

# Installation Files

## Writing Installation XML Files

### What You Need

#### Your editor

In order to write your XML installation files, you just need a plain text editor. Of course it's always easier to work with color coded text, so you might rather want to work with a text editor having such a feature. Here is a list of free editors that work well :

- Jext : <http://www.jext.org/>
- JEdit : <http://www.jedit.org/>
- classics like Vim and (X)Emacs.

If you are a developer and tend to write your own patches, extension or features to IzPack sources, or, if you wish to debug your compilation, installation and uninstallation, we recommend these IDE:

- IntelliJ IDEA : <http://www.jetbrains.com/idea/>
- Eclipse : <http://www.eclipse.org/>
- Netbeans : <http://www.netbeans.org/>

For the first one, JetBrains has granted us an Open Source License. All project members can ask the Licence Key to one of the project manager.

The other ones are well know open source projects (Just like us 😊). We provide a tutorial on how to develop/debug IzPack using Eclipse in the chapter *Getting Started > How to develop and debug IpPack using Eclipse*

### Writing XML

Though you might not know much about XML, you have certainly heard about it. If you know XML you can skip this subsection as we will briefly present how to use XML.

XML is a markup language, really close to HTML. If you've ever worked with HTML the transition will be fast. However there are a few little things to know. The markups used in XML have the following form : `<markup>`. Each markup has to be closed somewhere with its ending tag : `</markup>`. Each tag can contain text and other markups. If a markup does not contain anything, it is just reported once : `<markup/>`. A markup can contain attributes like : `<markup attr1="123" attr2="hello !"/>`. Here is a sample of a valid XML structure :

```
<chapter title="Chapter 1">
  <section name="Introduction">
    <paragraph>
      This is the text of the paragraph number
      1. It is available
      for the very low
      price of <price currency="dollar">1 000
      000</price>.
    </paragraph>
  </section>
  <section name="xxx">
    xxx
  </section>
</chapter>
```

You should be aware of the following common mistakes :

**markups\*are** case sensitive : `<markup>` is different from `<Markup>`.  
**you\*must** close the markups in the same order as you create them : `<m1><m2>( . . . )</m2></m1>` is right but `<m1><m2>( . . . )</m1></m2>` is not.

Also, an XML file must start with the following header : `<?xml version="1.0" encoding="iso-8859-1 standalone="yes" ?>`. The only thing you should modify is the encoding (put here the one your text editor saves your files to). The `standalone` attribute is not very important for us.

This (brief !) introduction to XML was just meant to enable you to write your installation specification. For a better introduction there are plenty of books and articles/tutorials dealing with XML on the Internet, in book stores, in magazines and so on.

## Variable Substitution

During the installation process IzPack can substitute variables in various places with real values. Obvious targets for variable substitution are resource files and launch scripts, however you will notice many more places where it is more powerful to use variables rather than hard coded values. Wherever variables can be used it will be explained in the documentation.

There are three types of variables:

- **Built-In variables.** These are implemented in IzPack and are all dynamic in nature. This means that the value of each variable depends on local conditions on the target system.

- Environment variables. These are provided by the operating system the installer is run on.
- Variables that you can define. You also define the value, which is fixed for a given installation file.

You define your own variables in the installation XML file with the `<variable>` tag. How to do this is explained in detail later in this chapter.

**Please note** that when using variables they must always appear with a '\$' sign as the first character, even though they are not defined this way.

