated:

```
test.1.2.dat
  file_12.txt
test.1.3.dat
  file_13.txt
test.2.3.dat
  file_23.txt
```

Finally, a data file is generated that includes files that appear in all three components:

```
test.1.2.3.dat
  file_123.txt
```

When generalizing this partitioning to N installation components, a maximum number of $2^N - 1$ data files is created. In practice, it is more likely that each installation component only has exclusive files and that there will be `N` data files.

For the downloadable data file mode, only the downloadable installation components are included in this partition. Files that belong to other installation components are included in the installer and do not play any role in the creation of data files.

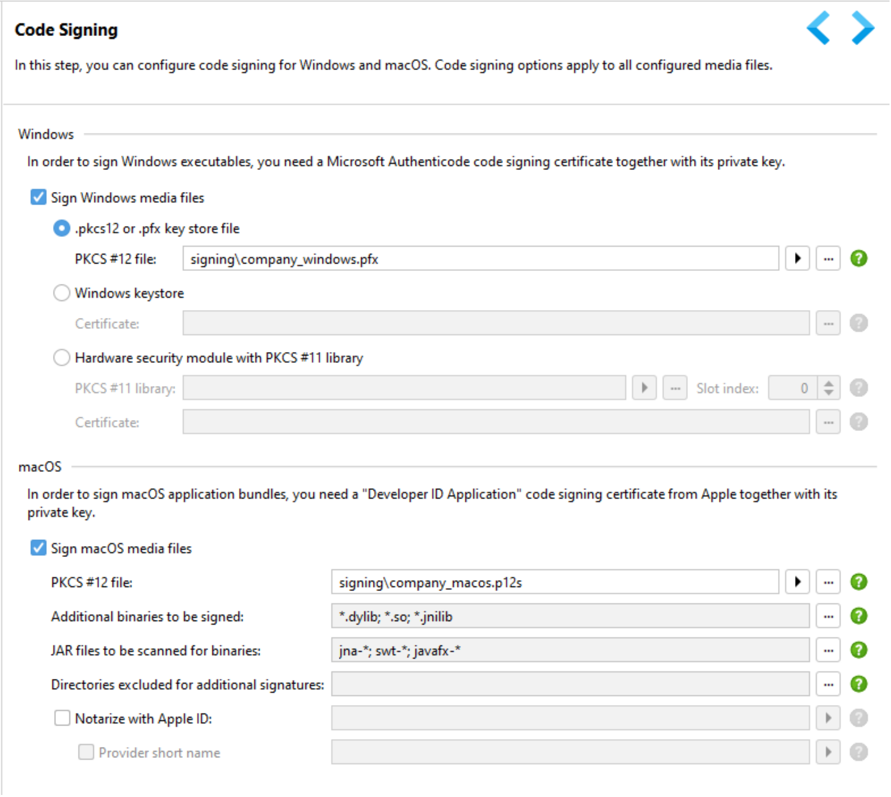
A.23 Code Signing

Code signing ensures that the installer, uninstaller and launchers can be traced back to a particular vendor. A third party certificate authority guarantees that the signing organization is known to them and has been checked to some extent. The certificate authority has the ability to revoke a certificate in case it gets compromised.

The basis for code signing is a public and private key pair (1) that you generate on your computer. The private key is only known to yourself and you never give it to anyone else. The certificate provider takes your public key and signs it with its own private key. That key in turn is validated by an official root certificate that is known to the operating system. The private key, the public key and the certificate chain provided by the certificate provider are all required for code signing.

Code signing is important for installers on Windows and macOS. For unsigned applications that require admin privileges, Window will display special warning dialogs to alert the user that the application is untrusted and may harm the computer. Also, the SmartScreen (2) filter will make it very difficult for the user to execute unsigned executables.

On macOS, the Gatekeeper (3) prevents non-expert users from installing an unsigned application that was marked as downloaded from the internet, so code signing is practically required.



Code Signing

In this step, you can configure code signing for Windows and macOS. Code signing options apply to all configured media files.

Windows

In order to sign Windows executables, you need a Microsoft Authenticode code signing certificate together with its private key.

☑ Sign Windows media files

● .pkcs12 or .pfx key store file

PKCS #12 file: signing\company_windows.pfx

○ Windows keystore

Certificate:

○ Hardware security module with PKCS #11 library

PKCS #11 library: Slot index: 0

Certificate:

macOS

In order to sign macOS application bundles, you need a "Developer ID Application" code signing certificate from Apple together with its private key.

☑ Sign macOS media files

PKCS #12 file: signing\company_macos.p12s

Additional binaries to be signed: *.dylib; *.so; *.jnilib

JAR files to be scanned for binaries: jna-*; swt-*; javafx-*

Directories excluded for additional signatures:

☐ Notarize with Apple ID:

☐ Provider short name

You need different certificates for code signing on Windows and macOS. While it is technically possibly to use the same certificate, the recognized root certificates are different on both platforms.

(1) https://en.wikipedia.org/wiki/Public-key_cryptography
(2) https://en.wikipedia.org/wiki/Microsoft_SmartScreen
(3) https://en.wikipedia.org/wiki/Gatekeeper_(macOS)

Code signing for Windows

You can purchase a "Microsoft Authenticode" code signing certificate from a certificate provider such as DigiCert (4). In this process, you will create a public and private key pair on your computer as instructed by your certificate provider. You send them the public key and receive a certificate in PKCS #12 format (5) that can be used by install4j.

If you have private key, public key and certificate chain in some other format, you can use openssl (6) to convert them to a PKCS #12 file.

Depending on how you generate the private key while applying for a code signing certificate, the private key is located in the Windows keystore and the generated certificate is imported into the keystore as well. In that case, you can use the "Windows keystore" option in install4j and select the certificate from a list of available certificates. This is only interesting if your build runs on Windows, on other platforms this option is not visible in the install4j IDE unless it is already selected in the project.

EV-certificates (7) get preferential treatment by the Windows SmartScreen filter, but they require that the private key resides on a hardware security module (HSM), so that it cannot be copied. On Windows, such a hardware token can be usually accessed through the Windows keystore. On a different platform, you have to choose the "Hardware security module PKCS #11 library" option and configure a native library that provides access to the keystore in the HSM through the PKCS #11 API (8). Libraries can access multiple HSMs that are said to be in different "slots". By adjusting the slot index, you can switch to a different HSM. By default, the first available HSM in slot 0 is used. After the library has been configured, a certificate can be chosen from the keystore in the HSM. Even if you have just one code signing certificate, over time you will likely add certificate renewals to the same HSM.

Code signing for macOS

Certificates for code signing are only issued by Apple. To get started, open the Keychain Access app and select *Keychain Access->Certificate Assistant->Request a Certificate From a Certificate Authority*. The assistant will save a `certSigningRequest` file to your file system.

Then, log in to the Apple Developer Network (9) and request a "Developer ID Application" (10) macOS code signing certificate. Download the certificate and double-click to add it to the Keychain.

Finally, open the Keychain Access app, select the "Keys" category and export the key that belongs to your "Developer ID Application" certificate by selecting both the certificate as well as the private key and right clicking on the combined selection. Choose `.p12` as the file format. The keychain tool will ask you for a new password for the exported file. This is the password you will have to specify during the install4j build to access your key.

install4j will refuse to use certificates for code signing that have a certificate subject name other than "Developer ID Application". It is technically possible to sign with an arbitrary certificate, although such a signature will not be considered as valid by Gatekeeper. To enable signing with all kinds of certificates, set the compiler variable `sys.ext.macosAcceptAllCerts` to `true`. Expiration times will still be checked in that case, only the constraints on the certificate subject name will be removed.

(4) https://www.digicert.com/code-signing/microsoft-authenticode.htm

(5) https://en.wikipedia.org/wiki/PKCS_12

(6) https://www.openssl.org

(7) https://en.wikipedia.org/wiki/Extended_Validation_Certificate

(8) https://en.wikipedia.org/wiki/PKCS_11

(9) https://developer.apple.com

(10) https://developer.apple.com/support/developer-id/

You can find general information about code signing on macOS in the Apple code signing guide (11).

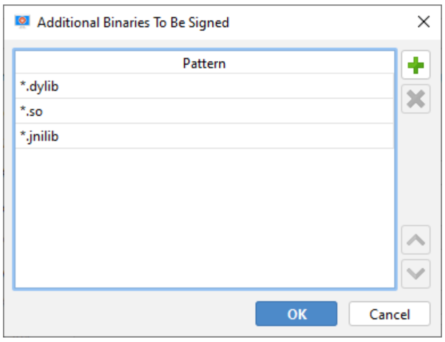
Notarizing media files on macOS

Apple offers a service that checks DMGs for security problems and adds them to their database. This is called "notarization" and is required starting with macOS 10.15. The exact steps for notarizing your application are described on the Apple developer web site (12).

However, Apple will only notarize applications that follow certain guidelines. The "hardened runtime" has to be enabled which install4j automatically does for you by adding the appropriate entries to the entitlements file. Also, all binaries in the DMG have to be signed. This also concerns binaries that are in a ZIP archive. Because JAR files are ZIP archives, the notarization process can detect binaries in JAR files. Some popular frameworks and libraries such as SWT or JNA ship native binaries in their JAR files. These contained binaries have to be signed as well.

For this purpose, install4j lets you configure a list name patterns for binaries. All files in the distribution tree are matched against these patterns and if a match is found, the corresponding file is signed if it is really a MACH-O binary (13). The reason why install4j cannot just automatically check all files in this way is that this check is rather expensive.

In addition, you can configure a list of name patterns for JAR files that should be scanned for binaries with the above name patterns. This only works for unsigned JAR files because the modification introduced by the signature would break the signature of a signed JAR file and install4j has no way of regenerating that signature.



Pattern
*.dylib
*.so
*.jnilib

The actual notarization of a media file is performed by uploading it with an Xcode command line tool to Apple while identifying yourself with the Apple ID that was used to create the code signing certificate and an app-specific password (14). If the app passes the inspection, another command line utility can be used to "staple" the notarization signature to the executable. That stapling is only necessary if a macOS machine is offline and cannot verify the notarization of an app by connecting to the internet.

If you build on macOS, install4j can perform the entire notarization process for you. In the install4j IDE, notarization must be enabled on the "General Settings->Code signing" step and an Apple ID has to be entered. If your Apple ID is used by multiple teams, you can optionally configure the provider short name corresponding to the `--asc-provider` command line parameter that you would use with `altool`. When building your project, install4j will ask for the associated password. For command line builds, you can avoid the interactive entry of a password by setting

(11) https://developer.apple.com/support/code-signing/

(12) https://developer.apple.com/documentation/security/notarizing_your_app_before_distribution

(13) https://en.wikipedia.org/wiki/Mach-O

(14) https://support.apple.com/en-us/HT204397

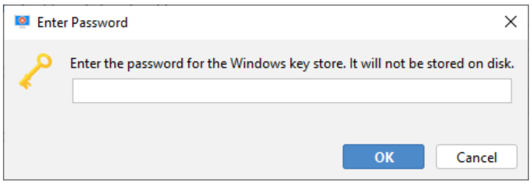
the `--apple-id-password` command line parameter or the equivalent parameter of the Gradle, Maven and Ant plugins.

If you build on other platforms, you will have to transfer the macOS media files to a macOS machine where Xcode is installed and perform a series of invocations to the altool command line tool (15)

Key store passwords

Private keys contain sensitive information and if they get into the wrong hands, your identity is compromised. Because of that, private keys are secured with a password. When install4j signs your installers and launchers, it needs to work with the private key.

When you start a build in the install4j IDE, you will be asked for the Windows and macOS key store passwords as required. install4j does not store those passwords to disk, but they are cached on a per-project level as long as the install4j IDE remains open.


Enter the password for the Windows key store. It will not be stored on disk.

When you run a command line build, the install4j command line compiler will ask you for the required passwords. If you want to fully automate a build with code signing, you can pass passwords on the command line by setting the `--win-keystore-password=[password]` and `--mac-keystore-password=[password]` command line parameters. The plugins for Gradle [p. 225], Maven [p. 230] and Ant [p. 239] offer the corresponding "winKeystorePassword" and "macKeystorePassword" attributes. Note that adding these passwords to shell scripts or ant build files constitutes a security risk.

In a setup where only a restricted number of people can build signed executables, you can use the `--disable-signing` command line parameter, the "disableSigning" attribute of the build system plugins or the corresponding build option in the "Build" step of the install4j IDE to temporarily disable code signing. In that way, other developers can build fully functional, unsigned installers without modifying the project file.

Time stamp counter-signing

Code signing certificates issued by certificate providers expire after a certain time. For Windows code signing, the expiry time is usually one to three years, after which you have to purchase a renewal from your certificate provider. Executables that were signed while the certificate was still valid are trusted indefinitely unless the certificate is revoked.

A computer that validates an executable compares the signing time and the expiry time of your certificate. Certificate providers have to prevent you from turning back the clock of your computer to circumvent the expiry of your certificate. This is why the signing time has to be counter-signed by a certificate provider. Certificate providers offer free web services that will confirm that a signature was performed at a particular time. This counter-signature is not related to a particular certificate, so you can use the web service of any certificate provider, regardless of where the certificate came from. install4j uses the DigiCert time stamp signing service at

```
http://timestamp.digicert.com
```

---

(15) https://developer.apple.com/documentation/security/notarizing_your_app_before_distribution/customizing_the_notarization_workflow

and falls back to the GlobalSign time stamp signing service at

```
http://timestamp.globalsign.com/?signature=sha2
```

if there is a failure.

To use a different service, define the compiler variable

```
sys.ext.timestampUrl=<URL>
```

where <URL> can contain multiple URLs separated by semicolons.

If the timestamp service call fails, install4j will retry up to 10 times or whatever the `sys.ext.counterSignRetry` compiler variable is set to.

Apple has its own time stamp signature server at

```
http://timestamp.apple.com/ts01
```

that can be changed with the compiler variable

```
sys.ext.macTimestampUrl=<URL>
```

Setting up a proxy for HTTP calls

The consequence of the time stamp counter-signature scheme is that you need an internet connection at build time. Many build servers are behind fire walls and you might need to set up a proxy to get internet connectivity and whitelist the above time stamp servers. install4j will try to auto-detect the proxy information. If that fails, the IDE will ask you for proxy information, but the command line builds will not ask for user-input in order to avoid hanging builds due to temporary internet connectivity problems.

For command line builds, you can pass the following VM parameters to the command line compiler:

- -DproxySet=true
- -DproxyHost=[host name]
- -DproxyPort=1234
- -DproxyAuth=true
- -DproxyAuthUser=[user name]
- -DproxyAuthPassword=[password]

The authentication parameters are optional, only the first 3 parameters are required to set up a proxy.

If you pass these parameters to the command line compiler, you have to prefix them with `-J` to mark them as VM parameters, such as

```
-J-DproxySet=true
```

The plugins for Gradle [p. 225], Maven [p. 230] and Ant [p. 239] offer way to set VM parameters without using the `-J` prefix.

A.24 Styling Of DMGs On MacOS

On macOS, software is usually delivered as a DMG. DMG stands for "Disk image" and contains a file system that can be mounted, rather than an archive that can be extracted. When the user double-clicks on a DMG file in the Finder, it is mounted to `/Volumes/[volume name]` and a new Finder window is opened for the mount point.

The Finder can by styled on a per-directory basis and the information about that styling is saved to a file named `.DS_Store` in every directory. This means that you can ship styling information with a DMG file. Styling includes setting a background image for the Finder window and that image file can be added to the DMG as well.

For single bundle GUI applications, a styled DMG generally includes a symbolic link to `/Applications` in the top-level folder of the DMG, so that user can drag the application bundle into the default installation directory with minimum effort.

install4j allows you to add any number of files and symbolic links to the DMG. All macOS media file types have a step named "DMG options and files" as a sub-step of the "Installer options" step. Here, you can add the top-level `.DS_Store` files, a background image and the symlink to `/Applications`.

Step-by-step instructions

To create your `.DS_Store` file, follow the steps below on a macOS machine where install4j is installed.

1. **Compile DMG**

   The first step is to compile your macOS media file from install4j without any custom styling. This DMG will be the template for which we will define the style. You cannot use just any other DMG, because each media file has a unique ID. When using background images, the `.DS_Store` file must have been created for a DMG with the same ID, otherwise the image will not be found reliably.

   When you recompile the media file in install4j, this ID remains the same, so you can add the `.DS_Store` file from a previously compiled DMG to the additional DMG files in the media wizard.

2. **Convert the read-only DMG to a writable DMG**

   The generated DMG is a read-only image. In order to make any modifications at all, we have to convert the DMG to a writable format.

   First, make sure that the DMG is not mounted. In a terminal, cd to the directory where the DMG was created and execute

   ```
   hdiutil convert hello.dmg -format UDRW -o hello_rw.dmg
   ```

   where "hello" has to be replaced by the actual name of your media file. Note that the last argument has "_rw" appended at the end, because the output DMG must be different from the input DMG.

3. **Enlarge the writable DMG**

   By default, a DMG generated by install4j is full. It is not possible to add any more files simply because the file system in it has no more available space. To enlarge the DMG, we first determine its current size by executing

```
hdiutil resize hello_4_0_rw.dmg
```

The "cur" column of the output shows the number 512-byte sectors. To add about 10 MB, we add 20000 to that number and execute

```
hdiutil resize -sectors <new number of sectors> hello_4_0_rw.dmg
```

To check the new size, run

```
hdiutil resize hello_4_0_rw.dmg
```

again.

4. **Mount DMG**

   We now mount the read/write DMG by executing

   ```
   hdiutil attach hello_4_0_rw.dmg
   ```

   and note the mount point /Volumes/[volume name] that is given by the output of the above command.

5. **Copy background image to DMG**

   To add a background image, we first have to copy the image to the DMG. We do not want the image file to show up in the finder, so we create a hidden directory in the DMG. To do that, we execute

   ```
   cd /Volumes/[volume name]
   mkdir .background
   ```

   To open this hidden directory in the Finder, we execute

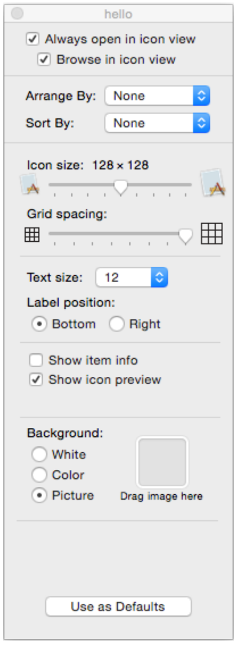
   ```
   cd .background
   open .
   ```

   Now, we open another Finder window, locate our background image and copy it to the hidden directory that is visible in the original Finder window.

6. **Select background image for DMG top-level folder**

   Because we need the Finder with the hidden directory in a minute, we leave it as it is, and double-click on the mounted volume on the desktop to open the default Finder window for the DMG. We position the new Finder window side-by side with the Finder window that shows the hidden directory.

   To start changing styles, we invoke *View->Show View Options*. This will show a tool window with styling controls. In the "Background" section, we choose "Picture" and notice the drop target for a picture file.



hello

☑ Always open in icon view
☑ Browse in icon view

Arrange By: None
Sort By: None

Icon size: 128 × 128

Grid spacing:

Text size: 12

Label position:
● Bottom ○ Right

☐ Show item info
☑ Show icon preview

Background:
○ White
○ Color
● Picture

Now we have to perform a somewhat tricky operation. From the Finder window that shows the hidden directory, we drag the image to the mentioned drop target in the view options dialog without activating that Finder window (otherwise the view options dialog would change its target folder).

Finally, we see can see the background image applied to our read/write DMG.

7. Adjust DMG finder window

Two properties of the Finder window should be adjusted: Invoke *View->Hide Toolbar* and resize the window so that it fits the size of the background image.

8. Add link to /Applications for single-bundle archives

If you have a single-bundle archive media file type, you probably want to add a drop-target for the installation. In the terminal, we execute

```
cd /Volumes/[volume name]
ln -s /Applications " "
```

This creates a link with an empty name that immediately shows up in the Finder window. The empty name is a good strategy to get around localization issues. The Applications folder has a special icon and is easily recognizable, so a name is not necessary.

9. Adjust icons

Now you can position the icons as needed and adjust the "Icon size" property in the view options dialog until they fit with your background image.

10. Extract .DS_Store file

The result of your work above is the `.DS_Store` file in the top-level folder of the DMG. Go to the terminal and copy it to your project folder so that you can reference it in the install4j IDE:

```
cp .DS_Store [project folder]/DS_Store
```

Note that we have omitted the leading dot before DS_Store in the target path. This makes it easier to work with the file and prevents confusion with the Finder.

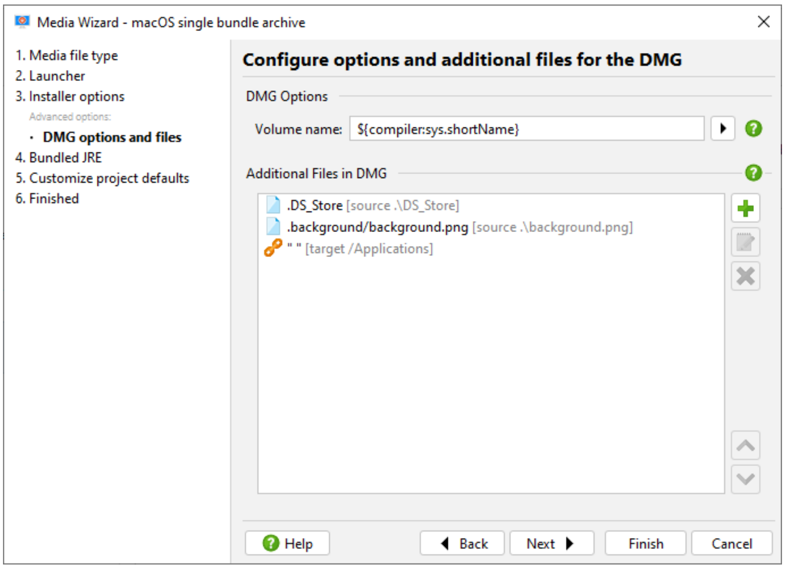
At this point, our work with the read/write DMG is finished. We should now delete it and also remove it from the Trash. If we don't do this, subsequent tests will automatically mount this DMG again. This is due to the "alias" feature in macOS. The .DS_Store contains an alias to the configured background image and as long as the original DMG still exists somewhere, it will open it from the template DMG instead of from the newly generated DMG.

Configuring the media file

In the media file wizard of the install4j project, we can now use the generated `.DS_Store` file. On the "Installer Options->DMG options and files" step we enter the `[project folder]/DS_Store` and give it the name `.DS_Store` in the DMG.

The background image is added with the name `.background/[image name with extension]` where the image name must be the same as on the read/write DMG. The `.background` folder will be created automatically.

If you have added a symbolic link to `/Applications`, you can add a corresponding symbolic link entry here, the name should also be set to the same name as in the read/write DMG. An empty name is entered as " " (with the quotes).



Media Wizard - macOS single bundle archive

Configure options and additional files for the DMG

Volume name: ${compiler:sys.shortName}

.DS_Store [source .\DS_Store]
.background/background.png [source .\background.png]
" " [target /Applications]

With the above files and symbolic links a newly generated DMG will look the same as the read/write DMG where the styling was added. When you tweak your styling in the future, you don't start from zero but with the styles that are already present in the generated DMG.

B Configuring Installer Beans

B.1 The Screens & Actions Configuration Step

The "Installer->Screens & Actions" step shows a tree representation of the installer, the uninstaller and other installer applications, such as updaters. The nodes in the tree are of the following types:

- **Applications** [p. 154]

  An application consist of a series of screens.

- **Screens** [p. 163]

  A screens displays information to the user, optionally gathers user input and optionally executes a series of actions when the user moves to the next screen.

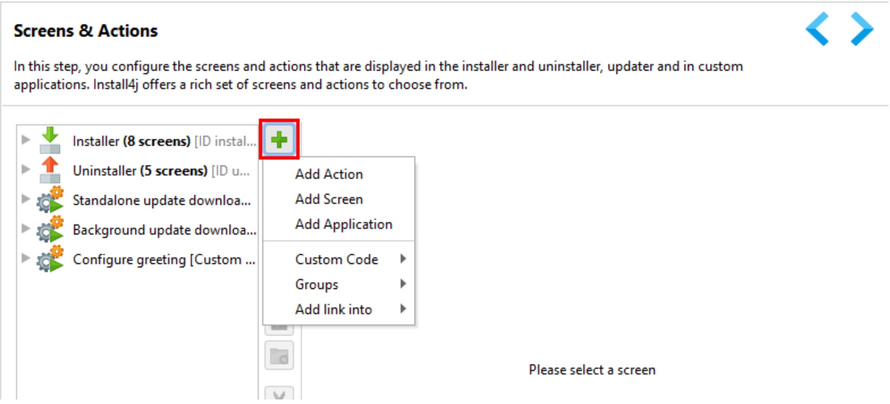
- **Actions** [p. 169]

  An action usually makes a modification to the installation.

In this chapter, the functionality and configuration options on the "Installer->Screens & Actions" step are discussed, the underlying concepts are discussed in a different help topic [p. 24].

Adding new installer elements

Installer elements are added by clicking the ✚ *Add* button.



Screens & Actions

In this step, you configure the screens and actions that are displayed in the installer and uninstaller, updater and in custom applications. Install4j offers a rich set of screens and actions to choose from.

Installer (8 screens) [ID instal...
Uninstaller (5 screens) [ID u...
Standalone update downloa...
Background update downloa...
Configure greeting [Custom ...
Add Action
Add Screen
Add Application
Custom Code
Groups
Add link into
Please select a screen

In the popup window you can select whether to add

- an action [p. 169], a screen [p. 163] or an application [p. 154]. Actions and screens are made available by install4j or are contributed by an installed extension [p. 218]. A registry dialog will be shown where you can select the desired screen or action. When adding an application, the application template dialog is displayed.
- an action or a screen that is contained in your custom code. New types of reusable actions or screens can be developed with the install4j API [p. 212]. In your custom code configuration [p. 152] you can specify code locations that are scanned for suitable classes.
- an action group or a screen group [p. 180]. The new group is initially empty. You can also create groups directly from a selection in the tree of installer elements.

Installer elements can only be added to appropriate parent elements. If no appropriate parent element is selected, install4j tries to find one by moving in the ancestor hierarchy from the current selection. If no appropriate parent element can be found, an error message is displayed.

- Applications are added at the top level.
- Screens and screen groups can be added to applications or screen groups.
- Actions and action groups can be added to screens or action groups.

Editing installer elements

If you select a single installer element in the tree of installer elements, you can edit its properties on the right side. Selecting multiple installer elements is possible on the same tree level, meaning that all selected elements have to be siblings in the tree.

When the configuration area is focused, you can transfer the focus back to the tree of installer elements with the keyboard by pressing `ALT-F1`.

The tree of installer elements provides the following actions in the toolbar on the right that operate on the current selection. You can also access these actions from the context menu or use the associated keyboard shortcuts.

- **Delete**

  All selected installer elements will be deleted after a confirmation dialog when invoking the *Delete* action. The deleted installer elements cannot be restored. You will be notified if deleting the selected installer elements would break links.

- **Rename**

  After you add an installer element, the tree of installer elements shows it with its default name. If you have multiple instances of the same installer element next to each other, a custom name makes it easier to distinguish these instances. You can assign a custom name to each installer element with the *Rename* action. The default name is still displayed in brackets after the custom name. To revert to the default, just enter an empty custom name in the rename dialog.

- **Comment**

  You can add comments to selected installer elements with the *Add Comments* action. When a comment is added, the affected installer elements will receive a "Comments" tab. After adding a comment to a single installer element, the comment area is focused automatically. Likewise, you can remove comments from one or more installer elements with the *Remove Comments* action.

  In order to visit all comments, you can use the *Show next comment* and *Show previous comment* actions. These actions will focus the comment area automatically and wrap around if no further comments can be found.

- **Disable**

  In order to "comment out" installer elements, you can use the *Disable* action. The configuration of the disabled installer elements will not be displayed, their entries in the tree of installer elements will be shown in gray and they will not be checked for errors when the project is built.

- **Copy and paste**

  install4j has a clipboard for installer elements. You can *Cut* or *Copy* installer elements to the clipboard and *Paste* them in the same project or in a different project. Note that references to launchers or references to files in the distribution tree might not be valid after pasting to a different project.

  Pasted installer elements are appended to the end of the same level that would be chosen if you added installer elements of that type. Sequence restrictions with respect to the already present installer elements may force a different order.

- **Reorder**

  If your selection is a single contiguous interval, you can move the entire block up or down in the list. The selection can only be moved on the same level with the reorder actions. To move the selection to a different parent, you can cut and paste it.

- **Group**

  You can create a screen group or an action group [p. 180] from the selected installer elements with the *Create Group* action. The new group will be inserted in place of the selected installer elements.

  You can dissolve a group with the *Dissolve Group* action. This action is only enabled if the selection consists of a single screen group or action group. The elements contained in the group will be inserted in place of the group. Nested groups will not be dissolved.

- **Link**

  You can reuse screens and actions by linking to a single definition. This is particularly useful if you define an installer maintenance application [p. 154] that should repeat parts of the installer, such as a number of forms that query the user for initial values to set up your application. Also, links are the only way to integrate screens and actions from merged project [p. 108] into the main project.

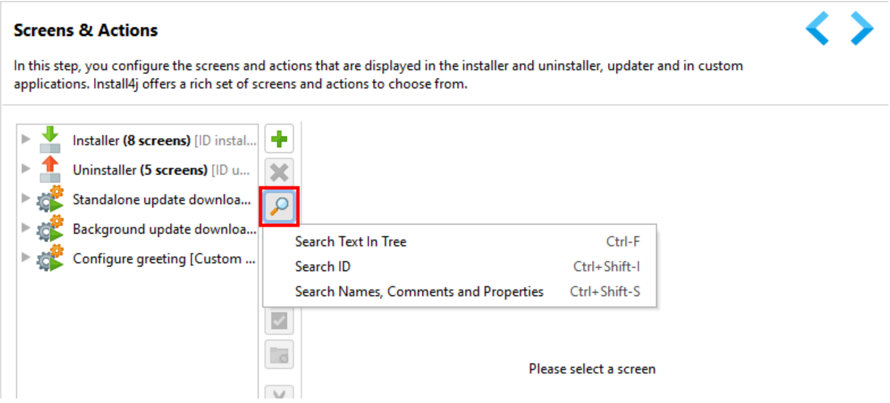
  In order to link to a screen, action, screen group or action group, you click on the add button and select *Add Link Into* from the popup menu. The first entry in that popup menu is always "This project" for links into the current project. If you have set up merged projects [p. 108], then you get an entry for each merged project. The configuration area of a link will only contain a button that selects the original definition in the tree of installer elements. For merged projects, the merged project is opened in a new window, unless it is already open.

  Another way to add a link into the same project is to select the installer element and invoke the *Copy Link* action. Then you navigate to the installer element where the link should be inserted and invoke the *Paste Link* action.

  For links into the same project, install4j ensures that there are no broken links in the tree of installer elements. When you delete an installer element, all links to it will be deleted as well. If that is the case, the deletion message will tell you how many links are about to be deleted. Links into merged projects may be broken, this condition is shown in in the configuration panel.

Searching for installer elements

In the log files, actions and screens are logged with their IDs. You can navigate to installer element if you know their ID by clicking on the search icon and choosing "Search ID" from the popup menu.



Screens & Actions

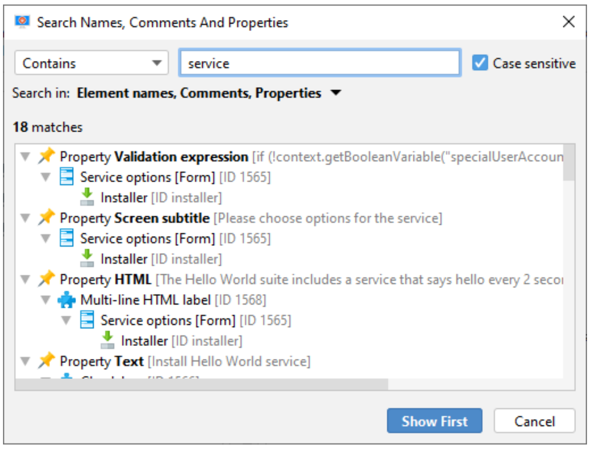
In this step, you configure the screens and actions that are displayed in the installer and uninstaller, updater and in custom applications. Install4j offers a rich set of screens and actions to choose from.

Installer (8 screens) [ID instal...
Uninstaller (5 screens) [ID u...
Standalone update downloa...
Background update downloa...
Configure greeting [Custom ...
Search Text In Tree Ctrl-F
Search ID Ctrl+Shift-I
Search Names, Comments and Properties Ctrl+Shift-S

Please select a screen

When a match is found the result tree shows the match at the top together with the reverse chain of installer elements that lead to it. You can either show the match itself or select any other element in the result tree and show that element instead when closing the dialog with the *Show* button. This works even if the target element is in a form component dialog or an action list or a property. The scope of the search is always rooted in the installer elements that are reachable from the current view.

A separate action "Search Names, Comments and Properties" is available to search for arbitrary patterns. You can disable any of the search types to narrow down the scope of the search.



Search Names, Comments And Properties

Contains ▾ service ☑ Case sensitive

Search in: **Element names, Comments, Properties** ▾

18 matches

- Property **Validation expression** [if (!context.getBooleanVariable("specialUserAccoun
  - Service options [Form] [ID 1565]
    - Installer [ID installer]
- Property **Screen subtitle** [Please choose options for the service]
  - Service options [Form] [ID 1565]
    - Installer [ID installer]
- Property **HTML** [The Hello World suite includes a service that says hello every 2 secoı
  - Multi-line HTML label [ID 1568]
    - Service options [Form] [ID 1565]
      - Installer [ID installer]
- Property **Text** [Install Hello World service]

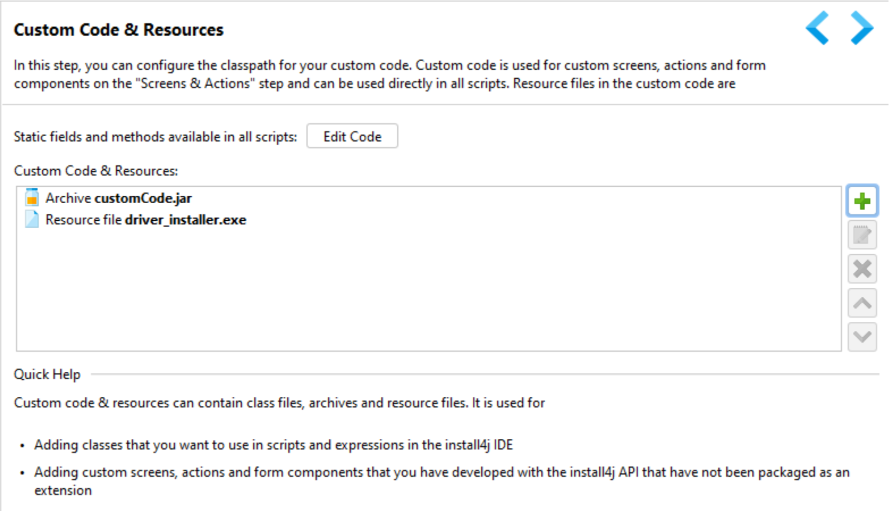
Display options for installer elements

When using the install4j API, you reference installer elements with IDs. You can show IDs in the tree of installer elements by toggling the *Show IDs* tool bar button.

In order to adjust the information density in the tree of installer elements, you can change the **icon size** by choosing large or small icons in the *Icon Size* sub-menu in the context menu. The default setting is to show large icons.

B.2 Custom Code & Resources Step

Custom code is configured on the "Installer->Screens & Actions->Custom Code" step.



Custom Code & Resources

In this step, you can configure the classpath for your custom code. Custom code is used for custom screens, actions and form components on the "Screens & Actions" step and can be used directly in all scripts. Resource files in the custom code are

Static fields and methods available in all scripts:

Custom Code & Resources:

Archive **customCode.jar**

Resource file **driver_installer.exe**

Quick Help

Custom code & resources can contain class files, archives and resource files. It is used for

- Adding classes that you want to use in scripts and expressions in the install4j IDE
- Adding custom screens, actions and form components that you have developed with the install4j API that have not been packaged as an extension

Entries in the custom code are used for

- specifying additional libraries that can be used in scripts and expressions [p. 29] of screens [p. 163], actions [p. 169] and form components [p. 183].
- developing new types of actions, screens or form components with the install4j API. See the help topic on using the API [p. 212] for more information.

  Before you start to develop a new action, have a look at the available actions [p. 169] and screens [p. 163]. If it is just a few lines of code, you can use the "Run script" action to enter them directly into install4j. If you would like to collect user input, most use cases can be solved with a form screen [p. 46].

  An alternative way of adding your beans to the install4j is packaging them as an extension [p. 218]. In that case, you can select them directly from the standard registry dialogs instead of having to go through the "Search in custom code" menu entries when adding beans to the installer.
- including resource files into the installer. Resource files are arbitrary files like DLLs, external executables or text files that have to be available before the "Install files" action has run. All class files are packed into a single `user.jar` file, archives and resource files are extracted to the `user` subdirectory in the working directory of the installer. You can access a resource file named `file.txt` with the following expression in custom code:

  ```
  new File("user", "file.txt")
  ```

  To specify resource files in text fields in the installer configuration, use the `sys.resourceDir` installer variable:

  ```
  ${installer:sys.resourceDir}/file.txt
  ```

  To load native libraries in custom code, do not use `System.load(..)`, but rather `Util.loadNativeFromResources(...)` to load the library in the same class loader that loads scripts. For example, if you have added a native library `jni.dll` to your custom code, you can load it in a "Run script" action by calling

```
Util.loadNativeFromResources("jni.dll");
```

The following types of custom code locations are available:

- **Class or resource files**

  For simple actions, screens or form components that do not depend on other classes, it is easiest to insert their class files directly, especially if you build your installer extensions together with your application. Anonymous inner classes will be included automatically. If you select a resource file, for example an image, it will be added to the top-level directory of the custom JAR file and will be available via `Class.getResourceAsStream()`.

- **Directories**

  With this type of entry you can add an entire directory. Make sure to select a classpath root directory, otherwise your classes cannot be loaded.

- **Scan Directories**

  Use this type of entry to add all JAR and ZIP files in a selected directory.

- **Archives**

  Use this type of entry to add a JAR file. Files that are present in both the custom code as well as the distribution tree will **not be packaged twice**. Files that are also present in the distribution tree can be freely added to your custom code, they will not increase the size of your installer. The compiler checks the source path of included files to determine if they are already present in the installer.

B.3 Configuring Applications

Applications are configured on the Screens & and actions step [p. 148].

The top-level nodes represent the different applications that can be configured for the project. There are 3 types of applications:

- **Installer**

  The installer is the application that is executed when the media file is invoked by the user, for example, when the user double-clicks on the installer executable in the Windows explorer. The installer cannot be deleted from the tree of installer elements.

- **Uninstaller**

  The uninstaller is a special application for uninstalling an installation. It is used in various contexts and can be

  - directly invoked by the user
  - invoked from the Windows software registry
  - invoked by the "Uninstall previous installation" action

  The uninstaller cannot be deleted from the tree of installer elements. If you do not wish to generate an uninstaller, you can disable it [p. 148].

- **Custom installer application**

  You can add any number of custom installer applications that can be invoked after the installation. install4j comes with several templates for auto-updater downloaders [p. 114]. Custom applications can also be used for writing maintenance applications for your installation.

  You can add a new custom installer application by clicking on the ✚ *Add* button on the right side of the list and choosing `Add Application` from the popup. The application templates dialog will be displayed and lets you choose a starting point for your custom installer application. Application templates are entirely made up of existing screens, actions and form components. You can modify the selected application template after adding it.

  Unlike the installer and uninstaller above, custom applications are also created for archive media files [p. 126]. See the help topic on media files [p. 126] for more information on how to create first-run installers for archives.

  Custom installer applications with a non-empty "Executable directory" property are automatically added to the "Default file set". If you leave the executable directory empty, the custom installer application is added to the `.install4j` directory and will always be included, regardless of the installation component configuration.

Each installer application has a **startup sequence** of actions [p. 169]. Those actions are executed before the installer application presents a user interface. If any of these actions fails and has a "Quit on failure" failure strategy, the installer application will not be shown.

Properties of installer applications

Common properties of installer applications are:

- **Executable icon [Executable]**

  By default, a standard installer icon is used for the executable. To customize the icon, press the customizer button in the configuration pane.

- **Allow unattended mode [Execution Modes]**

  If selected, the user can pass `-q` as an argument to run the installer application without a GUI. No user input is required, the installer applications works with the default values. Please see the corresponding help topic on installer modes for more information. All standard actions and standard screens support unattended installations. If your policy forbids unattended installations or if you include custom code that cannot handle unattended installations, you can disable them by deselecting this property.

- **Progress interface creation script [Configuration]**

  If you would like to implement your own way of displaying progress information for unattended installations, you can do so by returning a custom implementation of `com.install4j.api.context.UnattendedProgressInterface` from this script. If you return `null`, no progress information will be shown just as if this script had not been set. There is a default implementation `com.install4j.api.context.DefaultUnattendedProgressInterface` that does nothing for all its operations. You can derive from that class if you just need to implement a few particular methods in the progress interface.

  If you just need a simple dialog that shows progress information in unattended mode, please choose the "Unattended mode with progress dialog" execution mode instead.

  This property is only visible if "Allow unattended mode" is selected.

- **Allow console installations [Execution Modes]**

  If selected, the user can pass `-c` as an argument to run the installer application on the console. The installer asks for user input on the console in that mode. Please see the corresponding help topic on installer modes for more information. All standard actions and standard screens support console installations, form screens are also fully mapped to console installers. If your policy forbids console installations or if you include custom code that cannot handle console installations, you can disable them by deselecting this property.

- **Console screen change handler [Configuration]**

  By default, a screen in console mode does not show any particular separation. You insert your own custom display with this script. The title parameter gives you access to the title of the screen. In console mode, screens display their subtitle only, so the title string will not be displayed again.

  This property is only visible if "Allow console installations" is selected.

- **Disable console mode on Windows [Configuration]**

  Offer console mode only on non-Windows platforms.

  This property is only visible if "Allow console installations" is selected.

- **Fall back to console mode on Unix [Configuration]**

  On Unix, users often operate in environments where no X11 server is available and no GUI can be displayed. The installer will fallback to console mode if console mode execution is allowed and this option is selected. Otherwise an error message will be displayed that tells the user how to invoke the installer in console mode.

  This property is only visible if "Allow console installations" is selected.

- **Default execution mode [Execution Modes]**

  The default execution mode for the installer application. By default, a GUI wizard will be shown, but it is also possible to run in console mode or unattended mode by default.

- **Title for progress dialog [Configuration]**

  The title for the progress dialog, for example "Updating installation".This title and the unattended mode with a progress window can also be set by passing `-splash [title]` as an argument from the command line.

  This property is only visible if "Default execution mode" is set to "Unattended mode with progress dialog".

- **Show alerts [Configuration]**

  By default, no alerts are shown in unattended mode. This includes messages boxes, error alerts and questions. By selecting this property, alerts are enabled for unattended executions with a progress dialog.

  This mode can also be activated by passing `-alerts` as an argument from the command line.

  This property is only visible if "Default execution mode" is set to "Unattended mode with progress dialog".

- **Windows console executable [Execution Modes]**

  If selected, a console executable will be created on Windows. A non-hideable console will be shown when the installer is double-clicked in the explorer. This improves the user experience for a console-only installer (default execution mode set to console) and allows execution through `rsh`.

- **VM parameters [Execution Options]**

  If you need to pass special VM parameters to the installer application, you can enter them here. A common case would be to raise the maximum heap size with a different -Xmx parameter if your installers require a lot of memory.

- **Arguments [Execution Options]**

  If you need to pass fixed default arguments to the installer application, you can enter them here. For example, if you want to display a splash screen in unattended mode by default, you can set the arguments to `-splash "Installing ..."`. Please note that command line arguments will be appended to this list, so it is not possible to "override" a fixed argument from the command line.

- **Rollback on failure [Execution Options]**

  If selected, the installer application will try to restore the state before the last rollback barrier by rolling back all actions that were executed since the last barrier. Any screen or action can be selected as a rollback barrier with the property "Rollback barrier". If no rollback barrier was encountered, all executed actions will be rolled back.

- **Help customizer script [General Customization Options]**

  If the user starts the installer application with one of the arguments `-h -help /?`, help regarding the available command line options will be displayed. If you have your own command line options you can customize this help with this script. The script receives a `List` containing `String` arrays of length 2 with the options and explanations. You can add options like this: `options.add(new String[] {"/mySwitch", "Explanation of mySwitch"}}`. You can also delete default options in the list.Attention: The context parameter has not been initialized at that point.

  In order to get extra command line arguments in the installer, call `context.getExtraCommandLineArguments()` in any script.

- **Customize version info [Windows]**

  If selected, you can customize the fields of the Windows version info in the nested properties. A windows version info is always generated for the executable with default values for product name and file version taken from the general settings.

- **Copyright [Configuration]**

  The copyright field in the version resource. If empty, the publisher name from the general settings is used.

  This property is only visible if "Customize version info" is selected.

- **File description [Configuration]**

  The file description field in the version resource. If empty, the full name from the general settings is used.

  This property is only visible if "Customize version info" is selected.

- **File version [Configuration]**

  The file version field in the version resource. If empty, the version from the general settings is used. The file version must consist of 4 numbers separated by spaces, commas or dots.

  This property is only visible if "Customize version info" is selected.

- **Internal name [Configuration]**

  The internal name field in the version resource. If empty, the short name from the general settings is used.

  This property is only visible if "Customize version info" is selected.

- **Product name [Configuration]**

  The product name field in the version resource. If empty, the full name from the general settings is used.

  This property is only visible if "Customize version info" is selected.

- **macOS entitlements file [macOS]**

  If you have configured code signing for macOS, an entitlements file can unlock certain features on macOS, such as iCloud storage or push notifications.

- **Custom fragment for Info.plist [macOS]**

  On macOS, you may want to add additional elements to the Info.plist file of the application bundle in order to customize its behavior in ways that are not directly supported by install4j.

- **Custom script fragment [Unix]**

  On Unix and Linux, the JVM for an installer application is launched by a shell script. To add your own code to the shell script, you can specify a script fragment that is added immediately before the java invocation takes place.

- **Style [GUI Options]**

  The default screen style for this installer application. Screens and screen groups can override this style.

- **Window width [GUI Options]**

  The width of the window displayed by the installer application. The default value is 500. If the "Size client area" property is selected, this does not include the size of the window frame border.

- **Window height [GUI Options]**

  The height of the window displayed by the installer application. The default value is 390.If the "Size client area" property is selected, this does not include the size of the window frame border.

- **Size client area [GUI Options]**

  If selected, the supplied size for the window will not be applied to the outer dimensions of the window, but to the actually usable area inside the window. Unusually large window frame borders can occur due to user settings (accessibility, window themes, etc.) and may interfere with banner images or introduce unwanted scroll bars to form screens.

- **Resizable [GUI Options]**

  If selected, the window displayed by the installer application is resizable.

- **Action elevation type [Privileges]**

  If any contained actions should run in the elevated helper process, if their "Action elevation type" property is set to "Inherit from parent".An elevated helper process is available on Windows and macOS if the process has been started without admin privileges and the "Request privileges" action has been configured to require full privileges.