## The Built-In Variables

The following variables are built-in :

- `$INSTALL_PATH` : the installation path on the target system, as chosen by the user
- `$INSTALL_DRIVE` : the drive letter part of the installation path on the target system, set on Windows systems, only (for instance C:)
- `$APPLICATIONS_DEFAULT_ROOT` : the default path for applications
- `$JAVA_HOME` : the Java™ virtual machine home path
- `$CLASS_PATH` : the Class Path used mainly for Java Applications
- `$USER_HOME` : the user's home directory path
- `$USER_NAME` : the user name
- `$APP_NAME` : the application name
- `$APP_URL` : the application URL
- `$APP_VER` : the application version
- `$ISO3_LANG` : the ISO3 language code of the selected langpack.
- `$IP_ADDRESS` : the IP Address of the local machine.
- `$HOST_NAME` : the HostName of the local machine.
- `$FILE_SEPARATOR` : the file separator on the installation system
- `$DesktopShortcutCheckboxEnabled` : When set to true, it automatically checks the "Create Desktop Shortcuts" button. To see how to use it, go to `The Variables Element `<variables>` Be careful this variable is case sensitive !
- `$InstallerFrame.logfilePath` : The path to the install log. This file contains the paths of all installed files. If set to "default" then the "`$INSTALL_PATH/Uninstaller/install.log`" path will be used. To see how to use it, go to `The Variables Element `<variables>`. If this variable is not set, no `install.log` will be created.

## Environment Variables

Environment variables can be accessed via the syntax `${ENVvariable}`. The curly braces are mandatory. Note that variable names are case-sensitive and usually in UPPER CASE.

Example: To get the value of the OS environment variable "CATALINA\_HOME", use `${ENVCATALINA_HOME}`.

## Dynamic Variables

Dynamic variables can be defined in the installation XML with the `<variable>` tag inside the `<dynamicvariables>` element. The value of dynamic variables will be evaluated every time a panel is switched, i.e. between the panels. Dynamic variables can have a condition which will be evaluated first. If it's true, the value would be assigned, otherwise nothing happens to the variable.

As an addition to normal variables, the value of a variable can either be defined by using the value attribute or by using a child element called value.

Example1: To change a certain directory based on user input, use the following `<variable name="test" value="/test/${USER_INPUT}" condition="hasuserinput" />` The condition has userinput has to be specified in the condition section of installation XML.

Example2: To comment out something in a xml file if a certain pack with id mycoolfeature is not activated, you could use "two" dynamic variables to create a xml comment or not.

```
<variable name="XML_Comment_Start"
condition="!izpack.selected.mycoolfeature">
  <value><![CDATA[<!-- ]]></value>
</variable>
<variable name="XML_Comment_End"
condition="!izpack.selected.mycoolfeature">
  <value><![CDATA[ --> ]]></value>
</variable>
<variable name="XML_Comment_Start" value=""
condition="izpack.selected.mycoolfeature" />
<variable name="XML_Comment_End" value=""
condition="izpack.selected.mycoolfeature" />
```

The condition `izpack.selected.mycoolfeature` is generated automatically when a pack with id `mycoolfeature` was specified. You would now use `${XML_Comment_Start}` and `${XML_Comment_End}` in a file which should be parsed.

## Parse Types

Parse types apply only when replacing variables in text files. At places where it might be necessary to specify a parse type, the documentation will mention this. Depending on the parse type, IzPack will handle special cases ~~such as escaping control characters~~ correctly. The following parse types are available:

- `plain` - use this type for plain text files, where no special substitution rules apply. All variables will be replaced with their respective values as is.
- `javaprop` - use this type if the substitution happens in a Java properties file. Individual variables might be modified to function properly within the context of Java property files.
- `java` - use this type for Java files.
- `xml` - use this type if the substitution happens in a XML file. Individual variables might be modified to function properly within the context of XML files.
- `shell` - use this type if the substitution happens in a shell script. Because shell scripts use `$variable` themselves, an alternative variable marker is used: `%variable` or `%{variable}`.
- `at` - use this type if the substitution must occur on files where parameters are marked with leading AT characters. The example: `@variable`.
- `ant` - use this type if the substitution must occur on files where parameters are surrounded with AT

characters (similar to ANT filters, hence the type name). The example: `{variable}`.

Unless using braces to surround variable's name (`{variable}` or `%{variable}`), the variable name can contain following characters: letters, digits, dots, dashes **–**, underbars (`_`). Example: `$this.is-my_variable`

If you want to have two variables separated by character that is allowed to appear in variable name, for example: `$major-version.$minor-version`, then you must use braces, and the above example should look like: `${major-version}.${minor-version}`.