Custom applications as well as the uninstaller are added to the distribution tree and have additional related properties:

- **Executable name [Executable]**

  The name of the executable for the . Please enter a name without any path components and without a file extension.

- **Executable directory [Executable]**

  The directory to which the executable of the will be written. If empty, it will be placed in the `.install4j` runtime directory.

- **Use custom application bundle name [macOS]**

  If selected, a different application bundle name is used on macOS. Executable names on macOS are localizable. Otherwise, the value of the "Executable name" property is used for the application bundle name.

- **Custom application bundle name [Configuration]**

  The application bundle name to be used for macOS media files. Bundle names on macOS are shown in the Finder and are localizable. For example, the executable name could be set to `${i18n:myLauncherName(${compiler:sys.fullName})}` where `myLauncherName` is an i18n message with value "Launcher for {0}".

  This property is only visible if "Use custom application bundle name" is selected.

- **Unix mode [Unix]**

  The executable mode for the on Unix.

The remaining properties that are specific to the installer are:

- **Suppress initial progress dialog [Execution Options]**

  If selected, the initial native progress dialog of the installer is not displayed.

- **Replacement script for language code [General Customization Options]**

  With this script you can replace the language that the installer will run with.

  Parameters: The parameter `languageCode` contains the 2-letter ISO 639 code of the auto-detected language. If auto-detection has not been enabled on the languages step of the general settings, the parameter will be `null`.

  Return value: If you return `null`, the language selection dialog will be shown, if you return a language code, the language selection dialog will not be shown and the returned language will be used. If the returned language code is a language that is not configured for this installer, the language selection dialog will be shown.

- **Create log file for stderr output [Windows]**

  If selected, and output on stderr is detected, a log file will be created and all output to stderr will be redirected to that file.

- **Log file for stderr [Configuration]**

  The log file for the stderr output relative to the installer media file.

  This property is only visible if "Create log file for stderr output" is selected.

Finally, custom installer applications have the following additional properties:

- **Create executable [Executable]**

  If selected, an executable for this installer application will be created. If not selected, this application launcher can only be invoked with the `com.install4j.api.launcher.ApplicationLauncher` API or an automatic launcher integration.

  For macOS single bundles, executables for installer applications are never created.

- **Single instance [Configuration]**

  If checked the application will ensure at startup that there is only one instance running per user account.

  This property is only visible if "Create executable" is selected.

- **File set [Executable]**

  Choose the file set to which the installer application is added. File sets can be defined on the Files->Define Distribution Tree step.

  This property is only visible if "Create executable" is selected.

- **Change working directory [Execution Options]**

  If selected the working directory will be changed to the value in 'Working directory' at startup.

- **Working directory [Configuration]**

  The working directory to be used when 'Change working directory' is selected.

  This property is only visible if "Change working directory" is selected.

- **Execution level [Windows]**

  The execution level for this application. If you want to modify files in the installation direction, you most likely need administrator rights. This is only relevant for Windows Vista and higher.

- **Window title [GUI Options]**

  The title of the application window.

- **Show message when user cancels [GUI Options]**

  If selected, a message will be shown when the user cancels the installer application by clicking on the "Cancel" button or closing the application frame.

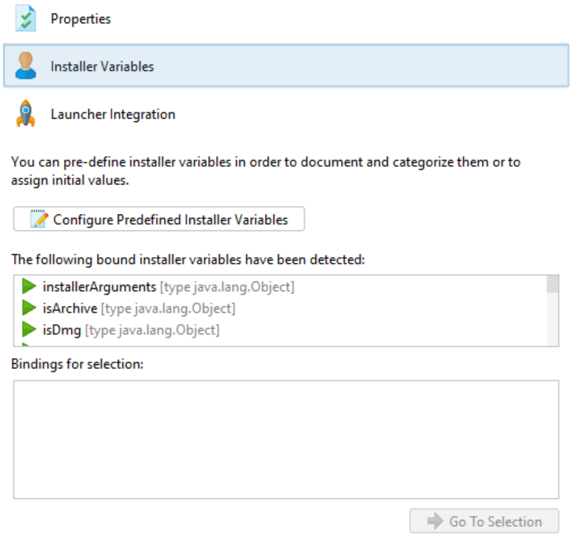
- **Cancel message [Configuration]**

  The message that is shown if the user cancels the installer application by clicking on the "Cancel" button or closing the application frame. The options that are presented to the user are "Cancel" or "Continue".

  This property is only visible if "Show message when user cancels" is selected.

Configuring installer variables

The second tab in the configuration area for installer applications is the **Installer variables** tab. Here, you can check the bindings for all detected installer variables and pre-define installer variables. For more information, see the help topic on variables [p. 63].



You can pre-define installer variables in order to document and categorize them or to assign initial values.

The following bound installer variables have been detected:

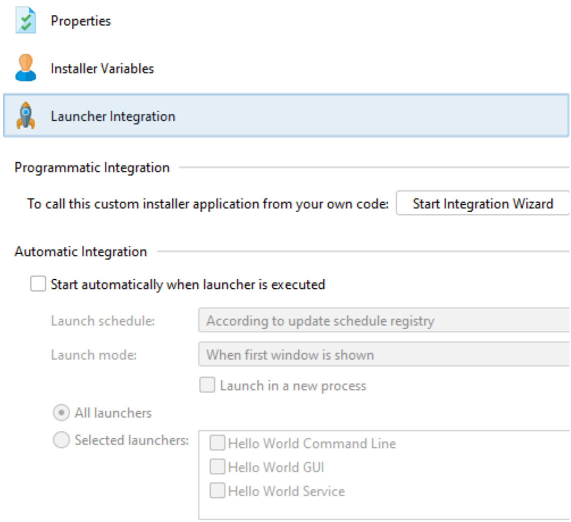
- installerArguments [type java.lang.Object]
- isArchive [type java.lang.Object]
- isDmg [type java.lang.Object]

Bindings for selection:

An additional feature with respect to the variable selection dialog is that you can navigate to a binding by selecting an element in the binding tree at the bottom and click on the *Go To Selection* button.

Launcher integrations

Custom installer applications have a **Launcher integrations** tab in the configuration area that helps you to start them when launchers are executed.



Programmatic Integration

To call this custom installer application from your own code:

Automatic Integration

☐ Start automatically when launcher is executed

Launch schedule: According to update schedule registry

Launch mode: When first window is shown

☐ Launch in a new process

● All launchers

○ Selected launchers:
☐ Hello World Command Line
☐ Hello World GUI
☐ Hello World Service

One way to start an installer application is programmatically, by using the install4j API [p. 212]. To get the code snippet for starting the selected installer application, click on the *Start integration wizard* button. The integration wizard will present a number of options that control the condition and possible call backs from the installer application.

Another way to start an installer application is automatically, by defining a **launch schedule** and a **launch mode**. The launch schedule is one of

- **Always**

  Every time you start the launcher, the installer application will be started as well.

- **According to update schedule**

  install4j provides a built-in update schedule registry that can be configured by the user on a form screen with an "Update schedule selector" form component. Also, you can programatically modify the update schedule through the class `com.install4j.api.update.UpdateScheduleRegistry` in the API. The selected installer application will be started only if the update schedule requires an update check.

- **First run of any launcher in archive media file by the current user**

  For archive media files (such as a Windows ZIP file), no installer is available. To execute a sequence of screens and actions when a launcher is started for the first time after the archive has been extracted, use this launch schedule. It may be convenient to link to screen groups in the installer in order to avoid duplicating configuration in your custom installer application.

  In your launcher, you can check for this condition with

  ```
  com.install4j.api.launcher.ApplicationLauncher.isNewArchiveInstallation()
  ```

  in case you want to perform some actions outside of a custom installer application.

The launch mode is one of

- **Blocking at start up**

  When the launcher is started, the selected installer application will be started first. When the installer application terminates, the launcher will then start up, unless a "Shut down calling launcher" action has been executed.

- **Non-blocking at start up**

  When the launcher is started, the selected installer application will be started immediately. The launcher continues to start up in parallel.

- **When first window is shown**

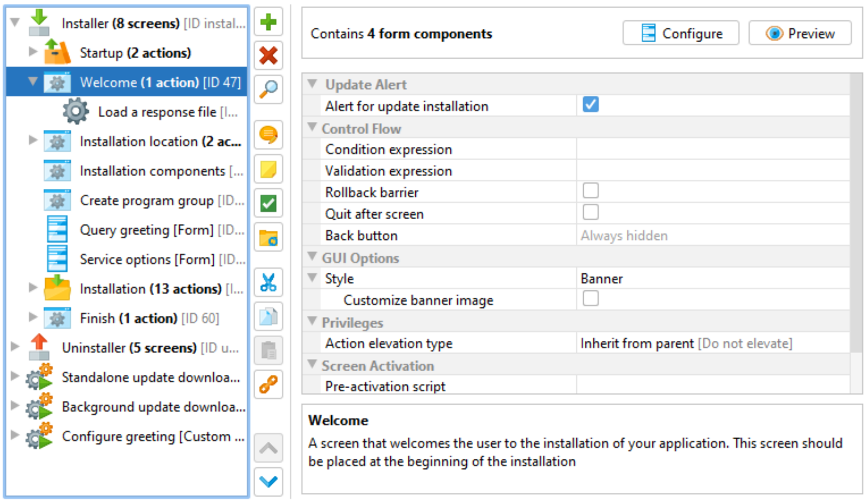
  The selected installer application will be started when the first window is shown. This works for AWT, Swing and SWT applications. If you have an SWT application, the "Uses SWT" check box in the "Executable info" step of the launcher wizard [p. 36] must be selected.

Just like with the API, the installer application can be started in the launcher process itself or in a new process. By default, the installer application is started in the same process. If the "Blocking at start up" or "Non-blocking at start up" launch modes are selected, the look and feel is set to the system look and feel. For the "When first window is shown" launch mode, the look and feel is not changed, so your own look and feel will be used. When the installer application is executed in the same process, the "Shutdown calling launcher" action has a different effect: The whole process will be terminated when the installer application exits.

By default, the selected installer application is started for all launchers in your project. If this is not desired, you can restrict the integration to selected launchers. Note that if "All launchers" is selected and the project is merged into another project, the integration will be performed for all launchers in the main project as well.

B.4 Configuring Screens

Screens are configured on the Installer->Screens & Actions step [p. 148]. A screen is a single step in an installer application. It displays information to the user or gathers user input.


Installer (8 screens) [ID instal...
Startup (2 actions)
Welcome (1 action) [ID 47]
Load a response file [I...
Installation location (2 ac...
Installation components [...
Create program group [ID...
Query greeting [Form] [ID...
Service options [Form] [ID...
Installation (13 actions) [I...
Finish (1 action) [ID 60]
Uninstaller (5 screens) [ID u...
Standalone update downloa...
Background update downloa...
Configure greeting [Custom ...
Contains 4 form components
Configure
Preview

Update Alert
Alert for update installation ☑
Control Flow
Condition expression
Validation expression
Rollback barrier ☐
Quit after screen ☐
Back button: Always hidden
GUI Options
Style: Banner
Customize banner image ☐
Privileges
Action elevation type: Inherit from parent [Do not elevate]
Screen Activation
Pre-activation script

Welcome

A screen that welcomes the user to the installation of your application. This screen should be placed at the beginning of the installation

If a screen has attached actions [p. 169], there will be an expand control to the left of the screen icon that allows you to show the associated actions.

Some screens only make sense when corresponding actions are used later on in the installer or uninstaller. For example, the "Services" screen will only be displayed at runtime if there are "Install a service" actions present on a subsequent screen. If such a dependency is not fulfilled after adding a screen, a corresponding notification is displayed.

Properties of screens

Common properties of screens are:

- **Action elevation type [Privileges]**

  If any contained actions should run in the elevated helper process, if their "Action elevation type" property is set to "Inherit from parent".An elevated helper process is available on Windows and macOS if the process has been started without admin privileges and the "Request privileges" action has been configured to require full privileges.

- **Style [GUI Options]**

  The default screen style for this installer application. Screens and screen groups can override this style.

- **Condition expression [Control Flow]**

  This expression is evaluated to decide whether the screen is displayed. If the expression or script returns false, the current screen will be skipped. This expression or script should not have any side-effects, it will be called while another screen is still being displayed.

- **Rollback barrier [Control Flow]**

  If the screen should be a rollback barrier. When a rollback barrier is completed, none of the preceding actions will be rolled back. You can use this property to prevent an incomplete rollback of complex changes or to protect actions from rollback when the user hits "Cancel" in the post-install phase.

- **Exit code [Control Flow]**

  If the "Rollback barrier" property is selected, and a rollback terminates at this screen, this property determines the exit code of the installer. By default, reaching a rollback barrier during a rollback is considered a success, but you can signal a failure by specifying a non-zero exit code here.

  This property is only visible if "Rollback barrier" is selected.

- **Validation expression [Control Flow]**

  This expression or script is called when the user clicks the next button. If it returns false, the current screen will be displayed again. You can use this to validate user input. Error messages are not displayed automatically, you can use the Util.showErrorMessage(String errorMessage) method in your script.

- **Quit after screen [Control Flow]**

  If the screen should have a "Finish" button instead of a "Next" button. The installer or uninstaller will quit after this screen. The "Cancel" button will not be visible if this option is checked.

- **Back button [Control Flow]**

  Allowing the user to go back to previous screens can be problematic if the previous screen has actions attached that cannot be executed multiple times. By default, every action is just executed once, all actions have a property to allow multiple execution. The default behavior is the "Safe back button", where the back button is hidden if the previous screen has actions attached that cannot be executed multiple times.

- **Wizard index [Screen Activation]**

  Every screen can set or change the current wizard index. The wizard index is an optional panel on the left side of the wizard that shows overall installation progress. You can leave the index unchanged as it was set by a previous screen, change the step in the current wizard index, removed the current wizard index ot configure a new wizard index. For conditional construction of a wizard index, please use the `com.install4j.api.context.WizardIndex` class in the "Pre-activation" script.

- **Step key**

  The key for the step in the wizard index that should be activated.

  This property is only visible if "Wizard index" is set to "Activate another step".

- **Steps**

  The steps that are displayed by the wizard index. Each step has a key that you can use to switch to that step later on by setting the wizard index property to "Activate another step" and specifying that key.

  This property is only visible if "Wizard index" is set to "Set a new wizard index".

- **Initial key**

  The key of the step in the wizard index that should be initially selected. Leave empty to select the first step.

  This property is only visible if "Wizard index" is set to "Set a new wizard index".

- **Partially defined**

  If selected, the list of wizard index steps will be partially defined. This means that a "..." entry will be appended at the bottom.

  This property is only visible if "Wizard index" is set to "Set a new wizard index".

- **Numbered**

  If selected, the steps in the wizard index are numbered.

  This property is only visible if "Wizard index" is set to "Set a new wizard index".

- **Maximum width**

  The maximum width of the wizard index in pixels. The preferred with is determined by the longest step name, the maximum width is an upper bound for the actual width.

  This property is only visible if "Wizard index" is set to "Set a new wizard index".

- **Minimum width**

  The minimum width of the wizard index in pixels. The preferred with is determined by the longest step name, the minimum width is a lower bound for the actual width.

  This property is only visible if "Wizard index" is set to "Set a new wizard index".

- **Background color**

  The background color for the index panel. Set to "None" to restore the default color.

  This property is only visible if "Wizard index" is set to "Set a new wizard index".

- **Foreground color**

  The foreground color for the index panel. Set to "None" to restore the default color.

  This property is only visible if "Wizard index" is set to "Set a new wizard index".

- **Background image**

  The image file for the background of the wizard index panel. Leave empty if no background image is required.

  This property is only visible if "Wizard index" is set to "Set a new wizard index".

- **Image anchor**

  The anchor for the background image. The default value is "North".

- **Pre-activation script [Screen Activation]**

  This script is called each time just before the screen is displayed.

- **Post-activation script [Screen Activation]**

  This script is called each time just after the screen has been displayed. It is not invoked in console or unattended mode.

Available screens

The following standard screens are available in install4j:

Empty form

An empty form to which form components can be added. By default, form components are layouted along the vertical axis, but you can use layout groups for greater flexibility. Form components with user input are bound to installer variables that can by referenced by other elements in the installer, for example by actions.

Category: Form templates

Banner with header at the top

A form that has "Banner" as the default style and a configurable header label at the top.

Directory selection

A form that asks the user to select a directory. All displayed messages are configurable.

Display PDF file

A form that displays a PDF file in an embedded cross-platform PDF viewer.

Display progress

A form that displays a progress bar with a status line capturing the progress information of associated actions. The default post-activation script executes any associated actions immediately when the screen is activated. All displayed messages are configurable.

Display text

A form that displays text to the user, either plain text or HTML. All displayed messages are configurable.

Program group selection

A screen that allows the user to select a program group on Microsoft Windows. All displayed messages are configurable.

Category: Standard screens

Welcome

A screen that welcomes the user to the installation of your application. This screen should be placed at the beginning of the installation

Display license agreement

A screen that displays a license agreement to the user, either plain text or HTML. The license agreement must be accepted before the installation continues.

Installation location

The screen that asks the user where to install the application. This determines the principal installation directory.

Installation type

A screen that displays a list of installation types that correspond to configurable set of installation components. The default types "Full","Standard" and "Customize" are provided by default, with localized names and descriptions. Installation components are configured in the install4j IDE on the "Files->Installation Components" step

The "Installation components" screen may be hidden by this screen, depending on the installation type selected by the user. This screen will not be shown if no installation components are defined.

You can choose for each installation type if it should be customizable or not. If the installation type that is selected by the user is customizable, the "Installation components" screen will be shown if present, otherwise that screen will be skipped. This condition can also be checked by inspecting the boolean value of the installer variable `sys.preventComponentCustomization`.

Installation components

A screen that displays all installation components and asks the user which components should be installed. This screen will not be shown if no installation components are defined.

Create program group

A screen that allows the user to select the default program group. Under Windows, this screen sets installer variables that influence "Create program group" and "Create start menu entry" entry actions. Under Unix, the screen asks the user whether and where symbolic links to launchers should to be created. Under macOS, the screen is not shown.

File associations

A screen that displays a list of all subsequent file association actions and asks the user which associations should be made. This screen will not be shown if there are no corresponding file association actions after this screen.

Additional confirmations

A screen that displays a list of confirmations as check boxes whose results can be used in condition expressions for actions. While other types of form components can be added to this screen, only check boxes and other simple elements are consistent with the displayed text. For arbitrary forms, use the "Configurable form" screen instead.

Installation

The screen that displays displays the installation progress. Where possible, installation actions should be added to this screen.

Display information

A screen that displays text to the user, either plain text or HTML. In contrast to the "Display text" form template, all messages on this screen are pre-defined and localized.

Finish

A screen that tells the user that the installation is finished. This screen should be placed at the end of the installation.

Uninstall Welcome

A screen that welcomes the user to the uninstallation of your application. This screen should be placed at the beginning of the uninstallation.

Uninstallation

The screen that displays displays the uninstallation progress. Where possible, uninstallation actions should be added to this screen.

Uninstallation failure

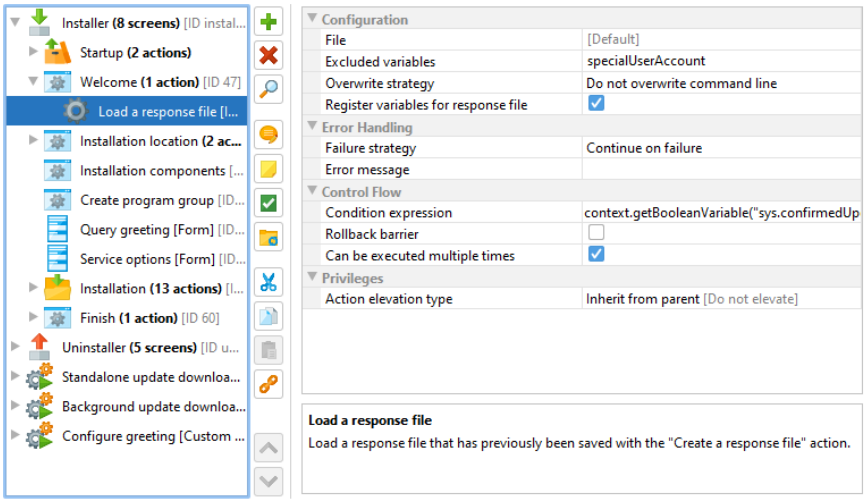
The screen that is displayed if the uninstallation was not completed successfully. Further information regarding the uninstallation problems is displayed to the user. This screen is not shown if the uninstallation was completed successfully or if it is placed before the uninstallation screen. The uninstaller will terminate after showing this screen in case of failure.

Uninstallation success

The screen that is displayed if the uninstallation was completed successfully.

B.5 Configuring Actions

Actions are configured on the Installer->Screens & Actions step [p. 148]. An action performs a configurable unit of work of the installer application.


Installer (8 screens) [ID instal...
Startup (2 actions)
Welcome (1 action) [ID 47]
Load a response file [I...
Installation location (2 ac...
Installation components [...
Create program group [ID...
Query greeting [Form] [ID...
Service options [Form] [ID...
Installation (13 actions) [I...
Finish (1 action) [ID 60]
Uninstaller (5 screens) [ID u...
Standalone update downloa...
Background update downloa...
Configure greeting [Custom ...

Configuration	
File	[Default]
Excluded variables	specialUserAccount
Overwrite strategy	Do not overwrite command line
Register variables for response file	☑
Error Handling	
Failure strategy	Continue on failure
Error message	
Control Flow	
Condition expression	context.getBooleanVariable("sys.confirmedUp
Rollback barrier	☐
Can be executed multiple times	☑
Privileges	
Action elevation type	Inherit from parent [Do not elevate]

Load a response file

Load a response file that has previously been saved with the "Create a response file" action.

Actions are attached to screens [p. 163] or they are part of the "Startup sequence" that allows you to perform actions before the installer or uninstaller is displayed. If any one of these actions fails and has a "Quit on failure" failure strategy, the installer application will not be shown.

Most often, actions are added to the "Installation" or "Uninstallation" screens. The advantage of those screens is that they have a progress bar and a status display that is utilized by actions. If a screen does not expose a progress interface, the status and progress messages of attached actions are lost. This is no problem for near-instantaneous actions such as setting an environment variable, but for time-consuming operations the user should be informed about progress, even if it is only an indeterminate progress bar. As an alternative to the "Installation" or "Uninstallation" screens, you can use "Display progress" screens to create additional installation phases.

Some actions have an "affinity" to a particular screen and will suggest to add themselves to that screen, such as the actions in the "Final options" category which would like to go to the "Finish" screen. However, this is only a suggestion to guide you for the most common use cases.

Some actions have an associated screen that allows the user to modify the behavior of the action. For example, the "Install a service" action has a corresponding "Services" screen where the user can decide whether the service should be installed and started when booting. If such a relationship exists, a corresponding notification is displayed after adding an action.

Properties of actions

Common properties of actions are:

- **Action elevation type [Privileges]**

  If the action should run in the elevated helper process.An elevated helper process is available on Windows and macOS if the process has been started without admin privileges and the "Request privileges" action has been configured to require full privileges.

- **Condition expression [Control Flow]**

  This expression is evaluated to decide whether the action is executed. If the expression or script returns false, the current action will be skipped. This expression or script should not have any side-effects, it will be called while another screen is still being displayed.

- **Rollback barrier [Control Flow]**

  If the action should be a rollback barrier. When a rollback barrier is completed, none of the preceding actions will be rolled back. You can use this property to prevent an incomplete rollback of complex changes or to protect actions from rollback when the user hits "Cancel" in the post-install phase.

- **Exit code [Control Flow]**

  If the "Rollback barrier" property is selected, and a rollback terminates at this action, this property determines the exit code of the installer. By default, reaching a rollback barrier during a rollback is considered a success, but you can signal a failure by specifying a non-zero exit code here.