## The IzPack Elements

When writing your installer XML files, it's a good idea to have a look at the *IZPACK* installation DTD.

### The Root Element `<installation>`

The root element of an installation is `<installation>`. It takes one required attribute: `version`. The attribute defines the version of the XML file layout and is used by the compiler to identify if it is compatible with the XML file. This should be set to **1.0** for the moment.

### The Information Element `<info>`

This element is used to specify some general information for the installer. It contains the following elements :

- `<appname>` : the application name
- `<appversion>` : the application version
- `<appsubpath>` : the subpath for the default of the installation path. A variable substitution and a maskable slash-backslash conversion will be done. If this tag is not defined, the application name will be used instead.
- `<url>` : the application official website url
- `<authors>` : specifies the author(s) of the application. It must contain at least one `<author>` element whose attributes are :
  - `name` : the author's name
  - `email` : the author's email
- `<uninstaller>` : specifies whether to create an uninstaller after installation, and which name to use for it. This tag has the `write` attribute, with default value `yes`. If this tag is not specified, the uninstaller will still be written. The `name` attribute can be used to change the default name of the generated uninstaller, *i.e.* `uninstaller.jar`. The `condition` attribute can be used to specify a condition which has to be fulfilled for creating the uninstaller. The `path` attribute can be used to define the destination path where the uninstaller is written to, *i.e.* `${INSTALL_PATH}/Uninstaller`.
- `<javaversion>` : specifies the minimum version of Java required to install your program. Values can be `1.2`, `1.2.2`, `1.4`, etc. The test is a lexical comparison against the `java.version` System property on the install machine.
- `<requiresjdk>`: (yes or no) specifies whether a JDK is required for the software to be installed and executed. If not, then the user will be informed and given the option to still proceed with the installation process or not.
- `<webdir>` : Causes a *web installer* to be created, and specifies the URL packages are retrieved from at install time. The content of the tag must be a properly formed URL.
- `<summarylogfilepath>` : specifies the path for the logfile of the ``SummaryLoggerInstallerListener``.
- `<writeinstallationinformation>` : (yes or no) specifies if the file `.installinformation` should be written which includes the information about installed packs. The default if not specified is `yes`.
- `<pack200/>`: adding this element will cause every JAR file that you will add to your packs to be compressed using Pack200 (see <http://java.sun.com/j2se/1.5.0/docs/guide/deployment/deployment-guide/pack200.html>).

As a special exception, signed JARs are not compressed using Pack200, as it would invalidate the signatures. This makes the compilation process a little bit longer, but it usually results in drastically smaller installer files. The decompression is relatively fast. Please note that Pack200 compression is destructive, i.e., after decompression a JAR won't be identical to its original version (yet the code in the class files remains semantically equivalent).

- `<run-privileged/>`: adding this element will make the installer attempt to launch itself with administrator permissions. It also supports a `condition` attribute to reference a condition id so that the elevation is not always attempted (e.g., you may want to activate it only for Windows Vista). This is not supported on all platforms, in which case a message will be provided to the user before continuing the installation. You can disable this feature for the uninstaller by specifying `uninstaller="yes"` as an attribute. Only use this feature if you really need to be an administrator as part of your installation process.
- `<rebootaction>`: defines what to do if there were pending file operations left which require a reboot; otherwise any of the options below will be ignored. Possible values are:
  - `ignore` (default): Doesn't reboot at all even if there are pending operations. Pending operations can be recognized only on the installer command line output (for all options).
  - `notice`: Doesn't reboot, but notices the user interactively at the end of an installation, which must be confirmed. Notification works only for interactive installation types (no auto-installation).
  - `ask`: Reboots only if the user confirms it interactively at the end of an installation.
  - `always`: Reboots always without any confirmation at the end of an installation.Notes:
  1. The usage of `<rebootaction>` is bound to using the attribute `blockable_` with the values `auto` or `force` at least in one of the elements `<file>`, `<singlefile>` or `<fileset>`, for being able to recognize blocked files before IzPack tries to overwrite them (which would result in a failing installation of the file). See the documentation of these elements later.
  2. `<rebootaction>` works and makes sense only on Windows, where target files (as device drivers, EXE or DLL files) might be blocked during an installation. On other platforms than Windows the `<rebootaction>` element will be ignored.
  3. `<rebootaction>` supports the `condition` attribute to limit reboot processing on particular conditions.
  4. Without setting at least one attribute `blockable = auto` or `blockable = force` `rebootaction` will not have any effect.  
See also the description of the `blockable_` attribute.  
For getting more information see the chapter *Advanced features*.