  This property is only visible if "Rollback barrier" is selected.

- **Can be executed multiple times [Control Flow]**

  If the action can be executed multiple times. If unselected, the action will only be executed once and do nothing for subsequent invocations of the containing screen. The default settings for screens ensure that a screen with actions that cannot be executed multiple times is only shown once. However, if the "Back button" property is changed of if you skip screens programmatically, a screen might be shown multiple times.

- **Failure strategy [Error Handling]**

  If an action fails (i.e. returns `false`), the installer or uninstaller can continue, quit, or ask the user what to do. If you select something other than "Continue on failure", you should enter an error message in the "Error message" property unless the action displays the error itself.

  For "Return to the parent screen", no further actions will be executed and the previous screen will be displayed again. If the action is contained in the "Startup" node, the first screen will be shown and in unattended mode the application will quit.

- **Error message [Error Handling]**

  If the action fails, this error message is displayed to the user, otherwise the action fails silently.

Available actions

The following standard actions are available in install4j:

Category: Control

Change cancel button state

Changes the visibility and the enabled state of the cancel button. This action works in GUI mode as well as in unattended mode when the `-splash` option has been passed on the command line and the simple unattended progress dialog with a cancel button is shown.

Run script

Runs a custom script. The script must return a boolean value. If it returns false, the installation will be canceled.

Set a variable

Sets a variable by running a custom script. The script can return any `java.lang.Object`.

Set messages

Sets the messages in the progress interface.

Set the progress bar

Change the value of the progress bar or set it to indeterminate mode.

Sleep

Sleep a specified number of milliseconds. This is useful to ensure that a progress screen is displayed for at least a certain period of time.

Category: Desktop integration

Add a desktop link

Create a link on the desktop to an installed executable or file. This action will be automatically reverted by the 'Uninstall files' action.

Add a startup executable on Windows and macOS

Add an installed executable to the startup folder on Windows or to the login items on macOS so that it will be started automatically when the user logs in. This action will be automatically reverted by the 'Uninstall files' action.

Add an executable to the dock

Add an installed executable to the dock on macOS. This action will be automatically reverted by the 'Uninstall files' action.

Create a Windows URL link

Create a URL link on Windows. This is a special text file with a .url link that is supported by the Windows desktop, start menu and explorer. To create links in the start menu, the "Create program group" action can be used as well. This action will be automatically reverted by the 'Uninstall files' action.

Create a file association

Create an association between a file extension and a launcher, so that the launcher is invoked when the user double-clicks a file with the selected extension.

On Windows and Linux, if the application has not yet been started, the arguments to the main method will contain the file name. Subsequent invocations and all invocations on macOS can be intercepted with the `com.install4j.api.launcher.StartupNotification` class. This action will be automatically reverted by the 'Uninstall files' action.

Create program group

Create standard program group entries on Windows and freedesktop.org compatible UNIX desktops. This action will be automatically reverted by the 'Uninstall files' action.

Create start menu entry

Create a single start menu entry on Windows and Unix. For creating multiple program group entries, please see the "Create program group" action. This action will be automatically reverted by the 'Uninstall files' action.

Register Add/Remove item

Register an Add/Remove item in the Windows software registry. This action will be automatically reverted by the 'Uninstall files' action.

Register a URL handler

Register a URL handler for a custom scheme, so that the launcher is invoked when the user clicks on a link with the specified scheme.

On Windows and Linux, the arguments to the main method will contain the URL. On macOS, the arguments are available from the `com.install4j.api.launcher.StartupNotification` class. If the "Allow only a single running instance of the application" check box is selected on the "Java invocation" step of the launcher wizard, subsequent invocations are intercepted by the `com.install4j.api.launcher.StartupNotification` class on all platforms.

This action will be automatically reverted by the 'Uninstall files' action.

Category: File operations

Add Windows file rights

Adds access rights to files and directories on Windows.

If a helper process with elevated privileges has been created by the "Request privileges" action, this action is pushed to the helper process. Please see the help topic on "Elevation Of Privileges" for more information.

Copy files and directories

Copy files and directories. This action will be automatically reverted by the 'Uninstall files' action.

Create a symbolic link

Creates a symbolic link. This action has no effect on Windows.

Delete files and directories

Deletes files and directory. Directories can be deleted recursively.

Move files and directories

Moves files and directories. The newly created files are subject to removal by the 'Uninstall files' action.

Set the UNIX access mode of files and directories

Sets the UNIX access mode of files and directories. This action has no effect on Windows.

Set the modification time of files

Sets the modification time of files.

Set the owner of files and directories

Sets the owner and optionally the group of files and directories. This action has no effect on Windows.

Category: Final options

Execute launcher

Execute an installed launcher and return immediately. This action is intended to be placed on the "Finish" screen. A confirmation can be added automatically to the "Finish" screen.

If the main installation process has been elevated by the "Request privileges" action, this action is pushed to the original process with limited rights. Please see the help topic on "Elevation Of Privileges" for more information.

Open PDF viewer

Displays a PDF file in a cross-platform PDF viewer. A separate window will be opened.

Reboot computer

Reboot the computer on Windows and macOS. This action will trigger a reboot that takes place at the end of installation or uninstallation. By default, the user will be asked whether to reboot or not.

Show URL

Show a URL in the default browser. This action is intended to be placed on the "Finish" or the "Uninstallation success" screen.

If the main installation process has been elevated by the "Request privileges" action, this action is pushed to the original process with limited rights. Please see the help topic on "Elevation Of Privileges" for more information.

Show file

Show a file with the associated application. Usually, a text file or an HTML file is appropriate. This action is intended to be placed on the "Finish" screen. A confirmation can be added automatically to the "Finish" screen.

If the main installation process has been elevated by the "Request privileges" action, this action is pushed to the original process with limited rights. Please see the help topic on "Elevation Of Privileges" for more information.

Category: HTTP and network

Download file

Download a URL and save it to a file

HTTP request

Make an HTTP request to a specified URL. All common HTTP request methods are supported for REST calls. For mime types starting with `text` or containing "charset" information, the response body can be saved to an installer variable. To download large files, use the "Download file" action instead.

The action will succeed if a HTTP response code in the 2xx range is received, otherwise it will fail. You can save the response code to a variable to inspect it in a later action.

Upload file

Upload a file to an HTTP server with a POST request.

Wait for HTTP server

Wait until an HTTP or HTTPS port becomes available. This is useful if you start a server, for example with a "Start a service" action, and need to wait until the server is operational before proceeding with the installation.

Wait for Socket

Wait until a socket can be connected to. This is useful if you start a non-HTTP server. For HTTP and HTTPS, use the "Wait for HTTP server" action instead.

Category: JDBC

Check JDBC connection

Check if a connection can be made to the configured JDBC database. If no connection can be made, the action will fail. If the action is attached to a form screen that queries a database location, set its "Error message" property to an appropriate error message and the "Failure strategy" property to "Return to the parent screen".

Execute SQL query

Execute a single SQL query and store the result in an installer variable. If only the first row is taken, the row value is stored directly, otherwise the variable will contain an instance of `java.util.List` with the row values. If the query is for a single column, the row value is the Java object representation of the return type, e.g. `java.lang.String` for `VARCHAR` or `java.lang.Long` for `INT`.

Execute SQL script

Execute a single SQL statement or a script of SQL statements.

JDBC container action

This action allows you to configure connection properties just once and then execute a list of JDBC actions with the same connection.

Category: Java preference store

Delete a node or key in the Java preference store

Delete an entire package node or a key-value pair in the Java preference store.

Load installer variables from the Java preference store

Load installer variables from the Java preference store that have been previously saved by the "Save installer variables to the Java preference store" action.

Read a key from the Java preference store

Read the value of a key from the Java preference store and save it to an installer variable. Only string values can be read.

Save installer variables to the Java preference store

Save installer variables to the Java preference store. This can be used to communicate installer variables to the uninstaller or to installers with different application IDs.

Set a key in the Java preference store

Set a key-value pair in the Java preference store. The package node is created if necessary. This is the most convenient way to communicate settings to related installers. Only string values can be set.

Category: Miscellaneous

Add VM options

Adds VM options for a launcher by modifying or creating a `.vmoptions` file or by changing the Info.plist file. This action will be automatically reverted by the 'Uninstall files' action.

Check for running processes

Check for installed launchers and additional running processes on Windows and macOS.

Modify an environment variable on Windows

Sets, appends to, prepends to or removes an environment variable on Windows. This action can be automatically reverted by the 'Uninstall files' action.

Modify classpath

Changes the classpath of a launcher by modifying or creating a `.vmoptions` file or by changing the Info.plist file. This action will be automatically reverted by the 'Uninstall files' action.

Request privileges

Requests configurable administrator privileges. On Windows Vista and higher and on macOS, the installer will be restarted with the requested privileges or a helper process will be created that can perform certain actions in a privileged context. When you restart the installer, you should not install files before this action.

Please see the help topic on "Elevation Of Privileges" for a detailed discussion of this action.

Require installer privileges

Require the same privileges as the ones that were obtained during the installation. On Windows Vista and higher and on macOS, the uninstaller or custom installer application will be restarted with the requested privileges if necessary. This action only has an effect if a "Load response file" action is executed previously.

Please see the help topic on "Elevation Of Privileges" for a detailed discussion of this action.

Run executable or batch file

Runs an executable or a Windows batch file. The action can optionally wait for termination of the executable.

Category: Persistence of installer variables

Create a response file

Create a response file at an arbitrary location to save user input for subsequent installations. This file can be used with the `-varfile` command line option.

Load a response file

Load a response file that has previously been saved with the "Create a response file" action.

Modify a response file

Update all variables in an existing response file. The action does not delete variables in the response file for which no installer variables are defined, but keeps them as they are.

This action is useful for updating a response file from a custom installer application, where not all installer variables are available.

Category: Properties files

Read a properties file

Read a properties file and save a `java.util.Map` object with the properties to an installer variable. If you use a "Write properties to file" action to write the variable back to disk, the comments on the existing property definitions will be preserved.

Remove keys from properties file

Remove selected keys from a properties file. The line separator of the properties file is conserved.

Write properties to file

Write property definitions to a properties file. The properties can come from an installer variable with a `java.util.Map` object, another properties file or from direct entry.

If the "Merge into existing file" property is selected, the new property definitions will be added to the existing ones.

Category: Services

Install a service

Installs a service. On Windows, this is done by executing the service launcher with the appropriate arguments. On Unix, if systemd is detected, a config file will be created in `/etc/systemd/system`, otherwise a link will be placed in `/etc/init.d`. On macOS, a LaunchDaemon will be created. This action will be automatically reverted by the 'Uninstall files' action.

If a helper process with elevated privileges has been created by the "Request privileges" action, this action is pushed to the helper process. Please see the help topic on "Elevation Of Privileges" for more information.

Start a service

Starts a service by executing the service launcher with the appropriate arguments.

If a helper process with elevated privileges has been created by the "Request privileges" action, this action is pushed to the helper process. Please see the help topic on "Elevation Of Privileges" for more information.

Stop a service

Stops a service by executing the service launcher with the appropriate arguments.

If a helper process with elevated privileges has been created by the "Request privileges" action, this action is pushed to the helper process. Please see the help topic on "Elevation Of Privileges" for more information.

Category: Text files

Fix line feeds

Changes the line feeds of text files to the platform specific type.

Modify text files

Modify installed text files by replacing a search value in the selected files. This action does not read the entire file into memory and can work on arbitrarily large text files.

Modify text files with regular expressions

Modify installed text files by applying a regular expression.

Read text from file

Read the content of a text file and save it to an installer variable. The variable value will be of type `String`.

Replace installer variables in text files

Modify installed text files by replacing all occurrences of installer variables of the form `${installer:myVariable}` with their current values. The action also replaces i18n variables like ${i18n;myKey} and compiler variables like `${compiler:myCompilerVariable}`

Write text to a file

Write text to a new file or append text to an existing file.

Category: Update

Check for update

Load the update descriptor from the a URL and save it to the a variable. If successful, the variable will contain an instance of `com.install4j.api.UpdateDescriptor`

Schedule update installation

Schedule a downloaded media file to be started upon the next start of a launcher configured accordingly or by calling UpdateChecker.executeScheduledUpdate().

Shut down calling launcher

Shut down the launcher that called this application if it was started with the `com.install4j.api.launcher.ApplicationLauncher` API.

Category: Windows registry

Add access rights for a key in the Windows registry

Add access rights for a key in the Windows registry.

If a helper process with elevated privileges has been created by the "Request privileges" action, this action is pushed to the helper process. Please see the help topic on "Elevation Of Privileges" for more information.

Delete a key or value in the Windows registry

Delete a key or value in the Windows registry.

Read a value from the Windows registry

Read a value from the Windows registry and save it to an installer variable. The type of the value depends on the type in the registry, it will be an instance of one of the following classes: `String, Integer, String[], byte[], WinRegistry.ExpandString`.

Set a value in the Windows registry

Set a value in the Windows registry. This action can also create the appropriate key if necessary.

Category: XML files

Apply an XSLT transform

Transform an installed file by applying an XSLT stylesheet.

Count nodes in XML file

Count the occurrences of an XPath expression in an XML file and save the result to an installer variable.

Insert XML fragment into XML files

Insert an XML fragment into the position defined by an XPath expression. The fragment can replace an existing element node, or it can be inserted as a child or a sibling.

Read value from XML file

Read a string value from an XML file specified by an XPath expression and save the result to an installer variable.

Remove nodes from XML files

Remove selected nodes from XML files by specifying an XPath expression.

Replace text in XML files

Modify installed XML files by selecting nodes with an XPath expression and applying a regular expression on the selected values.

Category: ZIP files and archives

Create a ZIP file

Create a ZIP file from the specified source files and directories.

Extract a DMG file on macOS

Extracts the content of a DMG file to an arbitrary location on macOS.

Extract a TAR file

Extracts the content of a tar or tar.gz file to an arbitrary location.

Extract a ZIP file

Extracts the content of a ZIP file to an arbitrary location.

Install content of a ZIP file

Installs the content of an external ZIP file to an arbitrary location. This action will be automatically reverted by the 'Uninstall files' action.

Modify a ZIP file

Modify the contents of a ZIP file with a configurable list of actions.

Download and install component

Download a specified downloadable component and install it. This action only works for installation components that have been marked as "downloadable" on the "Options" tab of the installation component configuration.

Note: The "Install Files" action already downloads and installs all selected downloadable installation components. This action is intended for scenarios where an installation component has to be downloaded after the "Install files" action has run. For example, you could use this in a custom installer application to install optional files.

Execute previous uninstaller

Uninstalls the previous installation of this application in the selected installation directory by executing the previous uninstaller.

Install files

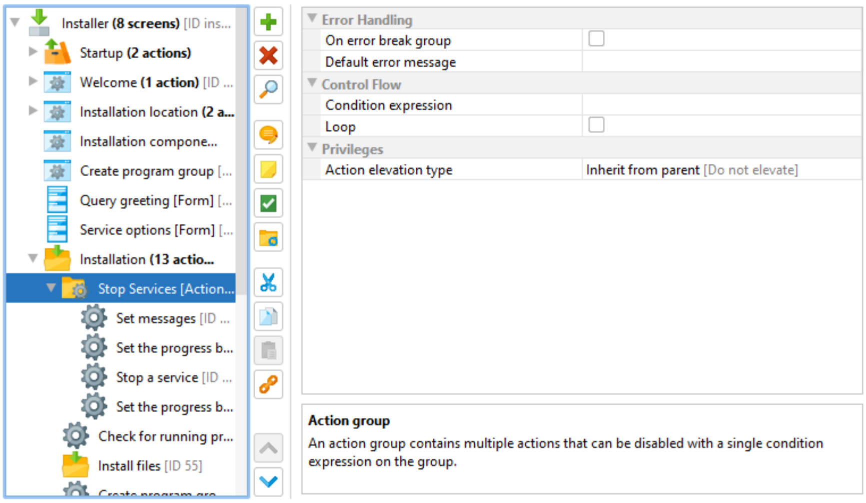
Install all files in the distribution tree that are contained in the selected installation components.

Uninstall files

Uninstall all installed files.

B.6 Configuring Screens And Actions Groups

Screen and action groups can be configured on the "Installer->Screens & Actions" step [p. 148].



Error Handling	
On error break group	☐
Default error message	
Control Flow	
Condition expression	
Loop	☐
Privileges	
Action elevation type	Inherit from parent [Do not elevate]

Action group

An action group contains multiple actions that can be disabled with a single condition expression on the group.

Actions and screens can be grouped in the tree of installer elements. Groups of the same type can be nested, meaning that you can put a screen group into a screen group or an action group into an action group.

You can nest as many levels of groups as you wish. Next to the label of the screen or action group in the tree of installer elements the number of all contained screens or actions is shown in bold where elements in nested groups are counted as well.

Grouping offers the following benefits:

- **Organization**

  If you have many screens or actions, groups emphasize which elements belong together. You can add a common comment to the group.

- **Common condition**

  Groups have a "Condition expression" property that allows you to skip the group with a common condition instead of having to repeat the condition for each contained element.

- **Single link target**

  If you want to reuse a set of adjacent screens or actions in a different part of your project, you can put them in a group and add a single link to that group instead of linking to each element separately.

- **Looping**

  A group has a "Loop expression" property that allows you to execute the group repeatedly until the loop expression returns `false`.

- **Jump targets (screen groups only)**

  When you jump to a screen programmatically with `context.gotoScreen(...)`, it is more maintainable to jump to a group instead of to a single screen. You can think of the group as a label in this case.

Properties of screen and action groups

The common properties of screen and action groups are:

- **Condition expression [Control Flow]**

  This expression is evaluated just before the screen is displayed. If the expression or script returns `false`, the entire screen group will be skipped.

- **Loop [Control Flow]**

  If selected, the screen group will be looped. With the child properties you can set an expression that terminates the loop and configure a loop index that is available inside the loop.

  Note: If actions should be repeated in a loop, their "Can be executed multiple times" property has to be selected. If form components in a screen should be re-initialized on each loop, their "Reset initialization on previous" property has to be selected.

- **Loop expression [Configuration]**

  This expression is evaluated when the end of the screen group is reached. If it returns `true`, all screens will be repeated. If you leave the expression empty, no loop will be performed.

  This property is only visible if "Loop" is selected.

- **Loop index start value [Configuration]**

  The start value for the loop index variable that is passed to the "Loop expression"

  This property is only visible if "Loop" is selected.

- **Loop index step [Configuration]**

  The step for the loop index variable that is passed to the "Loop expression". At the end of each loop, this step is added to the loop index. It is added before the "Loop expression" is evaluated. To decrement, specify a negative value.

  This property is only visible if "Loop" is selected.

- **Loop index variable name [Configuration]**

  If you want to use the loop index in a screen that is contained in the group, you can optionally save the value to an installer variable. Specify the variable name to which the value should be saved as a `java.lang.Integer`.

  This property is only visible if "Loop" is selected.

- **Style [GUI Options]**

  The default screen style for this installer application. Screens and screen groups can override this style.

- **Action elevation type [Privileges]**

  If any contained actions should run in the elevated helper process, if their "Action elevation type" property is set to "Inherit from parent".An elevated helper process is available on Windows and macOS if the process has been started without admin privileges and the "Request privileges" action has been configured to require full privileges.

In addition, action groups have the following properties:

- **On error break group [Error Handling]**

  If selected, and one of the contained actions returns with an error, the control flow will step out of the action group and continue with the next element after the group. This behavior only takes effect if the problematic action has its failure strategy set to "Continue on failure".

- **Error message [Configuration]**

  If the action group fails, this error message is displayed to the user, otherwise the action group fails silently.

  This property is only visible if "On error break group" is selected.

- **Failure strategy [Configuration]**

  The failure strategy that should be chosen if the action group fails. The "Error message" property will be used for the option dialog. If you also define a "Default error message", you will get two option dialogs, the first one from the action that causes the failure.

  This property is only visible if "On error break group" is selected.

- **Retry expression [Configuration]**

  If this expression is set and returns `true`, the action group is repeated. If the action group is configured to loop, the loop index will not be incremented.

  This property is only visible if "On error break group" is selected.

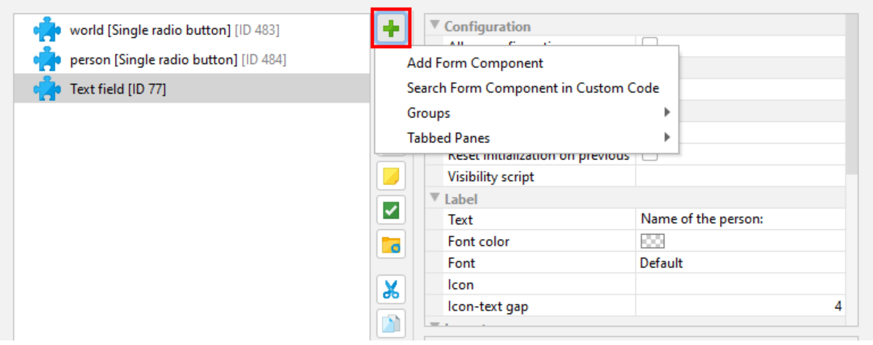
- **Default error message [Error Handling]**

  A default error message used by all actions that have no dedicated error message.

B.7 Configuring Form Components

Form components are configurable units that can be added to a form screen. In this chapter, the functionality and configuration options on the form components dialog are discussed, the underlying concepts are discussed in a different help topic [p. 46].

Form elements are added by clicking the ✚ *Add* button.


world [Single radio button] [ID 483]
person [Single radio button] [ID 484]
Text field [ID 77]
Add Form Component
Search Form Component in Custom Code
Groups
Tabbed Panes
Configuration
Reset initialization on previous
Visibility script
Label

Text	Name of the person:
Font color	▨
Font	Default
Icon	
Icon-text gap	4

In the popup window you can select whether to add

- a form component. Form components are made available by install4j or are contributed by an installed extension [p. 218]. A registry dialog will be shown where you can select the desired form component.
- a form component that is contained in your custom code. New types of reusable form components can be developed with the install4j API [p. 212]. In your custom code configuration [p. 152] you can specify code locations that are scanned for suitable classes. A class selector will be shown where you can select the desired class.
- a layout group [p. 189], either a vertical group or a horizontal group. The new layout group is initially empty. You can also create layout groups directly from a selection in the tree of installer elements.

You can preview a form screen with the ◉ *Preview* button which is also available on the property page of a screen. For screens that embed forms, the preview may not show the actual screen. However, the layout of the form itself will be the same at runtime.

Properties of form components

Common properties of form components are:

- **Insets [Layout]**

  This insets around the form component. The format is top;left;bottom;right, use the drop-down button at the right side to show the insets editor.

- **Initialization script [Initialization]**

  A script that initializes the form component. To configure the contained principal component, such as a JCheckBox, use the configurationObject parameter (if available). This script will run after the internal initialization of the form component, just before the component appears on the screen. It will not be invoked in console mode.

- **Reset initialization on previous [Initialization]**

  If set, the component will be initialized each time the user enters in the forward direction. Otherwise, the initialization will be performed only once. This setting affects both the internal initialization as well as the initialization script.

- **Visibility script [Initialization]**

  A script that determines whether the form component will be visible or not. This works for both GUI and console modes. In GUI mode, the script will be invoked each time just before the form component is initialized.

Available form components

The following standard form components are available in install4j:

Category: Action components

Button

A standard button with an optional leading label. When the user clicks on the button, an action script is executed.

Dark mode switcher

A button that switches between dark and light mode. If the current look and feel does not support switching between dark and light mode, the button is invisible.

Hyperlink URL label

A label that displays a hyperlink. When the user clicks on the hyperlink, the appropriate action is performed, depending on the protocol of the URL.

Hyperlink action label

A label that displays a hyperlink. When the user clicks on the hyperlink, an action script is executed

Category: Labels and spacers

Horizontal separator

A horizontal separator with an optional label.

Key value pair label

A pair of labels. The first ('key') label aligns with other leading labels on the form, the second ('value') label consumes the remaining horizontal space,

Label

A single label. It is left-aligned with leading labels from other form components and extends beyond other leading labels.

Leading label

A form component that only has a leading label and no central component. This can also be used to create standalone help tooltips.

Multi-line HTML label

A multi-line label that wraps text as needed and displays simple HTML. In particular you can include HTML links that open a browser.

Multi-line label

A multi-line label that wraps text as needed.

Spring

An invisible spring that can be used in horizontal and vertical layout groups to push subsequent components to the right or to the bottom

Vertical spacer

An invisible vertical spacer of configurable height.

Category: Option selectors

Check box

A check box with an optional leading label. The user selection (`Boolean.TRUE` or `Boolean.FALSE`) is saved to a variable.

Combo box

A combo box with an optional leading label. The user can enter arbitrary text into the combo box. The user selection (the selected item as a string) is saved to a variable.

Drop-down list

A drop-down list with an optional leading label. The user selection (the selected index as a `java.lang.Integer`) is saved to a variable.

List

A list with an optional leading label. The user selection (the selected indices) is saved to a variable.

Radio button group

A number of radio buttons in a common button group with an optional leading label. The user selection (the selected index as a `java.lang.Integer`) is saved to a variable.

Single radio button

A single radio button with an optional leading label. If selected, a specified string is saved to a variable. If you place multiple instances of this form component on a form screen and give them the same variable name, they will form a radio button group.

Category: Sliders and spinners

Slider

A slider with an optional leading label. The user input (a `java.lang.Integer`) is saved to a variable.

Spinner of dates

A spinner with date and time values with an optional leading label. The user input is saved to a variable.

Spinner of enumerated values

A spinner with enumerated values with an optional leading label. The user input is saved to a variable.

Spinner of integer values

A spinner with integer values with an optional leading label. The user input is saved to a variable.

Category: Special selectors and displays

Directory chooser

A directory chooser with an optional leading label. The user selection is saved to a variable.

File associations selector

A form component that displays a list of all subsequent file association actions and asks the user which associations should be made. This form component will be empty if there are no corresponding file association actions after this screen.

File chooser

A file chooser with an optional leading label. The user selection is saved to a variable.

HTML or text display

A scroll panel that displays HTML or plain text. The HTML or plain text is easily localizable because the file selection allows you to enter separate files for all supported languages.

Installation components selector

A form component that displays all installation components and asks the user which components should be installed.

Installation directory chooser

An installation directory chooser with an optional display of required and free space. The user selection is set as the installation directory.

License agreement

A form component that displays a license agreement to the user, either plain text or HTML. The license agreement must be accepted before the next screen can be shown.

Log file viewer

A text area that shows the contents of a text file. The the viewer follows additions to the file like the UNIX command `tail -f`, with a configurable maximum number of displayed lines.

The log file does not have to exist when the form is shown, it can be created later on. Also, the file can be deleted and re-created. Modifications before the previously observed end of the file will not be picked up by the viewer unless the length of the file decreases.

PDF display

Displays a PDF file in an embedded cross-platform PDF viewer.

Program group selector

A form component that allows the user to select a program group on Microsoft Windows.

Progress display

An progress display that can show the progress of the actions attached to the containing screen.

Update alert

A pair of radio buttons offering the user a choice whether to update an existing installation or not. If the existing installation should be updated, the installer variable sys.confirmedUpdateInstallation is set to `true`. Several standard screens use that installer variable in their default condition expression.

Update schedule selector

Drop-down box that lets the user select an update schedule for your application. You can use the `com.install4j.api.update.UpdateScheduleRegistry` class in your application to check if you should launch an updater. Please see the Javadoc for more information. Please note that simply adding this form component does not automatically launch an updater at regular intervals.

Windows user selector

A component for selecting Windows users or groups in the native Windows user dialog. Optionally, you can display a button to create a new user. The selection is saved as a SID (1) to a string variable. If multiple selection is enabled, the result is a string array of SIDs.

This component does not do anything in console mode, since it requires the native Windows dialog for selecting users and groups.

Category: Text fields

Password field

A password text field with an optional leading label. The user input is displayed with '*' characters. The user input is saved to a variable.

Text area

A text area with an optional leading label. The user input is saved to a variable.

Text field

---

(1) https://en.wikipedia.org/wiki/Security_Identifier

A text field with an optional leading label. The user input is saved to a variable.

Text field with date format

A text field with an optional leading label and a date format. The user input (a `java.util.Date`) is saved to a variable.

Text field with format mask

A text field with an optional leading label and an arbitrary format mask. The user input is saved to a variable. The default mask is that of an SSN. For more information, please see the javadoc of `javax.swing.text.MaskFormatter`.

Text field with integer format

A text field with an optional leading label and an integer format. The user input is saved to a variable with type `java.lang.Long`.

Text file editor

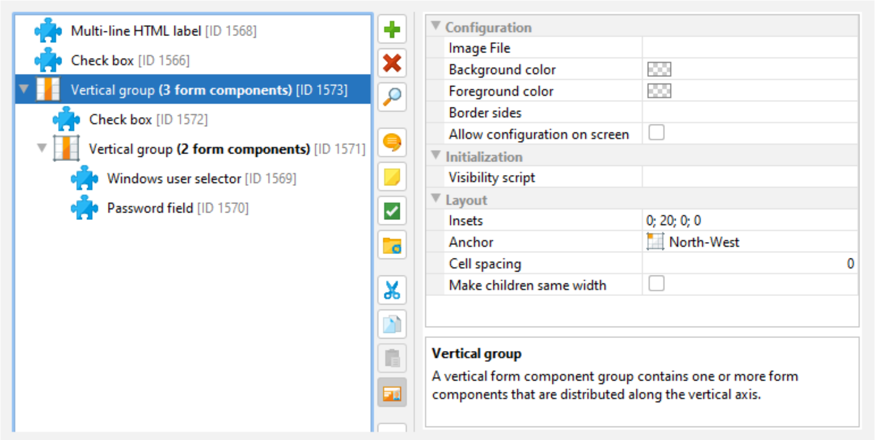
A text area for editing a file. If the file does not exist, a configurable initial text is presented to the user and the file is created. The file is saved when the user clicks on the "Next" button.

Console handler

Allows you to interact with the user in a console installer. All standard form components expose appropriate behavior in console mode, however, there are situations where you need to fine-tune your console installer with additional messages or questions. In GUI or unattended mode, this form component does not have any effect.

B.8 Configuring Layout Groups

Layout groups can be configured in the form components [p. 183] configuration dialog. This chapter discusses the configuration options for layout groups, for more information on layout groups, see the corresponding help topic [p. 51].



- Multi-line HTML label [ID 1568]
- Check box [ID 1566]
- Vertical group **(3 form components)** [ID 1573]
  - Check box [ID 1572]
  - Vertical group **(2 form components)** [ID 1571]
    - Windows user selector [ID 1569]
    - Password field [ID 1570]

Configuration	
Image File	
Background color	▨
Foreground color	▨
Border sides	
Allow configuration on screen	☐
Initialization	
Visibility script	
Layout	
Insets	0; 20; 0; 0
Anchor	North-West
Cell spacing	0
Make children same width	☐

Vertical group

A vertical form component group contains one or more form components that are distributed along the vertical axis.

You can create a layout group [p. 189] from selected form components with the *Create Horizontal Group* and *Create Vertical Group* actions. The new group will be inserted in place of the selected elements.

You can dissolve a group with the *Dissolve Group* action. This action is only enabled if the selection consists of a single layout group. The elements contained in the group will be inserted in place of the group. Nested groups will not be dissolved.

Grouping features

Form components can be grouped in horizontal and vertical layout groups and you can nest groups to an arbitrary depth.

Grouping offers the following benefits:

- **Custom layout**

  Instead of a simple sequence of form components on a form screen, you can use horizontal layout groups to put form components side-by-side. Nesting vertical and horizontal form components allows you to achieve virtually any layout.

  Sometimes, enclosing groups and sibling groups create a cell that cannot be entirely filled by a layout group. With the **"Anchor"** property you determine where the group should be placed in that case. By default, horizontal layout groups are anchored at "West" and vertical layout groups are anchored at "North-West".

  Layout groups have a **configurable cell spacing**. For vertical layout groups, this is the vertical gap between two form components (0 pixels by default), for horizontal layout groups this is the horizontal gap between two adjacent form components (5 pixels by default)

  For each layout group, you can specify **insets** that are inserted around the entire layout group. By default, the insets are zero in all directions.

  By default, a horizontal layout group aligns a leading label of its first form component with the leading label of the first form component from a direct vertical parent group. This is usually appropriate when horizontal groups are used to attach additional form components to the right side. If this alignment is not desired, you can use the "Align first label" property of a horizontal layout group to switch off the alignment.

Vertical layout groups always break the alignment of leading labels: Within a vertical group, leading labels are aligned, but between vertical groups, the width of leading labels is unrelated.

- **Organization**

  If you have many form components on a screen, vertical groups emphasize which form components belong together. You can add a common comment to the group.

- **Common visibility script**

  Groups have a "Visibility script" property that allows you to hide the entire group with a common condition instead of having to repeat the condition for each contained form component.

- **Single target for coupled form components**

  If a set of form components should be coupled to the selection state of a check box or a single radio button, you can select the containing layout group as the target instead of selecting all coupled form components separately.

- **Styling**

  Layout groups have properties for setting background images and borders, as well as background and foreground colors. Styles [p. 193] use layout groups to achieve visual effects.

Properties of layout groups

Common properties of horizontal and vertical layout groups are:

- **Image File [Configuration]**

  An image that is shown on the edge or as a background. Apart from an image that is anchored to the center of the group, the image can optionally cut off an entire edge from the group. In that case it is possible to set a background color for the edge stripe so that the image can blend into the surroundings. Can be empty.

  To add a high-resolution image, create a file with double the resolution and an additional `@2x` after the name (e.g. `image.png` and `image@2x.png`) next to the selected image. To use different images in dark mode, add files with an additional `_dark` suffix (e.g. `image_dark.png` and `image@2x_dark.png`)

  The install4j runtime JAR file `i4jruntime.jar` contains a number of image files that you can reference here by prefixing the icon file name with "icon:". For example, `icon:lock_open_32.png` loads a 32x32 icon showing an open lock.

- **Image anchor [Configuration]**

  The anchor where the image will be attached to in the layout group. If Center is chosen, the image is always displayed in the background.

- **Image edge [Configuration]**

  For corner anchors, you have to select either the horizontal or the vertical edge that will optionally be filled with the image edge background color and that will be cut of from the layout group if the image is not displayed in the background.

- **Image edge background color [Configuration]**

  The background color that the image edge should be filled with. If the image terminates with the same color, the image will blend with the background and the entire edge will look like a single visual element.

  Not available if the anchor is set to "Center"

- **Image edge border [Configuration]**

  If selected, the image edge will be separated by a line border from the content area.

  Not available if the image overlaps the contained components.

- **Image edge border color [Configuration]**

  The color of the image edge border. Leave empty to choose the default separator color of the current look and feel.

  This property is only visible if "Image edge border" is selected.

- **Image edge border width [Configuration]**

  The width of the image edge border in pixels.

  This property is only visible if "Image edge border" is selected.

- **Image insets [Configuration]**

  The insets around the image. The format is top;left;bottom;right, use the drop-down button at the right side to show the insets editor.

- **Overlap with contained components [Configuration]**

  If selected, the image will by used as a background image and form components contained in the layout group will overlap with the image. Otherwise, the image edge will be cut off from the layout group and form components will not overlap with the image. In that case, the insets of the layout group will be applied to the actual content area that excludes the image edge.

  Not available if the anchor is set to "Center"

- **Background color [Configuration]**

  The background color of the layout group. Can be empty.

- **Foreground color [Configuration]**

  The foreground color of the layout group. Can be empty. If set, all contained form components will use this foreground color except those that have an explicitly configured foreground color.

- **Border sides [Configuration]**

  On which sides a line border should be painted around the form component, a list of "top", "right", "bottom" and "left", separated by semicolons. Use the drop-down button to select the sides visually.

- **Border color [Configuration]**

  The color of the drawn border sides. Leave empty to choose the default separator color of the current look and feel.

- **Border title [Configuration]**

  A title that is displayed in the top-left corner of the border. Leave empty if no title should be displayed.

- **Border width [Configuration]**

  The width of the drawn border sides in pixels.

- **Visibility script [Initialization]**

  A script that determines whether form components in the group (and all descendant components in nested groups) will be visible or not. This works for both GUI and console modes. In GUI mode, the script will be invoked each time just before the form components are initialized. Visibility scripts of nested form components can further hide single form components, but they cannot show them if a parent layout group is already hidden.

- **Insets [Layout]**

  The insets around the entire group. The format is top;left;bottom;right, use the drop-down button at the right side to show the insets editor.

- **Anchor [Layout]**

  The position in the available space where the group is anchored in the layout. This is only relevant if the group takes less space than the cell that is created by the surroundings.

- **Cell spacing [Layout]**

  The cell spacing determines how many pixels are inserted between single components in the layout group.

Vertical layout groups have the additional properties:

- **Make children same width [Layout]**

  If all contained elements should have the same width.

and horizontal layout groups have the following specific properties:

- **Align first label [Layout]**

  If the horizontal group is directly added to a vertical group or to the top-level of a form, the leading label in the horizontal group is aligned with other leading labels in the vertical parent group. If this alignment is not desired, you can deselect this property.

- **Make children same height [Layout]**

  If all contained elements should have the same height.

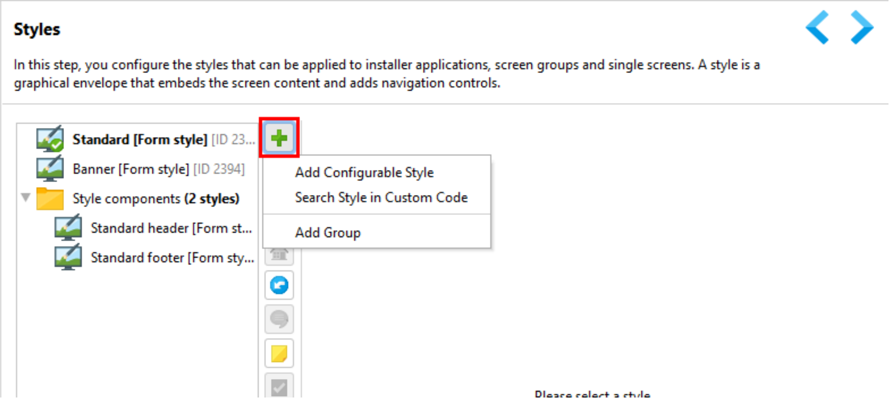
Tabbed panes

In addition to horizontal and vertical layout groups, you can add **tabbed panes** to a form. A tabbed pane is added by choosing *Tabbed Panes->Add Tabbed Pane* from the dropdown menu displayed by the ✚ *Add* button. Below the tabbed pane, you have to add one or more single tabs by choosing *Tabbed Panes->Add Single Tab For Tabbed Pane*. Each single tab can then contain arbitrary form components or layout groups.

B.9 Configuring Styles

Styles determine how screens look like in GUI installers. For more information on styles, see the corresponding help topic [p. 55].

Styles are added by clicking the *Add* button.



Styles

In this step, you configure the styles that can be applied to installer applications, screen groups and single screens. A style is a graphical envelope that embeds the screen content and adds navigation controls.

Standard [Form style] [ID 23...
Banner [Form style] [ID 2394]
Style components (2 styles)
Standard header [Form st...
Standard footer [Form sty...
Add Configurable Style
Search Style in Custom Code
Add Group
Please select a style

In the popup window you can select whether to add

- a configurable style. Styles can be constructed with a restricted set of the form components [p. 183] for screens that do not take user input and some special form components that are relevant in a styling context.
- a style that is contained in your custom code. New types of reusable styles can be developed with the install4j API [p. 212]. In your custom code configuration [p. 152] you can specify code locations that are scanned for suitable classes. A class selector will be shown where you can select the desired class.
- a group for organizing styles, so you have a better overview of which styles belong together.

For organizing styles in your project, you can create a group from selected styles with the *Create group from selection* action and dissolve groups with the *Dissolve Group* action. This action is only enabled if the selection consists of a single layout group. The elements contained in the group will be inserted in place of the group. Nested groups will not be dissolved.

You can preview a style with the *Preview* button which is also available on the property page of a style.

Properties of styles

Form styles have the following properties:

- **Standalone style**

  If selected, the style can be selected for installer applications, screen groups and screens. If a style is not standalone, it can only be used in other styles.

- **Fill horizontal space**

  If selected, all available horizontal space is filled by this style. This setting is also used when it is nested in another style by a "Nested style" form component.

- **Horizontal anchor**

  If "Fill horizontal space" is not selected, the style can be placed at different locations in the available space.

  This property is only visible if "Fill horizontal space" is selected.

- **Fill vertical space**

  If selected, all available vertical space is filled by this style. This setting is also used when it is nested in another style by a "Nested style" form component.

- **Vertical anchor**

  If "Fill vertical space" is not selected, the style can be placed at different locations in the available space.

  This property is only visible if "Fill vertical space" is selected.

C Generated Installers

C.1 Installer Modes

Installers generated by install4j can be run in three modes:

- **GUI mode**

  The default mode for installer applications is to display a GUI installer or uninstaller.

- **Console mode**

  If the installer application is invoked with the `-c` argument, the interaction with the user is performed in the terminal from which the installer was invoked.

- **Unattended mode**

  If the installer is invoked with the `-q` argument, there is no interaction with the user and the installation is performed automatically with the default values.

The flow of screens and action sequence is executed in the same way for all three modes. If some actions or screens should not be traversed for console or unattended installations, you can set their "Condition expression" properties to

```
!context.isConsole()
```

or

```
!context.isUnattended()
```

GUI mode

In GUI mode, the keyboard shortcut `CTRL-SHIFT-L` shows the log file in the Explorer on Windows, in the Finder on macOS and in the file manager on Linux/Unix. This shortcut is not advertised to the user, but you can communicate it to the user for debug purposes.

Console mode

Installers generated by install4j can perform console installations, unless this feature has been disabled in the application configuration [p. 154] of the "Installer->Screens & Actions" step. In order to start a console installation, the installer has to be invoked with the `-c` argument.

All standard screens and form components in install4j present their information on the console and allow the user to enter information as in the GUI installer. Not all messages in the style are displayed in the console installer. By default only the subtitle of a screen is displayed as the first message, but you can change this behavior with the "Console screen change handler" script of the installer application.

The subtitle is appropriate to display in in console mode, because all standard screens in install4j have a question as their subtitle. If you add your own forms to the screen sequence [p. 148], you should phrase their subtitles as questions in order to create a consistent user experience for the console installer.

On Windows, the information of whether an executable is a GUI executable or a console executable has to be statically compiled into the executable. Installers are GUI executables, otherwise a console would be displayed when starting the installer from the explorer. This is also the reason why the JRE supplies both the `java.exe` console executable and the `javaw.exe` GUI executable on Windows.

However, a GUI executable can attach to a console from which it was started. GUI executables are started in the background by default, which means that you have to use the `start` command to put it in the foreground and be able to enter information:

```
start /wait installer.exe -c
```

If you develop new screens or form components, you have to override the method

```
boolean handleConsole(Console console) throws UserCanceledException
```

to implement the behavior for console mode. Displaying default data on the console and requesting user input is made easy with the `Console` class that is passed as a parameter.

Unattended mode

Installers generated by install4j can perform unattended installations, unless this feature has been disabled on the application configuration [p. 154] of the "Installer->Screens & Actions" step. In order to start an unattended installation, the installer has to be invoked with the `-q` argument. The installer will perform the installation as if the user had accepted all default settings.

There is no user interaction on the terminal. In all cases, where the installer would have asked the user whether to overwrite an existing file, the installer will not overwrite it. You can change this behavior by passing `-overwrite` as a parameter to the installer. In this case, the installer will overwrite such files. For the standard case, it is recommended to fine-tune the overwrite policy in the distribution tree [p. 14] instead, so that this situation never arises.

The installer will install the application to the default installation directory, unless you pass the `-dir` parameter to the installer. The parameter after `-dir` must be the desired installation directory, for example:

```
installer.exe -q -dir "D:\MyApps\My Application"
```

For the unattended mode of an installer, response files [p. 202] are an important instrument to pre-define user input.

On Windows, the output of the installer is not printed to the command line for unattended installation. If you pass the `-console` parameter after the `-q` parameter, the executable will try to connect to the invoking console and display output to the user. This is useful for debugging purposes.

If the installation was successful, the exit code of the installer will be `0`, if no suitable JRE could be found it will be `83` and for other types of failures it will be `1`.

If you develop new screens or form components, you have to override the method

```
boolean handleUnattended()
```

in order to support unattended installations.

C.2 Command Line Options For Generated Installers

Installers generated by install4j recognize the following command line parameters:

Name	Explanation
-h or -help or /?	Show help for common command line parameters. This will be shown in a message box, regardless of the default execution mode. If the GUI display fails, it will be printed on the console.
-manual	This option only applies to Windows. In GUI mode, the default JRE search sequence [p. 204] will not be performed and bundled JREs will not be used either. The installer will act as if no JRE has been found at all and display the dialog that lets you choose a JRE or download one if a JRE has been bundled dynamically. If you locate a JRE, it will be used for the installed application. On Unix, you can define the environment variable INSTALL4J_JAVA_HOME_OVERRIDE instead to override the default JRE search sequence.
-c	Executes the installer in console mode [p. 195].
-q	Executes the installer in unattended mode [p. 195].
-g	Forces the installer to be executed in GUI mode. This is only useful if the default execution mode [p. 154] of the installer has been configured as console mode or unattended mode.
-console	If the installer is executed in unattended mode and `-console` is passed as a second parameter, status messages will be printed on the console from which the installer was invoked.
-overwrite	Only valid if `-q` is set. In the unattended installation mode, the installer will not overwrite files where the overwrite policy [p. 14] would require it to ask the user. If `-overwrite` is set, all such files will be overwritten. The default value for this option can be changed with the system property `-Dinstall4j.quietOverwrite=true`
-nofilefailures	Only valid if `-q` is set. In the unattended installation mode, the installer will not fail if an error occurs during a file installation. The default value for this option can be changed with the system property `-Dinstall4j.noFileFailures=true`
-wait <timeout in seconds>	Only valid if `-q` is set. In unattended installation mode, the installer will perform the installation immediately. On Windows, this can lead to locking

Name	Explanation
	errors if the installer is called by an updater or by a launcher. If -wait is specified, the installer application will wait until all installed launchers and installer applications (including the updater) have shut down. If this does not happen within the specified timeout, the installer application exits with an error message.
-dir <directory>	Only valid if `-q` is set. Sets a different installation directory for the unattended installation mode. The next parameter must be the desired installation directory. The directory can be absolute or relative. If it is relative, it will be resolved relative to the media file.
-splash <title>	Only valid if `-q` is set. Instead of being completely quiet in unattended installation mode, a small window with a progress bar and the specified title will be shown to inform the user about the progress of the installer application. This is useful if you start the installer application programmatically and do not require user input.
-alerts	Only valid if `-q` and `-splash` are set. By default, in unattended mode, no alerts are shown. This includes messages boxes, error alerts and questions. By setting this command line parameter, alerts are enabled for unattended executions with a progress dialog.
-temp <directory>	Change the temporary directory for the installer application on Windows. An installer may extract a lot of files and it also extracts executables to its temporary directory. If the default temporary directory of the system is not suitable for this purpose, you can change the directory with this parameter. The specified directory must exist and must be writable. This is useful for trouble-shooting problems caused by anti-virus software.
-Dinstall4j.nolaf=true	Do not set the native look and feel but use the default. In some rare cases, the native look and feel is broken and prevents the use of the installer or any other Java GUI application.
-Dinstall4j.debug=true	By default, install4j catches all exceptions, creates a "crash log" and informs the user about the location of that log file. This might be inconvenient when debugging an installer, so this system property switches off the default mechanism and lets exceptions be printed to stderr.