Here is an example of a typical `<info>` section :

```
<info>
  <appname>Super extractor</appname>
  <appversion>2.1 beta 6</appversion>

  <appsubpath>myCompany/SExtractor</appsubpath
  >
  <url>http://www.superextractor.com/</url>
  <authors>
    <author name="John John Doo"
email="jjd@jjd-mail.com"/>
    <author name="El Goyo"
email="goyoman@mymail.org"/>
  </authors>
  <javaversion>1.2</javaversion>
</info>
```

Here is one where the privileges elevation is attempted on Windows Vista and Mac OS X :

```

<info>
  <appname>IzPack</appname>
  <appversion>4.2.0</appversion>
  <authors>
    <author email="" name="Julien Ponge
(project founder)"/>
    <author email="" name="The fantastic
IzPack developers and contributors"/>
  </authors>
  <url>http://izpack.org/</url>
  <javaversion>1.5</javaversion>
  <requiresjdk>no</requiresjdk>
  <run-privileged
condition="izpack.windowsinstall.vista|izpac
k.macinstall"/>

  <summarylogfilepath>$INSTALL_PATH/installinf
o/Summary.htm</summarylogfilepath>
</info>

```

## The Packaging Element <packaging>

This element allows to specify packaging options. If not specified, the default will be to create an all in one installer. This element will usually be used to create an installer which spans over multiple volumes, e.g. the output will be two CDs. The packaging-element contains the following elements:

- <packager> : specifies options used by the packager. The packager tag has the `class` attribute, which specifies the class to use for packaging. Currently two implementations are available (`com.izforge.izpack.compiler.packager.impl.Packager`, `com.izforge.izpack.compiler.packager.impl.MultiVolumePackager`). The packager-element can contain the <options> element which can have different attributes for the different implementations of packagers. For the `MultiVolumePackager`, it can have the following attributes:
  - `volumesize`: the size of the volumes
  - `firstvolumefreespace`: free space on the first volume used for the installer jar and additional resources like readme-files etc.

- `<unpacker>` : specifies which unpacker class should be used. Currently there are two unpacker implementations (`com.izforge.izpack.compiler.UnPacker`, `com.izforge.izpack.compiler.MultiVolumeUnPacker`).

Here's an example how to specify an installer which will create multiple volumes. In this example the volumes shall be CDs with 650 megabytes. There will be an additional free space of 150 megabytes on the first volume. This will result in the creation of an `installer.jar` and multiple `installer.pak*` files. The `installer.jar` plus `installer.pak` plus the additional resources have to be copied on the first volume, each `installer.pak.<number>` on several volumes.

```
<packaging>  
  <packager  
class="com.izforge.izpack.compiler.packager.  
impl.MultiVolumePackager">  
  <!-- 650 MB volumes, 150 MB space on the  
first volume -->  
    <options volumesize="681574400"  
firstvolumefreespace="157286400"/>  
  </packager>  
  <unpacker  
class="com.izforge.izpack.installer.MultiVol  
umeUnpacker" />  
</packaging>
```

### The Variables Element `<variables>`

This element allows you to define variables for the variables substitution system. Some variables are built-in, such as `$INSTALL_PATH` (which is the installation path chosen by the user). When you define a set of variables, you just have to place as many `<variable>` tags in the file as needed. If you define a variable named `VERSION` you need to type `$VERSION` in the files to parse. The variable substitutor will then replace it with the correct value. One `<variable>` tag take the following attributes :

- `name` : the variable name
- `value` : the variable value

Here's a sample `<variables>` section :

```
<variables>
  <variable name="app-version" value="1.4"/>
  <variable name="released-on"
value="08/03/2002"/>
</variables>
```

Here's a precise sample on how to use desktopshortcutcheckboxenabled and InstallerFrame.logfilePath variables:

```

<variables>
  <variable
name="InstallerFrame.logfilePath"
value="$INSTALL_PATH
  /My-install.log"/>
  <!-- This means that the log name will
be My-install and that
  it will be stored at the root of the
installation. -->
  <!-- Any path is fine. If value is set
to "Default" then
  "$INSTALL_PATH/uninstall/install.log" is
used. -->
  <!-- And if variable isn't defined then
no log is written. -->
  <variable
name="desktopshortcutcheckboxenabled"
value="true"/>
  <!-- This automatically checks the
"Create Desktop Shortcuts"
  button. Default value is "False". -->
</variables>

```

### The dynamic Variables Element <dynamicvariables>

This element allows you to define dynamic variables for the variables substitution system. In contrast to the static <variables>, dynamic variables will be evaluated every time, a panel switch is done.

When you define a set of variables, you just have to place as many <variable> tags in the file as needed. Normally you would use the condition attribute to specify, when a certain value will be set.

One <variable> tag take the following attributes :

- name : the variable name

- `value` : the variable value
- `condition` : a condition for this variable, which has to be true to set the value

Here's a sample `<dynamicvariables>` section :

```
<dynamicvariables>
  <variable name="app-version" value="1.4"
condition="mycondition1" />
  <variable name="app-version" value="1.4b"
condition="!mycondition1" />
  <variable name="released-on"
value="08/03/2002" />
</dynamicvariables>
```

## The Conditions Element `<conditions>`

This element allows you to define conditions which can be used to dynamically change the installer, e.g. the panels shown, the variables set, files parsed, files executed and much more. When you define a condition it will get a type and an id. The id has to be unique. Conditions can be referenced based on this id (e.g. with the `RefCondition`).

There are several built-in types of conditions. At the time of writing this, Izpack has the following built-in types:

- `VariableCondition`: a condition based on the value of a certain variable
- `PackSelectionCondition`: a condition based on a pack selected for installation
- `JavaCondition`: a condition based on a static java field or method.
- `CompareNumericsCondition`: a condition based on the comparison of a certain variable with a given value and operator.

There are also boolean types to combine more than one condition:

- `AndCondition`: both conditions have to be true
- `OrCondition`: only one of both conditions has to be true
- `XOrCondition`: one condition has to be true, the other one has to be false
- `NotCondition`: the condition has to be false

When you define a set of conditions, you just have to write as many `<condition>` tags as you like. A condition can take the following attributes:

**\*\*;** `type`: the type of the condition. For built-in types, this is the lowercase portion of the condition class

- `:` name without condition appended (`variable,packselection,java, ...`). Custom condition types should be referenced by the full qualified class name, e.g. `de.dr.rules.MyCoolCondition`.
- `id`: the id of the condition. This will be used to refer to this conditions in other elements

The condition element can have several child elements depending on the type of this conditions. E.g. the

VariableCondition has a name and value child element to specify, which variable should have a certain value to fulfill this condition.