Name	Explanation
-Dinstall4j.log=<path>	install4j creates a log file prefixed with `i4j_log` in the temporary directory when an installer application is executed. This log file can be helpful for debugging purposes. If your installer contains an "Install files" action and terminates successfully, the log file is copied to `<installation dir>/.install4j/installation.log`, otherwise it will be deleted after the installer application terminates. With the `-Dinstall4j.log=<path>` the log file will be written to the file specified with `<path>` instead and will not be deleted in any case. If a relative path is specified, it will be resolved relative to the installer media file for installers and relative to the working directory for uninstallers and custom installer applications.
-Dinstall4j.keepLog=true	As an alternative to `-Dinstall4j.log=<path>`, you can ask the installer or the installer application to not delete the temporary log file under any circumstances. For situations where you cannot modify the command line arguments, you can set the environment variable `INSTALL4J_KEEP_LOG=true`.
-Dinstall4j.logTimestamps=true	If set, each message in the log file is prepended with a time stamp.
-Dinstall4j.logToStderr=true	In addition to the log file created by the installer application, you can duplicate all log messages to stderr with this argument.
-Dinstall4j.logEncoding=<character set name>	By default, the installer will write the log file in the default encoding of the system where the installer is running. Should you wish to choose a different encoding you can pass this VM parameter to the installer. Some common character set names are • UTF-8 • UTF-16 • ISO-8859-1 The class `java.nio.charset.StandardCharsets` lists the encodings that are guaranteed to be available in any JRE.
-Dinstall4j.suppressStdout=true	In unattended mode, status messages of actions that are displayed in the installer are printed on stdout. To suppress these messages, you can set this VM parameter.

Name	Explanation
-Dinstall4j.detailStdout=true	In unattended mode, detailed messages regarding file installations are not printed on stdout. To enable these messages, you can set this VM parameter.
-Dinstall4j.suppressUnattendedReboot=true	In unattended mode, a reboot may be undesirable. To prevent reboots, you can set this VM parameter.
-Dinstall4j.language=<ISO code>	Overrides the language selection for a multi-language installer. The language selection dialog will not be displayed in this case, unless the specified language is not included in the installer.
-Dinstall4j.helperDebugPort=<port>	Debugging the installer application can be done by passing `-agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=<port>` on the command line, on Windows this argument has to be prefixed with `-J`. However, this will not debug the elevated helper process that is started by the "Request privileges" action. By setting the `install4j.helperDebugPort` VM parameter, the same `-agentlib` parameter is passed to the JVM of the helper process and you can then attach to it with a debugger. If you debug both the unelevated and the elevated JVM at the same time, you have to assign different ports and start two separate debugging sessions.
-Dsun.locale.formatasdefault=true	Forces the installer locale to be detected from the "Format" language setting and not from the "Display language" setting in the Windows "Region and Language" control panel.
`-J<VM parameter>`	Specifies a VM parameter, for example `-J-Xmx512m`. Can be specified more than once.
-D`propertyName=value`	You can set further arbitrary system properties with standard command line parameters. There is no need to prefix them with `-J` on Windows.
-V`variableName=value`	You can set arbitrary installer variables with the -V parameter. If you pass `-VvariableName=value`, you can use the variable value by inserting `${installer:variableName}` in text fields in the install4j IDE. The variable value will be a `java.lang.String` object.
-varfile <fileName>	Instead of repeatedly using the >`-V` command line option, you can specify a property file containing the variables you want to set. This option shares the same mechanism with response files [p. 202].

On macOS, you can use the INSTALL4J_ARGUMENTS environment variable to pass arguments to the installer.

On Unix, the environment variable INSTALL4J_TEMP determines the base directory for self-extraction. If the environment variable is not set, the parent directory of the installer media file is used.

C.3 Response Files

With a response file, you can change the default user selection in all screens. A response file is a text file with name-value pairs that represent installer variables. All screens and form components provided by install4j ensure that user input is bound to appropriate installer variables that are registered for being written to the response file.

Installer variable values are of the general type `java.lang.Object`. In a response file, only variables with values of certain types can be represented: In addition to the default type `java.lang.String`, the types `java.lang.Boolean`, `java.lang.Integer`, `java.util.Date`, `java.lang.String[]` and `int[]` are supported.

In order to let the installer runtime know about these non-default types, the variable name in the response file is followed by a '$' sign and an encoding specifier like 'Integer' or 'Boolean'.

Response file variables are variables that have been registered with

```
String variableName = ...;
context.registerResponseFileVariable(variableName);
```

in the installer. All variables that are bound to form components are automatically registered as response file variables. Also, system screens register response file variables as needed to capture user input.

All installer variables live in the same name space. If you use an installer variable more than once for different user inputs, the response file only captures the last user input. If you would like to optimize your installers for use with a response file, you have to make sure that the relevant variable names are unique within your installer.

A response file can be used to

- Configure the installer for unattended execution mode
- Change the default settings in the GUI and console installer
- Get additional debugging information for an installation

When applying a response file to an installer, all variable definitions are translated into installer variables [p. 63]. The response file shares the same mechanism with the variable file offered by the -varfile [p. 197] command line option. You can add the contents of a response file to a variable file and vice versa.

Generating response files

There are two ways to generate a response file:

- A response file is generated automatically after an installation is finished. The generated response file is found in the `.install4j` directory inside the installation directory and is named `response.varfile`. When you request debugging information from a user, you should request this file in addition to the installer log file.
- install4j offers a "Create a response file" action [p. 169] that allows you to save the response file to a different file in addition to the automatically generated response file. Here, you can also specify variables that you would not like to be included in the response file.

Applying response files

When an installer is executed, it checks whether a file with the same name and the extension **.varfile** can be found in the same directory and loads that file as the response file. For example, if an installer is called `hello_setup.exe` on Windows, the response file next to it has to be named `hello_setup.varfile`.

You can also specify a response file explicitly with the -varfile [p. 197] installer option.

Response files work with all three installer modes [p. 195], GUI, console and unattended.

Response file variables

The variables that you see in the response file are realized as installer variables as soon as the response file is loaded. You can use these installer variables to access or change user selections on system screens. For example, the "Create program group" screen on Windows binds the user selection for the check box that asks the user whether to create the program group for all users to the variable `sys.programGroup.allUsers`. To access the current user selection from somewhere else, you can use the expression

```
context.getBooleanVariable("sys.programGroup.allUsers")
```

To change that selection, you can invoke

```
context.setVariable("sys.programGroup.allUsers", Boolean.FALSE)
```

C.4 How Installers Find A JRE

Installers generated by install4j are native executables or shell scripts and can start running without a JRE. However, the installer itself requires a JRE in order to perform its work and so the first action of the installer is to locate a JRE that is suitable for both the installer and your application. In this process it performs the following steps:

1. Look for a **statically bundled JRE**. If a statically bundled JRE is included with the installer, it will unpack it and use it. First, this JRE is unpacked to a temporary directory, later it is copied to a location that depends on whether the bundled JRE is configured as shared or not.

   - **Not shared**

     It is copied to the `jre` directory in the installation directory of your application. No other installer generated by install4j will find this JRE. It will not be made publicly available, for example in the Windows registry.

   - **Shared**

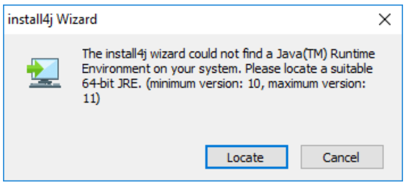
     The JRE is copied to the `i4j_jres` directory in a common folder which depends on the operating system:

     - `%CommonProgramFiles%` on Windows, which typically resolves to `C:\Program Files\Common Files` with an English locale.
     - `/opt` if it exists, otherwise `/usr/local` on Unix.

     If the above folder is not writable, the `i4j_jres` directory will be created in the use home directory and the shared JRE will only be shared for the current user.

     Other installers generated by install4j will find this JRE. It will not be made publicly available. For each Java version, only one such JRE can be installed. Shared JREs are never uninstalled.

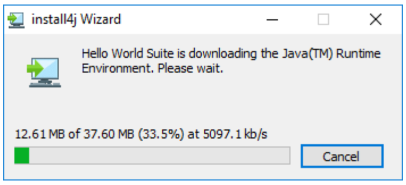
2. Look for a suitable JRE in the configured **search sequence**. The installer uses the same search sequence and Java version constraints as your launchers which are configured for the entire project [p. 36]. The most important search sequence element in this respect is the "Search Windows registry and standard locations" entry. On Windows, the registry contains information on installed JREs, on Unix platforms there is a number of standard locations which are checked, on macOS the location of installed JREs is always the same.
3. If no JRE has been found, the installer notifies the user

   The install4j wizard could not find a Java(TM) Runtime Environment on your system. Please locate a suitable 64-bit JRE. (minimum version: 10, maximum version: 11)

   and offers the following options:

   - Download a dynamically bundled JRE as configured in the Bundled JRE [p. 89] step of the media wizard [p. 126].



Hello World Suite is downloading the Java(TM) Runtime Environment. Please wait.

12.61 MB of 37.60 MB (33.5%) at 5097.1 kb/s

- Manually locate a JRE
- Cancel the installation

You can force the installer to skip the first two steps and show this dialog immediately with the `-manual` command line parameter [p. 197].

C.5 HTTP Requests

Actions that perform HTTP requests

install4j includes several actions that can perform HTTP or HTTPS requests:

- The "Install files" action downloads installation components that have been marked as "Downloadable" provided that the data files option has been set to "Downloadable" as well in the media file wizard.
- The "Check for updates" action downloads the update descriptor `updates.xml` from the specified web server in order to check if there is a new version available.
- The "Download file" action downloads the specified file from the web server.
- the "Upload file" action uploads a specified file with a POST request.
- The "HTTP request" action performs generic HTTP requests.
- The "Wait for HTTP server" action waits until a specified HTTP or HTTPS port becomes available.

When creating an HTTP/HTTPS connection to the requested resource there are three different concerns that may require user interaction: Proxy selection, proxy authentication and server authentication.

Proxy selection and authentication

On Windows, installer applications use native code to perform HTTP requests, so the native Windows proxy dialog will be shown. The proxy configuration of the operating system is used and the system properties for setting an HTTP proxy in Java do not apply. This has the advantage that a previously saved proxy password does not have to be entered by the user.

On other platforms, HTTP requests are made through the Java HttpClient for Java 11+ or a URLConnection for lower Java versions. If a proxy can be auto-detected from the system settings, it is used automatically. If the proxy requires credentials, an authentication dialog will be shown. User input in this dialog will be cached for the duration of the process. If the proxy uses basic authentication then HTTPS connections can only be tunneled if the VM parameter

```
-Djdk.http.auth.tunneling.disabledSchemes=
```

is set with an empty value as shown above. This is done automatically for installer applications, but not for generated launchers where you would have to set this VM parameter explicitly. If you do that, you should read about its security impact (1) in case you develop your own implementation of `java.net.Authenticator`.

Entering proxy data is supported in console mode as well. In unattended mode there is no user interaction, so the proxy information has to be provided to the installer via command line arguments. The following system properties for proxy configuration can be used:

```
-DproxyHost=<host name>
-DproxyPort=<port number>
```

If the proxy requires credentials, you also have to specify

```
-DproxyAuthUser=<user name>
-DproxyAuthPassword=<password>
```

---

(1) https://bugzilla.redhat.com/show_bug.cgi?id=1386103

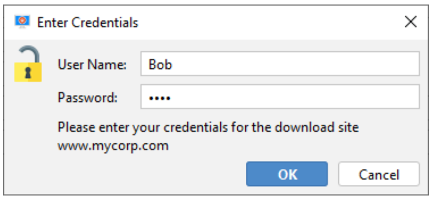
Except for the native Windows network connection, the above properties can also be used to configure the proxy from outside. Furthermore the global Java proxy properties

```
-Dhttp.proxyHost=<host name>
-Dhttp.proxyPort=<port number>
-Dhttp.proxyUser=<user name>
-Dhttp.proxyPassword=<password>
```

and the corresponding properties with the "https" prefix are also used for HTTP and HTTPS connections respectively. If you would like to use these properties on Windows as well, you can disable the native Windows network connection with the system property `-Dinstall4j.noWinInetConnection=true`.

Server authentication

The download URL can be password protected with basic HTTP authentication. In this case, the user has to supply a user name and a password.



User Name: Bob

Password: ••••

Please enter your credentials for the download site www.mycorp.com

Neither the user name nor the password is cached by install4j. In unattended mode you have to pass the arguments

```
-DserverAuthUser=<user name>
-DserverAuthPassword=<password>
```

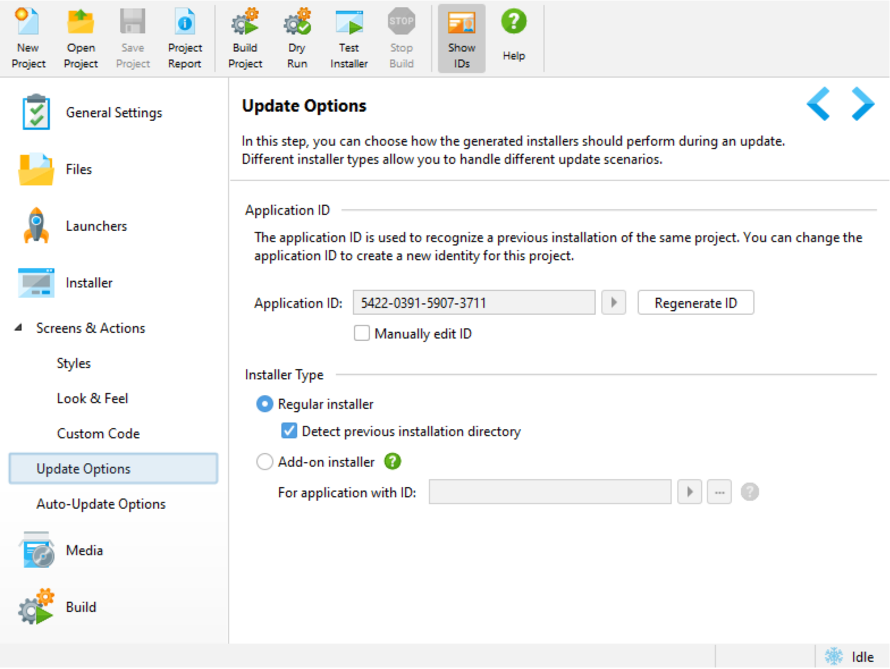
You can set these system properties via

```
System.setProperty("serverAuthUser", "<user name>");
System.setProperty("serverAuthPassword", "<password>");
```

programmatically.

C.6 Updates

On the "Installer->Update Options" step, you can configure how an installer should behave in the event of an update. An update occurs when the user installs an application into a directory where an installation with the same application ID already exists.



Update Options

In this step, you can choose how the generated installers should perform during an update. Different installer types allow you to handle different update scenarios.

Application ID

The application ID is used to recognize a previous installation of the same project. You can change the application ID to create a new identity for this project.

Application ID: 5422-0391-5907-3711

☐ Manually edit ID

Installer Type

● Regular installer

☑ Detect previous installation directory

○ Add-on installer

For application with ID:

Typically, minor upgrades of an application should be installed into the same directory as earlier installations. The default behavior of install4j is to suggest the previous installation directory and program group, so that the user is guided into installing the application into the same directory. If this behavior is not desired, you can switch off these suggestions or change the application ID on the "Installer->Update Options" step.

Updates into the same installation directory

The following points are of interest with respect to updates into the same installation directory:

- Generated installers will refuse to install on top of installations with a different application ID by default. You can change this behavior with the "Validate application id" property of the installation directory chooser on the "Installation location" screen.
- Generated installers will detect if any of the previously installed launchers are still running and will ask the user to shutdown these applications. This happens when the "Install files" action or a "Check for running processes" action is executed.
- Deployed services will be stopped and uninstalled before the installation. This happens when the "Install files" action is executed. You can optionally stop your services earlier with the "Stop a service" action if your update process requires it.
- During an update, the installation databases will be merged, so that files, menu entries, file associations and other modifications from old installations can still be uninstalled when the uninstaller is executed.
- After an update, only the uninstall actions of the newer installation will be executed when the uninstaller is executed. However, the auto-uninstall actions from previous installations will be executed, too, for example the uninstallation of a service that was registered by an "Install service" action during the installation.

If you would like to uninstall the previous installation before installing any new files, you can add the "Execute previous uninstaller" action before the "Install files" action. In this context, the uninstallation policies [p. 14] that exclude updates are important. With these uninstallation policies you can preserve certain files for updates, but uninstall them when the user manually invokes the uninstaller. The uninstaller invoked by the "Execute previous uninstaller" action is running in unattended mode. You can use

```
!context.isUninstallForUpgrade()
```

to exclude certain actions for an update uninstaller.

Add-on installers

install4j offers two types of installers that can be selected on the "Installer->Update options" step:

- **Regular installers**

  This option generates standalone installers. If the "Detect previous installation directory" check box is selected and a previous installation can be detected on the computer, the installer will suggest the directory of that previous installation. In that case, the "Update alert" form component on the "Welcome" screen will ask the user if the previous installation should be updated.

- **Add-on installers**

  This generates an installer that can **only** be installed on top of an installation of a certain installation. An add-on installer does not have a separate uninstaller. This is useful to distribute additional files that do not change the version number of the installation.

  If the add-on installer type is selected, you have to specify the application ID for the base application.

C.7 Error Handling

Debugging on Windows

On Windows, when an installer is executed it always generates a log file in the temp directory that contains information about the JRE search sequence and can be used for debugging purposes. The name of the log file starts with `i4j_nlog_`. If you have a problem with JRE detection or the installer startup, send this log file along with your support request.

It is also possible to generate this native debug log file for the generated Windows launchers. In order to switch on logging, define the environment variable

```
INSTALL4J_LOG=yes
```

and look for the newest text file whose name starts with `i4j_nlog_` in the temp directory. This is done silently, without notifying the user and is also suitable for situations where launchers are called automatically or repeatedly.

An easier way for a user to create a log file is to start the launcher with the argument

```
/create-i4j-log
```

The launcher will notify the user where the log is created and will offer to open an explorer window with the log file selected. After the message box is closed, the launcher will continue to start up.

Debugging on macOS

Similar to Windows, macOS launchers also support the `INSTALL4J_LOG=yes` environment variable definition for debug logging. Rather than writing a log file, they write to the system log. You can display the system log by starting the "Console" application which is located in `/Applications/Utilities`.

Setting the environment variable can be done by opening a terminal and executing

```
launchctl setenv INSTALL4J_LOG=yes
```

Then all newly started applications in the Finder will have this environment variable set. The current terminal will not be affected until you quit the Terminal application and start it again.

Rather than setting the environment variable for all install4j launchers, you can set it for a particular invocation only. To do that, call the `Contents/MacOS/JavaApplicationStub` inside the application bundle and prefix the call with the definition of the environment variable. For an application bundle "MyApp.app", the call looks like this:

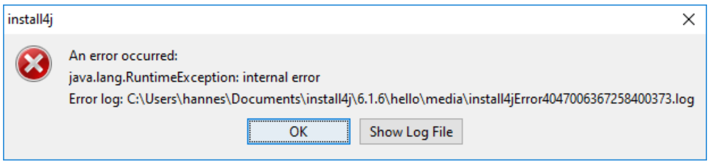
```
INSTALL4J_LOG=yes MyApp.app/Contents/MacOS/JavaApplicationStub
```

In this case, the log output will also be written to the terminal. Using `/usr/bin/open` will not work with this technique, because the latter gets the environment variables from the Finder.

Note that logging only works for GUI launchers and not for command line and service launchers which are implemented as Unix shell scripts. There is no command line argument that activates logging, like on Windows.

Error logs

If an exception is thrown in the installer, it prepares an error log and informs the user about its location



```
An error occurred:
java.lang.RuntimeException: internal error
Error log: C:\Users\hannes\Documents\install4j\6.1.6\hello\media\install4jError4047006367258400373.log
```

You can force the installer to print exceptions to stderr for debugging purposes with the `-Dinstall4j.debug=true` command line option [p. 197].

Installation log

All installer applications generate an installation log that can be used for debugging purposes. After a successful installation the log file is saved to

```
<installation dir>/.install4j/installation.log
```

For an uninstaller or if the installer exited before the "Install files" action was run, you can find it in the temporary directory if you pass `-Dinstall4j.keepLog=true` to the installer or uninstaller. The file is prefixed `i4j_log`.

If you would like the installer to log to stderr as well, you can pass `-Dinstall4j.logToStderr=true` to the installer. Both arguments can also be useful for debug installers and uninstallers, where they have to be passed as VM parameters.

Error handling of Actions

You can define the error handling for every installation or uninstallation action separately. Mor information is available in the DMG options and files on screens and actions [p. 24].

Return values

The process of an installer returns `0` if the installation was completed successfully, `1` if the installation fails and 83 if the installer could not find a suitable JVM to run. These exit codes are useful when checking the result of an unattended installation [p. 195].

D API

D.1 API For Installer Applications

There are two different use cases where the install4j API is required: Within expression/script properties [p. 29] in the configuration GUI and for the development of custom elements in install4j. The development of custom elements in install4j is rarely necessary for typical installers, most simple custom actions can be performed with a "Run script" action and most custom forms can be realized with a "Customizable form" screen.

If you would like use your IDE while writing more complex custom code, you can put a single call to custom code into expression/script properties. The location of your custom code classes must be configured on the "Installer->Screens & Actions->Custom Code" step, so install4j will package it with the installer and put in into the class path. In this way you can completely avoid the use of the interfaces required to extend install4j.

Expression/script properties

Using expression/script properties in install4j is required for wiring together screens and actions [p. 24] as well as for the conditional execution of screens and actions. The most important element in this respect is the **context** which is an instance of

- **com.api.install4j.context.InstallerContext**

  in an installer

- **com.api.install4j.context.UninstallerContext**

  in an uninstaller

The context allows you to query the environment and the configuration of the installer as well as to perform some common tasks.

See the documentation of the com.install4j.api.context package for the complete documentation of all methods in the context. Some common applications include:

- **Setting the installation directory**

  By using `context.setInstallationDirectory(File installationDirectory)` in the installer context, you can change the default installation directory for the installer. Typically, this call is placed into a "Run script" action on the "Startup" screen.

- **Getting and setting installer variables**

  The `getVariable(String variableName)` and `setVariable(String variableName, Object value)` methods allow you to query and modify installer variables. Note that besides the "Run script" action, there is also a "Set a variable action" where you don't have to call `setVariable` yourself.

- **Conditionally executing screens or actions**

  Often, condition expressions for screens and actions check the values of variables. In addition, the context provides a number of boolean getters that you can use for conditionally executing screens and actions depending on the installer mode and environment. These methods include `isConsole()`, `isUnattended()` and others.

- **Navigating between screens**

  Depending on the user selection on a screen, you might want to skip a number of screens. The `goForward(...)`, `goBack(...)` and `goBackInHistory(...)` methods provide the easiest way to achieve this.

Many other context methods are only useful if you develop custom elements for install4j.

Also have a look at the `com.install4j.api.Util` class which offers a number of utility methods that are useful in expression/script properties.

Development environment

To develop custom elements in your IDE, you have to add the install4j API to the compilation class path. The entire install4j API is contained in the single artifact with maven coordinates

```
group: com.install4j
artifact: install4j-runtime
version: <install4j version>
```

where the install4j version corresponding to this manual is 9.0.7.

Jar, source and javadoc artifacts are published to the repository at

```
https://maven.ej-technologies.com/repository
```

You can either add the API to your development class path with a build tool like Gradle or Maven, or use the JAR file

```
resource/i4jruntime.jar
```

in the install4j installation.