This is an example which defines four conditions, two VariableConditions, a JavaCondition and a AndCondition which will refer to two of the first conditions.



```
<conditions>
  <condition type="variable"
id="standardinstallation">
    <name>setup.type</name>
    <value>standard</value>
  </condition>
  <condition type="variable"
id="expertinstallation">
    <name>setup.type</name>
    <value>expert</value>
  </condition>
  <condition type="java"
id="installonwindows">
    <java>

<class>com.izforge.izpack.util.OsVersion</cl
ass>
    <field>IS_WINDOWS</field>
  </java>
  <returnvalue
type="boolean">true</returnvalue>
  </condition>
  <condition type="and"
id="standardinstallation.onwindows">
    <condition type="ref"
refid="standardinstallation"/>
    <condition type="ref"
refid="installonwindows" />
  </condition>
</conditions>
```

Note, from IzPack 3.11 on normally, you don't have to define the compound conditions because you can use a simple expression language. The language has the following operators:

- {+}: an operator for the AndCondition
- |: an operator for the OrCondition
- {}: an operator for the XOrCondition
- !: an operator for the NotCondition

Nevertheless if you define really complex conditions it's much easier to define them using the xml structure.

More types of conditions can be defined by inheriting `com.izforge.izpack.Condition` class.

## Built-in conditions

A number of built-in condition IDs are available for you.

Name	Condition
izpack.windowsinstall	The OS is Windows (any variant)
izpack.windowsinstall.xp	The OS is Windows XP
izpack.windowsinstall.2003	The OS is Windows Server 2003
izpack.windowsinstall.vista	The OS is Windows Vista
izpack.windowsinstall.7	The OS is Windows 7
izpack.linuxinstall	The OS is Linux (any variant)
izpack.solarisinstall	The OS is Solaris (any variant)
izpack.solarisinstall.x86	The OS is Solaris (any x86 variant)
izpack.solarisinstall.sparc	The OS is Solaris (any Sparc variant)

## The Installer Requirements Element <installerrequirements>

This element allows to specify requirements for running the installation. This will be done based on conditions defined in the conditions section.

An installer requirement consists of a condition and a message which will be shown if the condition is not fulfilled. If so, the installer will show the message and exit after that.

- `installerrequirement`: specifies a single installer requirement. You can define an unlimited number of them.

Installerrequirements have the following attributes: - `condition`: an id of a condition defined in the conditions section - `message`: a message text or a langpack key defining which message should be shown before exiting the installer in case of a missing requirement.

```
<installerrequirements>
  <installerrequirement
    condition="izpack.windowsinstall"
    message="This installer could only be run on
    Windows operating systems."/>
</installerrequirements>
```

### The GUI Preferences Element `<guiprefs>`

This element allows you to set the behavior of your installer GUI. This information will not have any effect on the command-line installers that will be available in future versions of IzPack. The arguments to specify are :

- `resizable` : takes `yes` or `no` and indicates whether the window size can be changed or not.
- `width` : sets the initial window width
- `height` : sets the initial window height.

Here's a sample :

```
<guiprefs resizable="no" width="800"
height="600" />
```

Starting from IzPack 3.6, the look and feel can be specified in this section on a per-OS basis. For instance you can use the native look and feels on Win32 and OS X but use a third-party one on Unix-like platforms. To do that, you have to add some children to the `guiprefs` tag:

- `laf`: the tag that specifies a look and feel. It has a `name` parameter that defines the look and feel name.
- Each `laf` element needs at least one `os` tag, specified like in the other parts of the specification that support this tag.
- Like you can add `os` elements, you can add any number of `param` elements to customize a look and feel. A `param` element has two attributes: `name` and `value`.

The available look and feels are:

- `Kunststoff`: `kunststoff`
- `Liquid`: `liquid`
- `Metouia`: `metouia`
- `JGoodies Looks`: `looks`
- `Substance`: `substance`

If you don't specify a look and feel for a particular operating system, then the default native one will be used: Windows on Windows, Aqua on Mac OS X and Metal on the Unix-like variants.

The *Liquid Look and Feel* supports the following parameters:

- `decorate.frames: yes` means that it will render the frames in Liquid style
- `decorate.dialogs: yes` means that it will render the dialogs in Liquid style

The *JGoodies Looks* look and feel can be specified by using the `variant` parameters. The values can be one of:

- `windows`: use the Windows look
- `plastic`: use the basic Plastic look
- `plastic3D`: use the Plastic 3D look
- `plasticXP`: use the Plastic XP look (default).