To browse the Javadoc, go to

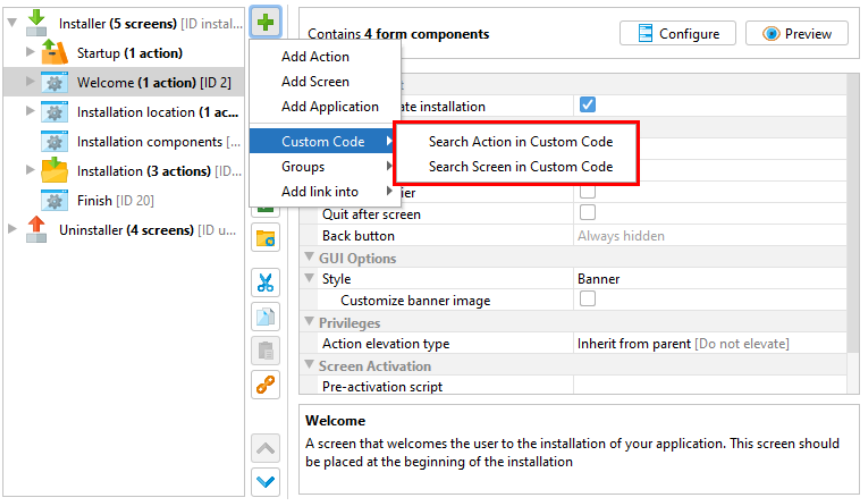
```
javadoc/index.html
```

For a general overview on how to start developing with the install4j API, how to set up your IDE and how to debug your custom elements, see the API overview in the javadoc.

Developing custom elements for install4j

install4j provides four extension points: actions, screens, form components and styles

All actions, screens and form components in install4j use this API themselves. To make your custom elements selectable in the install4j IDE, you first have to configure the custom code locations on the "Installer->Screens & Actions->Custom Code" step. When you add an action, screen or form component, the first popup gives you the choice on whether to add a standard element or search for suitable elements in your custom code.


Installer (5 screens) [ID instal...
Startup (1 action)
Welcome (1 action) [ID 2]
Installation location (1 ac...
Installation components [...
Installation (3 actions) [ID...
Finish [ID 20]
Uninstaller (4 screens) [ID u...
Add Action
Add Screen
Add Application
Custom Code
Groups
Add link into
Search Action in Custom Code
Search Screen in Custom Code
Contains 4 form components
Configure
Preview

...ate installation	☑
...ier	☐
Quit after screen	☐
Back button	Always hidden
GUI Options	
Style	Banner
Customize banner image	☐
Privileges	
Action elevation type	Inherit from parent [Do not elevate]
Screen Activation	
Pre-activation script	

Welcome

A screen that welcomes the user to the installation of your application. This screen should be placed at the beginning of the installation

If you want to ship your custom code to third parties, consider packaging an install4j extension [p. 218], which displays your custom elements alongside the standard elements that are provided by install4j and allows you to add dependency JAR files that are included in the installers if any of the contained elements are used in a project.

Serialization

install4j serializes all instances of screens, actions and form components with the default serialization mechanism for JavaBeans.

To learn more about JavaBeans serialization, visit

- https://docs.oracle.com/javase/8/docs/api/java/beans/XMLEncoder.html (1) for API documentation on the long-term persistence mechanism for JavaBeans.
- http://www.oracle.com/technetwork/java/persistence4-140124.html/ (2) for information on how to write your own persistence delegates. In your bean infos for screens, actions and form components you can specify a list of additional persistence delegates for non-default types. Writing custom persistence delegates will generally not be necessary unless you want to serialize special types from 3rd party libraries.

Compiler variables are replaced in the serialized representation of a bean. In this way, compiler variable replacement is automatically available for all properties of type `java.lang.String`. The values of installer variables and localization keys are determined at runtime, so you have to call the utility methods in `com.install4j.api.beans.AbstractBean` before you use the values in the installer or uninstaller. For more information on variables, see the separate help topic [p. 63].

Internationalization

install4j offers custom localization files in the install4j IDE to localize your own messages. `com.install4j.api.context.Context.getMessage(String key)` gives access to all messages.

If you develop your own user-configurable screens, actions or form components, you can replace all custom localization keys and installer variables in property values with calls to the `com.install4j.api.beans.AbstractBean.replaceVariables(...)` methods. All abstract base classes for beans extend `com.install4j.api.beans.AbstractBean`.

(1) https://docs.oracle.com/javase/8/docs/api/java/beans/XMLEncoder.html

(2) http://www.oracle.com/technetwork/java/persistence4-140124.html

The locale of the installer will always be set to the language selected by the user or configured for the media file, **not** the locale of the system that the installer is running on. You can call `com.install4j.api.context.Context.getLanguageId()` to find out what language your installer is running with.

Testing and debugging

To test and debug screens, actions and form components for your installer, enable the `Create additional debug launcher` build option in the "Build" section. After the build, your media file output directory will contain directories with the name `debug_[name of the media file minus extension]` for each media file that you have built.

The debug directories contain

- the Windows batch files `debug_installer.bat` and `debug_uninstaller.bat` for Windows media files
- the shell scripts `debug_installer.sh` and `debug_uninstaller.sh` for media files of Unix based platforms

These scripts start the installer and the uninstaller with a plain java invocation. All exceptions are directly printed to stderr and no separate error log files are created.

The file `user.jar` in the debug directory contains all your custom code. For interactive development you will not want to rebuild the project after each modification of your custom code. You can set up the installer or the uninstaller in your IDE by

- setting the working directory to the debug directory
- including your own code in the class path
- including i4jruntime.jar in the class path
- including user.jar in the class path. Your own code will also be contained in user.jar, but the IDE typically places project code at the beginning of the class path so it will override equivalent classes in user.jar.
- using the main class `com.install4j.runtime.installer.Installer` for the installer or `com.install4j.runtime.installer.Uninstaller` for the uninstaller
- passing the VM parameter `-Dinstall4j.debug=true`

Note that the working directory for the executed java process **must** be the debug directory, otherwise both the installer as well as the uninstaller will not work.

This procedure allows for an edit-compile-debug cycle that is much faster than building the media file and running the installer. In addition, output on stderr and stdout can be captured and you can debug your screens, actions and form components this way.

D.2 API For Generated Launchers

Generated launchers in install4j have some features that you can interact with from your own code. The corresponding API is contained in the `com.install4j.api.launcher` package. This chapter gives an overview of the most important use case, the detailed documentation is contained in the Javadoc.

install4j's launcher API is automatically available to an application deployed with install4j. For compiling your application, you have to add the runtime classes to your class path. You can learn how to set up a dependency in build systems in the API overview.

Receiving Startup Events in Single Instance Mode

If you have enabled the single instance mode [p. 36] for your executable, the application can only be started once. For a GUI application, the existing application window is brought to front when a user executes the launcher another time.

However, you might want to receive notifications about multiple startups together with the command line parameters. If you have associated you executable with a file extension, you will likely want to handle multiple invocations in the same instance of your application. Alternatively, you might want to perform some action when another startup occurs.

To do that, create a class that implements the `com.install4j.api.launcher.StartupNotification.Listener` interface and register it with `com.install4j.api.launcher.StartupNotification.registerStartupListener(listener)`. Your listener will then be notified when another startup occurs. See the Javadoc for more information.

Controlling the Splash Screen from your Application

If you have enabled a splash screen [p. 36] for a launcher, you will want to hide it once the application startup is finished. The splash screeb will be hidden automatically as soon as your application opens the first AWT, JavaFX or SWT window.

However, you might want to hide the splash screen programmatically by calling `com.install4j.api.launcher.SplashScreen.hide()` or update the contents of the status text line on the splash screen with `com.install4j.api.launcher.SplashScreen.writeMessage(...)` during the startup phase to provide more extensive feedback to your users. Also, if the UI subsystem is not loaded by the system class loader, install4j cannot automatically detect displayed windows and you have to hide the splash screen automatically. For example, this is the case for eclipse RCP applications.

Reading compiler and installer variables from response files

All installer variables that are registered for response files will be saved to the file `.install4j/response.varfile` just before the installer exits. This includes all variables that are bound to form components and variables for which you have called `context.registerResponseFileVariable(variableName)`.

Some of these variables will contain user input that you need at runtime. You can use the `com.install4j.api.launcher.Variables` class to access the variable values. The variable values from the response file are fixed and its backing file is usually not writable by the user. If you want to update the variable values at runtime, you can save variables to the preference store with a "Save installer variables to the preference store" action. The `com.install4j.api.launcher.Variables` class has methods for reading and saving these variables from the preference store.

In addition, all compiler variable values can be retrieved at runtime. See the Javadoc for detailed information.

Starting installer applications from your launchers

Installer applications like update downloaders are separate executables and can be started manually by the user. Most often, however, they will by launched by one of the generated launchers. install4j offers a configurable launcher integration mechanism that automatically executes an installer application when a launcher is started. For greater flexibility, you may want to execute the installer application from your code programmatically. On the "Installer->Screens & Actions" step, when an installer application is selected, the integration wizard on the "Launcher integration" tab produces code that uses the `com.install4j.api.launcher.ApplicationLauncher` class.

There are two ways to start installer applications: In-process and out-of-process. For an in-process invocation, the installer application will use the look and feel of your JVM. The AWT subsystem will be initialized which may be undesirable if you use a different UI toolkit like JavaFX. For greater isolation, out-of-process invocations are recommended. The ApplicationLauncher API offers both options. In both cases you can supply a callback that is notified when the installer application exits or if a "Shutdown calling launcher" action in the installer application request a shutdown of the launcher.

In addition, the `ApplicationLauncher` class provides a mechanism to run an installer application the first time a launcher from an archive installation is started. Archives do not have an installer, but you may still want to run some install4j actions, for example to configure a file association. With the `ApplicationLauncher.isNewArchiveInstallation()` method you can check at startup if this is the first time that the launcher is being executed.

D.3 Extensions

Introduction

All standard actions, screens and form components in install4j use the installer API [p. 212] themselves. With this API you can create new elements that are displayed in the standard registries by packaging a JAR file with a few special manifest entries and putting that JAR file into the `extensions` directory of your install4j installation.

Configurability

An extension to install4j will likely need to be configurable by the user. install4j uses the JavaBean specification (1) to control the user presentation of properties in the install4j IDE. Screens, actions and form components correspond to beans in this context.

Optionally, you can add BeanInfo classes. A BeanInfo class next to the bean itself describes which properties are editable and optionally gives details on how they should be presented. See the documentation of the com.install4j.api.beaninfo package for the complete documentation on how to develop BeanInfo classes. Also, `samples/customCode/src` in the installation directory contains sample beans with associated BeanInfo classes.

JAR manifest

In order to tell install4j which classes are screens, actions or form components, you have to use the following manifest keys:

- **Install-Action**

  for actions implementing `com.install4j.api.actions.InstallAction`
- **Uninstall-Action**

  for actions implementing `com.install4j.api.actions.UninstallAction`
- **Installer-Screen**

  for screens implementing `com.install4j.api.screens.InstallerScreen`
- **Uninstaller-Screen**

  for screens implementing `com.install4j.api.screens.UninstallerScreen`
- **Form-Component**

  for form components implementing `com.install4j.api.formcomponents.FormComponent`
- **Style-Component**

  for form components implementing `com.install4j.api.formcomponents.FormComponent` that should also be available in styles. Such form components should not take any user input because they will have a different life-cycle in styles than in screens.

Note that usually you do not implement these interfaces yourself, but rather extend one of the abstract base classes.

A typical manifest with one action and one screen looks like this:

---

(1) http://www.oracle.com/technetwork/articles/javaee/spec-136004.html

```
Depends-On: driver.jar common.jar

Name: com/mycorp/actions/MyAction.class
Install-Action: true

Name: com/mycorp/screens/MyScreen.class
Installer-Screen: true
Uninstaller-Screen: true
```

If you only have named sections and no global section in your manifest file, the first line must be an empty line since it separates the global keys from the named sections.

The `Depends-On` manifest key can specify a number of relative JAR files separated by spaces that must be included when the extension is deployed. That key can also occur separately for each named section.

As you see in the example for the screen, each class can have multiple keys if the appropriate interfaces are implemented.

Localization

Extensions can provide localized messages. During development, you can keep these messages in the custom localization file of the project that you use for testing purposes. When packaging the extensions, these custom localization files have to be given special names and be put into a particular location in the extension JAR file.

The names of the extension localization files have to be the same as those of the system localization files in the `resource/messages` directory, for example `messages_en.utf8` and similarly for other languages. The `java.util.Properties` file encoding is also supported if the file name has a .properties extension, like `messages_en.properties`.

When creating the extension JAR file, all extension localization files have to be put into the directory `messages`. No special directives in the manifest are required. Dependencies included with the `Depends-On` manifest key are not scanned for extension localization files.

Extension deployment

On startup, install4j will scan the manifests of all JAR files that it finds in the `extensions` directory. Any screens, actions or form components that are found in the manifests are added to the standard registries. If a bean cannot be instantiated, the exception is printed to stderr which is captured in `<temp directory>/install4j_error.log` and no further error is displayed.

If any of those screens, actions or form components are selected by the user, the required JAR files are deployed with the generated installers. This means that installing extensions does not create an overhead for installers that do not use them.

E Command Line Tools

E.1 Install4j Command Line Compiler

install4j's command line compiler `install4jc[.exe]` can be found in the `bin` directory of your install4j installation. It operates on project files with extension `.install4j` that have been produced with the install4j IDE. (`install4j[.exe]`). The install4j command line compiler is invoked as follows:

```
install4jc [OPTIONS] [config file]
```

A quick help for all options is printed to the terminal when invoking

```
install4jc --help
```

In order to facilitate usage of install4jc with automated build processes, the destination directory for the media files and the application version can be overridden with command line options. Furthermore you can achieve internationalization and powerful customizations with compiler variables [p. 63]. As a last resort, since the file format of install4j's config files is xml-based, you can achieve arbitrary customizations by replacing tokens or by applying XSLT stylesheets to the config file.

Options for the install4j command line compiler

The command line compiler has the following options:

- **-h or --help**

  Displays a quick help for all available options.

- **-V or --version**

  Displays the version of install4j in the following format:

  ```
  install4j version X.Y, built on YYYY-MM-DD
  ```

- **-v or --verbose**

  Enables verbose mode. In verbose mode, install4j prints out information about internal processes. If you experience problems with install4j, make sure to include the verbose terminal output with your bug report.

- **-q or --quiet**

  Enables quiet mode. In quiet mode, no terminal output short of a fatal error will be printed.

- **-t or --test**

  Enables test mode. In test mode, no media files will be generated in the media file directory.

- **-i or --incremental**

  Enables incremental test execution. A test installer [p. 11] for the current platform is updated with the latest screens, actions and form components and executed immediately. Because the files are taken from a previously built media file, the compilation is very fast.

- **-g or --debug**

  Create additional debug installers for each media file. For each built media file, a directory that is named like the media file will be created in the media file output directory.

- **-p or --preserve**

  Do not delete the temporary directory that the compiler uses for staging all files and launchers.

- **-w or --fail-on-warning**

  If a warning is printed and this option is specified, the build will fail at the end. It does not fail immediately, so you can see all warnings and fix them all at once. The exit code in this case is 2 instead of 1 for an actual error and 0 for a successful execution.

- **-n or --faster**

  Disable LZMA and Pack200 compression. If you have enabled LZMA or Pack200 compression on the "General Settings->Media File Options" step, this allows you to create development builds much faster, since LZMA and Pack200 are expensive compression algorithms.

- **-u or --disable-signing**

  Disable code signing. If you have configured code signing [p. 138], this allows you to skip code signing for a build. In that case you do not have to enter the passwords for the key stores.

- **-j or --disable-bundling**

  Disable JRE bundling. If you have configured JRE bundles [p. 89] for any media files, those bundles will not be used and the installer will be built without a contained JRE. This speeds up the build and the installation.

- **--win-keystore-password=<password>**

  Set the Windows keystore password for the private key that is configured for code signing [p. 138]. If code signing is enabled for Windows media files and this option is not set, the command line compiler will prompt you for the password.

- **--mac-keystore-password=<password>**

  Set the macOS keystore password for the private key that is configured for code signing [p. 138]. If code signing is enabled for macOS media files and this option is not set, the command line compiler will prompt you for the password.

- **--apple-id-password=<password>**

  Set the app-specific password for the Apple ID that is configured on the "General Settings->Code Signing" step. If notarization is enabled and this option is not set, the command line compiler will prompt you for the password. This option only has an effect on macOS, because notarization requires command line executables that are included in Xcode.

- **--disable-notarization**

  Disable notarization of media files on macOS. If you have enabled notarization for code signing [p. 138] and the build is running on macOS, this option allows you to skip notarization.

- **-L or --license=<key>**

  Update the license key on the command line and exit. This is useful if you have installed install4j on a headless system and cannot start the GUI. <key> must be replaced with your license key. If you use floating licenses, replace <key> with `FLOAT:server` where "server" is the host name or IP address where the floating license server is installed. For floating licenses, you can choose the requested edition by passing `--windows-edition` or `--multi-platform-edition`.

The config file that contains the license key has a platform-specific location:

- Windows: `%LOCALAPPDATA%\install4j\v<version>\config.xml`
- macOS: `~/Library/Application Support/install4j/v<version>/config.xml`
- Linux/Unix: `.config/install4j/v<version>/config.xml`, the root directory may be modified by the environment variable `XDG_CONFIG_HOME`

- **-r <string> or --release=<string>**

  Override the application version defined in the "General Settings->Application Info" step. `<string>` must be replaced with the actual version number. Version number components can be alphanumeric and should be separated by dots, dashes or underscores.

- **-d <string> or --destination=<string>**

  Override the output directory for the generated media files. `<string>` must be replaced with the actual directory. If the directory contains spaces, you must enclose `<string>` in quotation marks.

- **-s or --build-selected**

  Only build the media files which have been selected in the install4j IDE. By default, all media files are built regardless of the selection in the "Build" step.

- **-b <list> or --build-ids=<list>**

  Only build the media files with the specified IDs. `<list>` must be replaced with a comma separated list of numeric IDs. The IDs for media files can be shown in the install4j IDE by choosing *Project->Show IDs* from the main menu. Examples would be:

  ```
  -b 2,5,9
  --build-ids=2,5,9
  ```

- **-m or --media-types=<type>[,<type>]...**

  Only build media files of the specified type. `<type>` must be replaced with a media file type recognized by install4j. To see the list of supported media types, execute

  ```
  install4jc --list-media-types
  ```

  . Examples would be:

  ```
  -m win32,macos,macosFolder
  --media-types=win32,macos,macosFolder
  ```

- **-D <name>=<value>[,<name>=<value>]...**

  Override a compiler variable [p. 63] with a different value. You can override multiple variables by specifying a comma separated list of name value pairs. <name> must be the name of a variable that has been defined on the "General Settings->Compiler Variables" step. The value can be empty.

  To override a variable for a specific media file definition only, you can prefix <name> with `ID:` to specify the ID of the media file. The IDs for media files can be shown in the install4j IDE by choosing *Project->Show IDs* from the main menu.

Examples would be:

```
-D MYVARIABLE=15,OTHERVARIABLE=
"-D MYVARIABLE=15,OTHERVARIABLE=test,8:MEDIASETTITLE=my title"
```

A special system variable that you can override from the command line is `sys.languageId`. `sys.languageId` must be set to the ISO code of the language displayed in the language selection dialog and determines the principal installer language [p. 79] for the project or the media file.

- **-f <file> or --var-file=<file>**

  Load variable definitions from a file. This option can be used together with the `-D` option, which takes precedence if a variable occurs twice. The file can contain

  - **variable definitions**

    One variable definition per line of the form `NAME=VALUE`.

  - **blank lines**

    blank lines will be ignored.

  - **comments**

    lines that start with # will be ignored.

  The file is assumed to be encoded in the UTF-8 format. Should you require a different encoding you can prefix the filename with `CHARSET:`, where CHARSET is replaced with the name of the encoding.

  Instead of a single variable file you can also specify a list of files separated by semicolons. The optional charset prefix must be specified for each file separately.
  Examples would be:

  ```
  -f varfile.txt
  --var-file=ISO-8859-3:varfile.txt
  --var-file=one.txt;two.txt
  --var-file=ISO-8859-3:one.txt;ISO-8859-1:two.txt
  ```

- **-M or --list-media-types**

  Prints out a lists of supported media types for the `--media-types` option and quits.

E.2 Command Line Tool For Pre-Created JRE Bundles

To automate the creation of pre-created JRE bundles [p. 89], you can use the command line utility `createbundle[.exe]` in the `bin` directory of your install4j installation. The bundle creation tool is invoked as follows:

```
createbundle [OPTIONS] [JRE home directory]
```

The available options are:

```
-h,  --help               Displays this help.
-o,  --output             Output directory, default is the current directory.
-v,  --version=<VERSION>  JRE version to be used in the bundle file name.
                          The default is the version as reported by the JRE.
-i,  --id                 Sets custom id for bundle file name.
                          The default is the empty string.
-u,  --unpacked           Create bundle with unpacked JAR files as required
                          for the macOS single bundle archive.
-r,  --jdk-release        Release of JDK that provides the JDK tools. Only
         =<RELEASE>       required if the JRE does not contain the jlink tool
                          and if the JRE version is 9 or higher. This is not a
                          version number, but a release string as shown on the
                          "JRE Bundles" step in the install4j IDE.
-p,  --jdk-provider-id    JDK provider ID for the JDK that is specified with
         =<ID>            --jdk-release. By default "AdoptOpenJDK" is used.
-m,  --add-modules        Add a comma-separated list of modules to the JRE
                          bundle. Can be passed more than once.
-s,  --add-module-set     Add a set of modules to the JRE bundle, either a
       =min|jre|all|none  minimum set, a typical JRE, all modules, or none.
                          The default is "jre".
-j,  --add-jmod=<path>    Add a JMOD file to the JRE bundle. Can be passed
                          more than once.
-d,  --add-jmod-dir       Add a directory with JMOD files to the JRE bundle.
         =<path>          Can be passed more than once.
```

There are Ant [p. 239] and Gradle [p. 225] tasks as well as a Maven Mojo [p. 230] tasks that you can use to call this command line application from your build system.

E.3 Using Install4j With Gradle

You can execute the install4j compiler from gradle (1) with the install4j Gradle plugin. To make the Gradle plugin available to your build script, you have to apply the install4j Gradle plugin:

```
plugins {
    id "com.install4j.gradle" version "X.Y.Z"
}
```

If you do not want to use the Gradle plugin repository for this purpose, the Gradle plugin is distributed in the file `bin/gradle.jar`.

The plugin has two parts: The global configuration with the top-level `install4j {...}` configuration block and tasks of type `com.install4j.gradle.Install4jTask`.

The global configuration block must specify the install4j installation directory:

```
install4j {
    installDir = file("/path/to/install4j_home")
}
```

On macOS, the installation directory is the path of the application bundle, for example `/Applications/install4j.app`. The actual command line compiler is located under `/Applications/install4j.app/Contents/Resources/app/bin/install4j` in that case.

In addition, the global configuration block can set defaults for the `install4j` tasks.

Task parameters

The `install4j` task supports the following parameters, many of which are explained in greater detail for the command line compiler [p. 220].

Attribute	Description	Required	Global
projectFile	The install4j project file that should be build.	Yes	No
variableFiles	Corresponds to the `--var-file` command line option. Specify the list of variable files with variable definitions.	No	No
variables	A map of variable definitions. These definitions override compiler variables [p. 63] in the project and correspond to the `-D` command line option. Definitions with `variable` elements take precedence before definitions in the variable file referenced by the `variableFiles` parameter. The names of the variables must have been defined on the "General	No	No

---

(1) https://gradle.org

Attribute	Description	Required	Global
	Settings->Compiler Variables" step. The values can be of any type, `toString()` will be called on each value to convert the value to a `java.lang.String`. For example: `[variableOne: "One", variableTwo: 2]`.		
release	Corresponds to the `--release` command line option. Enter a version number like "`3.1.2`". Version number components can be alphanumeric and should be separated by dots, dashes or underscores.	No	No
destination	Corresponds to the `--destination` command line option. Enter a directory where the generated media files should be placed.	No	No
buildIds	Corresponds to the `--build-ids` command line option. Enter a list of media file ids. The IDs for media files can be shown in the install4j IDE by choosing *Project->Show IDs* from the main menu. For example: `[12, 24, 36]`.	No	No
verbose	Corresponds to the `--verbose` command line option. Either `true` or `false`.	No, verbose and quiet cannot **both** be `true`	Yes
quiet	Corresponds to the `--quiet` command line option. Either `true` or `false`.		Yes
license	Corresponds to the `--license` command line option. If the license has not been configured yet, you can set the license key with this attribute.	Yes	
test	Corresponds to the `--test` command line option. Either `true` or `false`.	No, test and incremental cannot **both** be `true`	Yes
incremental	Corresponds to the `--incremental` command line option. Either `true` or `false`.		Yes
debug	Corresponds to the `--debug` command line option. Either `true` or `false`.	No	Yes

Attribute	Description	Required	Global
preserve	Corresponds to the `--preserve` command line option. Either `true` or `false`.	No	Yes
faster	Corresponds to the `--faster` command line option. Either `true` or `false`.	No	Yes
disableSigning	Corresponds to the `--disable-signing` command line option. Either `true` or `false`.	No	Yes
disableBundling	Corresponds to the `--disable-bundling` command line option. Either `true` or `false`.	No	Yes
winKeystorePassword	Corresponds to the `--win-keystore-password` command line option.	No	Yes
macKeystorePassword	Corresponds to the `--mac-keystore-password` command line option.	No	Yes
appleIdPassword	Corresponds to the `--apple-id-password` command line option.	No	Yes
disableNotarization	Corresponds to the `--disable-notarization` command line option.	No	Yes
buildSelected	Corresponds to the `--build-selected` command line option. Either `true` or `false`.	No	Yes
mediaTypes	Corresponds to the `--media-types` command line option. Enter a list of media types. To see the list of supported media types, execute `install4jc --list-media-types`.	No	Yes
vmParameters	A list of VM parameters for the install4j command line compiler process. For example: `["-DproxySet=true", "-DproxyHost=myproxy", "-DproxyPort=1234", "-DproxyAuth=true", "-DproxyAuthUser=buildServer", "-DproxyAuthPassword= iq4zexwb8et"]` sets an HTTP proxy that is required for code signing.	No	Yes

The "Global" column shows if a parameter can also be specified in the global `install4j {... }` configuration block. Definitions in the task override global definitions.

Examples

Simple example:

```
install4j {
    installDir = file("/opt/install4j")
}
task media(type: com.install4j.gradle.Install4jTask) {
    projectFile = file("myProject.install4j")
}
```

More complex example:

```
if (!hasProperty("install4jHomeDir")) {
    File propertiesFile =
file("${System.getProperty("user.home")}/.gradle/gradle.properties")
    throw new RuntimeException("Specify install4jHomeDir in $propertiesFile")
}

boolean dev = hasProperty("dev")

install4j {
    installDir = file(install4jHomeDir)
    faster = dev
    disableSigning = dev
    winKeystorePassword = "supersecretWin"
    macKeystorePassword = "supersecretMac"

    if (dev) {
        mediaTypes = ["windows"]
    }
}

task media(type: com.install4j.gradle.Install4jTask) {
    dependsOn "dist" // example task that prepares the distribution for install4j

    projectFile = "myProject.install4j"
    variables = [majorVersion: version.substring(0, 1), build: 1234]
    variableFiles = ["var1.txt", "var2.txt"]
}
```

The "hello" sample project includes a Gradle build script that shows how to setup the install4j task. To install the sample projects, invoke *Project->Open Sample Project* from the install4j IDE. When you do this for the first time, the sample projects are copied to the "Documents" folder in your home directory.

In the `samples/hello` directory, execute

```
gradle media
```

to start the build. If you have not defined `install4jHomeDir` in `gradle.properties` next to `build.gradle`, the build will fail with a corresponding error message.

Creating JRE bundles

To create a JRE bundle from your Gradle build script, use the `com.install4j.gradle.CreateBundleTask` and and set its `javaHome` property to the JRE that you want to create a bundle for.

The `CreateBundleTask` invokes the createbundle command line executable [p. 224] in the install4j installation and has the following properties:

Attribute	Description	Required
javaHome	The home directory of the JRE that should be bundled	Yes
outputDirectory	Corresponds to the `--output` command line option.	No
version	Corresponds to the `--version` command line option.	No
id	Corresponds to the `--id` command line option.	No
unpacked	Corresponds to the `--unpacked` command line option.	No
jdkRelease	Corresponds to the `--jdk-release` command line option.	No
jdkProviderId	Corresponds to the `--jdk-provider-id` command line option.	No
addModules	Corresponds to the `--add-modules` command line option.	No
addModuleSet	Corresponds to the `--add-module-set` command line option.	No
jmodFiles	Corresponds to the `--add-jmod` command line option.	No
jmodDirs	Corresponds to the `--add-jmod-dir` command line option.	No
vmParameters	Like the vmParameters property of the `Install4jTask`	No

Example:

```
task createBundle(type: com.install4j.gradle.CreateBundleTask) {
    javaHome = "/usr/lib/jvm/jre-11/jre"
    outputDirectory = "/home/build/projects/myProject/jreBundles"
    version = "11"
    id="j3d"
    jmodDirs = ["jmods"]
    jmodFiles = ["one.jmod", "two.jmod"]
}
```

E.4 Using Install4j With Maven

You can execute the install4j compiler from maven [1] with the install4j Maven plugin.

The install4j maven plugin is available from the following repository:

```
<pluginRepositories>
  <pluginRepository>
    <id>ej-technologies</id>
    <url>https://maven.ej-technologies.com/repository</url>
  </pluginRepository>
</pluginRepositories>
```

Compile Mojo parameters

The `compile` Mojo supports the following parameters, many of which are explained in greater detail for the command line compiler [p. 220].

Parameter	Description	Required
installDir	The location of the install4j installation. User property of type `java.io.File`: install4j.home	Yes
projectFile	The install4j project file that should be build. User property of type `java.io.File`: install4j.projectFile	Yes
appleIdPassword	Set the app-specific password for notarizing macOS media files. This only has an effect when run on a macOS machine. Corresponds to the `--apple-id-password` command line option. User property of type `java.lang.String`: install4j.appleIdPassword	No
attach	Attach generated installers. Uses the media file ID as the classifier. User property of type `boolean`: install4j.attach	No
buildIds	Only build the media files with the specified IDs, separated by commas. Corresponds to the `--build-ids` command line option. User property of type `java.lang.String`: install4j.buildIds	No

---

(1) https://maven.apache.org/

Parameter	Description	Required
buildSelected	Only build the media files which have been selected in the install4j IDE. Corresponds to the `--build-selected` command line option. User property of type `boolean`: install4j.buildSelected	No
debug	Create additional debug installers for each media file. Corresponds to the `--debug` command line option. User property of type `boolean`: install4j.debug	No
destination	The output directory for the generated media files. By default this is set to `${project.build.directory}/media`, so this flag is always passed to the install4j compiler. Corresponds to the `--destination` command line option. User property of type `java.io.File`: install4j.destination	No
disableBundling	Disable JRE bundling. Corresponds to the `--disable-bundling` command line option. User property of type `boolean`: install4j.disableBundling	No
disableNotarization	Disable Notarization on macOS. Corresponds to the `--disable-notarization` command line option. User property of type `boolean`: install4j.disableNotarization	No
disableSigning	Disable code signing. Corresponds to the `--disable-signing` command line option. User property of type `boolean`: install4j.disableSigning	No
failOnWarning	If a warning is printed and this option is specified, the build will fail at the end. Corresponds to the `--fail-on-warning` command line option. User property of type `boolean`: install4j.failOnWarning	No

Parameter	Description	Required
faster	Disable LZMA and Pack200 compression. Corresponds to the `--faster` command line option. User property of type `boolean`: install4j.faster	No
incremental	Enables incremental test execution. The parameters "test" and "incremental" cannot both be true. Corresponds to the `--incremental` command line option. User property of type `boolean`: install4j.incremental	No
jvmArguments	Pass JVM arguments to the install4j command line compiler.	No
license	install4j license key. If the license has not been configured yet, you can set the license key with this attribute. Corresponds to the `--license` command line option. User property of type `java.lang.String`: install4j.license	No
macKeystorePassword	Set the macOS keystore password for the private key that is configured for code signing. Corresponds to the `--mac-keystore-password` command line option. User property of type `java.lang.String`: install4j.macKeystorePassword	No
mediaTypes	Only build media files of the specified types, separated by commas. Corresponds to the `--build-ids` command line option. User property of type `java.lang.String`: install4j.mediaTypes	No
preserve	Preserve temporary staging directory. Corresponds to the `--preserve` command line option. User property of type `boolean`: install4j.preserve	No
quiet	Enables quiet mode. The parameters "verbose" and "quiet" cannot both be true. Corresponds to the `--quiet` command line option.	No

Parameter	Description	Required
	User property of type `boolean`: install4j.quiet	
release	Override the application version. By default this is set to `${project.version}`, so this flag is always passed to the install4j compiler unless you set it to the special string `#project`. Corresponds to the `--release` command line option. User property of type `java.lang.String`: install4j.release	No
skip	Skip execution. User property of type `boolean`: install4j.skip	No
test	Enables test mode. In test mode, no media files will be generated in the media file directory. The parameters "test" and "incremental" cannot both be true. Corresponds to the `--test` command line option. User property of type `boolean`: install4j.test	No
variableFiles	Load variable definitions from files. Corresponds to the `--var-file` command line option.	No
variables	Override compiler variables with different values. Corresponds to the `-D` command line option.	No
verbose	Enables verbose mode. The parameters "verbose" and "quiet" cannot both be true. Corresponds to the `--verbose` command line option. User property of type `boolean`: install4j.verbose	No
winKeystorePassword	Set the Windows keystore password for the private key that is configured for code signing. Corresponds to the `--win-keystore-password` command line option. User property of type `java.lang.String`: install4j.winKeystorePassword	No

Example

A minimal example is:

```
<build>
  <plugins>
    <plugin>
      <groupId>com.install4j</groupId>
      <artifactId>install4j-maven</artifactId>
      <version>9.0.7</version>
      <executions>
        <execution>
          <id>compile-installers</id>
          <phase>package</phase>
          <goals>
            <goal>compile</goal>
          </goals>
          <configuration>
            <installDir>/path/to/install4j</installDir>

<projectFile>${project.basedir}/src/main/installer/myProject.install4j</projectFile>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Compilation can be skipped by setting the `install4j.skip` property on the command line:

```
mvn -Dinstall4j.skip
```

Using profiles for configuring parameters

Instead of using the `installDir` parameter, it is recommended to configure the installation location in `settings.xml` with the `install4j.home` user property:

```
<profiles>
  <profile>
    <id>development</id>
    <properties>
      <install4j.home>/path/to/install4j</install4j.home>
    </properties>
  </profile>
</profiles>

<activeProfiles>
  <activeProfile>development</activeProfile>
</activeProfiles>
```

Further parameters that are recommend to be configured in `settings.xml` are the license key and the passwords for code signing. The license key configuration is only required if it was not configured manually in advance for the user that is running the build.

```
<profiles>
  <profile>
    <id>development</id>
    <properties>
      <install4j.licenseKey>CHANGEME</install4j.licenseKey>
      <install4j.winKeystorePassword>SECRET</install4j.winKeystorePassword>
      <install4j.macKeystorePassword>SECRET</install4j.macKeystorePassword>
    </properties>
  </profile>
</profiles>

<activeProfiles>
  <activeProfile>development</activeProfile>
</activeProfiles>
```

Passing the build class path to the project

A common use case is the need to add all dependency JAR files from the build class path to the distribution tree. To do that, you first have to execute the "build-classpath" goal of the "maven-dependency-plugin" to set a property with the class path:

```
<plugin>
  <artifactId>maven-dependency-plugin</artifactId>
  <version>3.1.2</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>
      <goals>
        <goal>build-classpath</goal>
      </goals>
      <configuration>
        <outputProperty>my.classpath</outputProperty>
      </configuration>
    </execution>
  </executions>
</plugin>
```

In the configuration of the install4j plugin, you then pass this property as a compiler variable:

```
<configuration>
  ...
  <variables>
    <externalClassPath>${my.classpath}</externalClassPath>
  </variables>
</configuration>
```

On the "Files->Define distribution tree" step in the install4j step, you can add entries of type "Compiler variable" [p. 14]. This type of entry will split the variable value with a configurable path separator and add all contained files. Continuing the above example, you have to add a compiler variable entry with the compiler variable name "externalClassPath" and the default path list separator `${compiler:sys.pathlistSeparator}` to add all the dependency JAR files to the selected location in the distribution tree.

Attaching media files

Media files compiled by install4j can be attached to the Maven project when the attach parameter is set to `true`.

Attached files will be installed into the local repository and will also be deployed. The classifier for each deployed media files is the media file ID.

Creating JRE bundles

To create a JRE bundle from your Maven build, use the `createbundle` Mojo and set its `javaHome` property to the JRE that you want to create a bundle for.

The `createbundle` Mojo supports the following parameters, many of which are explained in greater detail for the command line compiler [p. 220].

Parameter	Description	Required
installDir	The location of the install4j installation. User property of type `java.io.File`: install4j.home	Yes
javaHome	The home directory of the JRE that should be bundled. User property of type `java.io.File`: install4j.bundleJavaHome	Yes
addModuleSet	Add a set of modules to the JRE bundle, one of "MIN", "JRE", "ALL", "NONE". Corresponds to the `--add-module-set` command line option. User property of type `com.install4j.buildtools.ModuleSet`: install4j.addModuleSet	No
addModules	Comma-separated list of modules to be added to the JRE bundle. Corresponds to the `--add-modules` command line option. User property of type `java.lang.String`: install4j.addModules	No
id	Optional custom ID for the bundle. Corresponds to the `--id` command line option. User property of type `java.lang.String`: install4j.bundleId	No
jdkProviderId	JDK provider ID for the JDK that is specified with `jdkRelease`. Corresponds to the `--jdk-provider-id` command line option. User property of type `java.lang.String`: install4j.jdkProviderId	No
jdkRelease	Release of a JDK that provides the JDK tools. Required only if the bundled JRE does not contain the jlink tool. Corresponds to the `--jdk-release` command line option. User property of type `java.lang.String`: install4j.jdkRelease	No

Parameter	Description	Required
jmodDirs	Directories with JMOD files to be added to the JRE bundle. Corresponds to the `--add-jmod-dir` command line option.	No
jmodFiles	JMOD files to be added to the JRE bundle. Corresponds to the `--add-jmod` command line option.	No
jvmArguments	Pass JVM arguments to the install4j command line compiler.	No
outputDirectory	Output directory for the bundle. Corresponds to the `--output` command line option. User property of type `java.io.File`: install4j.bundleOutputDir	No
skip	Skip execution. User property of type `boolean`: install4j.skip	No
unpacked	Create bundle with unpacked JAR files, required for macOS single bundle archives. Corresponds to the `--unpacked` command line option. User property of type `boolean`: install4j.bundleUnpacked	No
version	JRE version to be used, if different from the detected version. Corresponds to the `--version` command line option. User property of type `java.lang.String`: install4j.bundleVersion	No

An example that shows the usage of this Mojo is:

```
<build>
  <plugins>
    <plugin>
      <groupId>com.install4j</groupId>
      <artifactId>install4j-maven</artifactId>
      <version>9.0.7</version>
      <executions>
        <execution>
          <id>create-jre-bundle</id>
          <phase>package</phase>
          <goals>
            <goal>createbundle</goal>
          </goals>
          <configuration>
            <installDir>/path/to/install4j</installDir>
            <javaHome>/usr/lib/jvm/jre-11/jre</javaHome>
          <outputDirectory>/home/build/projects/myProject/jreBundles</outputDirectory>

            <jmodFiles>
              <param>one.jmod</<param>
              <param>two.jmod</<param>
            </jmodFiles>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

E.5 Using Install4j With Ant

To integrate install4j with your Ant script (1) use the `Install4JTask` that is provided in `$INSTALL4J_HOME/bin/ant.jar` and set theCreateBundleTask `projectFile` parameter to the install4j project file that you want to build.

To make the `install4j` task available to Ant, you must first insert a `taskdef` element that tells Ant where to find the task definition. Here is an example of using the task in an Ant build file:

```
<taskdef name="install4j"
         classname="com.install4j.Install4JTask"
         classpath="C:\Program Files\install4j\bin\ant.jar"/>

<target name="media">
  <install4j projectFile="myapp.install4j"/>
</target>
```

On macOS, the `ant.jar` file is inside the application bundle, for the default application directory the full path is `/Applications/install4j.app/Contents/Resources/app/bin/ant.jar`

The `taskdef` definition must occur only once per Ant build file and can appear anywhere on the top level below the `project` element.

Note that it is not possible to copy the `ant.jar` archive to the `lib` folder of your ant distribution. You have to reference a full installation of install4j in the task definition.

Task parameters

The `install4j` task supports the following parameters:

Attribute	Description	Required
projectFile	The install4j project file that should be build.	Yes
verbose	Corresponds to the `--verbose` command line option. Either `true` or `false`.	No, verbose and quiet cannot **both** be `true`
quiet	Corresponds to the `--quiet` command line option. Either `true` or `false`.	
failOnWarning	Corresponds to the `--fail-on-warning` command line option. Either `true` or `false`.	
license	Corresponds to the `--license` command line option. If the license has not been configured yet, you can set the license key with this attribute.	Yes
test	Corresponds to the `--test` command line option. Either `true` or `false`.	No, test and incremental cannot **both** be `true`
incremental	Corresponds to the `--incremental` command line option. Either `true` or `false`.	
debug	Corresponds to the `--debug` command line option. Either `true` or `false`.	No

---

(1) https://ant.apache.org

Attribute	Description	Required
preserve	Corresponds to the `--preserve` command line option. Either `true` or `false`.	No
faster	Corresponds to the `--faster` command line option. Either `true` or `false`.	No
disableSigning	Corresponds to the `--disable-signing` command line option. Either `true` or `false`.	No
winKeystorePassword	Corresponds to the `--win-keystore-password` command line option.	No
macKeystorePassword	Corresponds to the `--mac-keystore-password` command line option.	No
appleIdPassword	Corresponds to the `--apple-id-password` command line option.	No
disableNotarization	Corresponds to the `--disable-notarization` command line option.	No
release	Corresponds to the `--release` command line option. Enter a version number like "`3.1.2`". Version number components can be alphanumeric and should be separated by dots, dashes or underscores.	No
destination	Corresponds to the `--destination` command line option. Enter a directory where the generated media files should be placed.	No
buildSelected	Corresponds to the `--build-selected` command line option. Either `true` or `false`.	No
buildIds	Corresponds to the `--build-ids` command line option. Enter a list of media file ids. The IDs for media files can be shown in the install4j IDE by choosing *Project->Show IDs* from the main menu.	No
mediaTypes	Corresponds to the `--media-types` command line option. Enter a list of media types. To see the list of supported media types, execute `install4jc --list-media-types`.	No

Contained elements

- The `Install4JTask` can contain `variable` elements. These elements override compiler variables [p. 63] in the project and correspond to the `-D` command line option. Definitions with `variable` elements take precedence before definitions in the variable file referenced by the `variablefile` parameter.

  The `variable` element supports the following parameters:

Attribute	Description	Required
name	The name of the variable. This must be the name of a variable that has been defined on the "General Settings->Compiler Variables" step.	Yes
value	The value for the variable. The value may be empty.	Yes
mediaFileId	The ID of the media file for which the variable should be overridden. The IDs for media files can be shown in the install4j IDE by choosing *Project->Show IDs* from the main menu.	No

Example:

```
<install4j projectFile="myapp.install4j">
  <variable name="MY_VARIABLE" value="15"/>
  <variable name="OTHER_VARIABLE" value="test" mediaFileId="8"/>
</install4j>
```

- The `install4j` task can contain `variablefile` elements. These elements read text files containing compiler variables definitions. They correspond to the `--var-file` command line option

  The `variablefile` element supports the following parameters:

Attribute	Description	Required
file	The path of the variable file.	Yes

- The `install4j` task can contain `vmParameter` elements. These elements set VM parameters for the install4j command line compiler process.

  The `vmParameter` element supports the following parameters:

Attribute	Description	Required
value	The value of the VM parameter.	Yes

Example for setting an HTTP proxy (an internet connection is required for Windows code signing):

```
<install4j projectFile="myapp.install4j" winKeystorePassword="Kajjs7sgLg22">
  <vmParameter value="-DproxySet=true"/>
  <vmParameter value="-DproxyHost=myproxy"/>
  <vmParameter value="-DproxyPort=1234"/>
  <vmParameter value="-DproxyAuth=true"/>
  <vmParameter value="-DproxyAuthUser=buildServer"/>
  <vmParameter value="-DproxyAuthPassword=iq4zexwb8et"/>
</install4j>
```

Complete example

The "hello" sample project includes an Ant build script that shows how to setup the install4j task. To install the sample projects, invoke *Project->Open Sample Project* from the install4j IDE. When you do this for the first time, the sample projects are copied to the "Documents" folder in your home directory.

In the `samples/hello` directory, execute

```
ant media
```

to start the build. If you have not defined `install4jHomeDir` in `build.xml`, the build will fail with a corresponding error message.

Creating JRE bundles

To create a JRE bundle from your Ant build script, use the `CreateBundleTask` that is provided in `$INSTALL4J_HOME/bin/ant.jar` and set the `javaHome` parameter to the JRE that you want to create a bundle for.

The `CreateBundleTask` invokes the createbundle command line executable [p. 224] in the install4j installation. Just like for the `Install4JTask` above, a `taskdef` element is required:

```
<taskdef name="createbundle"
         classname="com.install4j.CreateBundleTask"
         classpath="C:\Program Files\install4j\bin\ant.jar"/>

<target name="media">
  <createbundle javaHome="c:\Program Files\Java\jre"/>
</target>
```

The `CreateBundleTask` task supports the following parameters:

Attribute	Description	Required
javaHome	The home directory of the JRE that should be bundled	Yes
outputDirectory	Corresponds to the `--output` command line option.	No
version	Corresponds to the `--version` command line option.	No
id	Corresponds to the `--id` command line option.	No
unpacked	Corresponds to the `--unpacked` command line option.	No
jdkRelease	Corresponds to the `--jdk-release` command line option.	No
jdkProviderId	Corresponds to the `--jdk-provider-id` command line option.	No

Attribute	Description	Required
addModules	Corresponds to the `--add-modules` command line option.	No
addModuleSet	Corresponds to the `--add-module-set` command line option.	No

The `CreateBundleTask` task can contain `vmParameter` elements like the `Install4JTask` as well as `jmod` elements with the following parameters:

Attribute	Description	Required
file	Corresponds to the `--add-jmod` command line option.	Either file or dir must be set, but not both
dir	Corresponds to the `--add-jmod-dir` command line option.	

Example:

```
<createbundle javaHome="/usr/lib/jvm/jre-11/jre"
              outputDirectory="/home/build/projects/myProject/jreBundles"
              version="11"
              id="j3d">
    <jmod dir="/home/build/projects/myProject/jmods">
    <jmod file="/home/build/projects/myProject/otherJmods/one.jmod">
    <jmod file="/home/build/projects/myProject/otherJmods/two.jmod">
</createbundle>
```