Here is a small sample:

```
<guiprefs height="600" resizable="yes"
width="800">
  <laf name="metouia">
    <os family="unix" />
  </laf>
  <laf name="looks">
    <os family="windows" />
    <param name="variant" value="extwin"
  />
  </laf>
</guiprefs>
```

The *Substance* look and feel *toned-down* themes can be specified using the `variant` parameter, with the value being one of: `business`, `business-blue`, `business-black`, `creme`, `sahara`, `moderate`, `officesilver`. We have reduced the choice to the toned-down themes since they are the only ones to actually look decent (the other families colors are way too saturated). Please consult <https://substance.dev.java.net/docs/skins/toneddown.html> for a gallery of the different toned-down themes.

Starting from IzPack 3.7, some characteristics can be customized with the `<modifier>` tag. There is a separate description in the `Advanced Features` chapter paragraph `Modifying the GUI`.

## The Localization Element `<locale>`

This element is used to specify the language packs (langpacks) that you want to use for your installer. You must set one `<langpack>` markup per language. This markup takes the `iso3` parameter which specifies the iso3 language code.

Here's a sample :

```

<locale>
  <langpack iso3="eng" />
  <langpack iso3="fra" />
  <langpack iso3="spa" />
</locale>

```

The supported ISO3 codes are :h3. h4. h4. h4. h2. ISO3 code Language h4. h4. h4. cat Catalunyan chn Chinese cze Czech dan Danish glg Galician deu German eng English eus Basque fin Finnish fra French hun Hungarian ita Italian jpn Japanese mys Malaysian ned Nederlands nor Norwegian pol Polish por Portuguese (Brazilian) prt Portuguese (European) rom Romanian rus Russian scg Serbian spa Spanish svk Slovakian swe Swedish ukr Ukrainianh3. h4. h4.h3. The Resources Element <resources> h3. Several panels, such as the license panel and the shortcut panel, require additional data to perform their task. This data is supplied in the form of resources. This section describes how to specify them. Take a look at each panel description to see if it might need any resources. Currently, no checks are made to ensure resources needed by any panel have been included. The <resources> element is not required, and no <res> elements are required within. The <resources> element is the only element besides the <packs> element that is taken into consideration in referenced pack-files (see `<packs> element` for more info)

You have to set one <res> markup for each resource. Here are the attributes to specify :

- `src` : the path to the resource file which can be named freely of course (for instance `my-picture.jpg`).
- `id` : the resource id, depending on the needs of a particular panel
- `parse` : takes `yes` or `no` (default is `no`) - used to specify whether the resource must be parsed at the installer compilation time. For instance you could set the application version in a `readme` file used by `InfoPanel`.
- `type` : specifies the parse type. This makes sense only for a text resource - the default is `plain`, other values are `javaprop`, `xml`, `plain`, `java`, `shell`, `at`, `ant` (Java properties file and XML files)
- `encoding` : specifies the resource encoding if the receiver needs to know. This makes sense only for a text resource.

Here's a sample :

```

<resources>
  <res id="InfoPanel.info"
src="doc/readme.txt" parse="yes" />
  <res id="LicencePanel.licence"
src="legal/License.txt" />
</resources>

```

Please note that in general a resource `id` is unique. Thus if you define multiple resources with the same `id` the later

definition (e.g. a resource defined in a referenced pack-file) will overwrite the previous definition. However there is an exception for `packLang.xml_xyz` files (see `Internationalization of the PacksPanel`). If multiple `packLang`-files were defined, all files will be merged into a single temporary file. This allows `refpack` files to provide their own internationalization-information.

You can define bundles of resources by using `<bundle>`. You have to set a unique id for the bundle `id`. One bundle can be marked to be default: `default="true"`

You need to define a variable to set the name of the `SystemProperty`: `resource.bundle.system.property`

Here's a sample:

```
<variables>
```

```
... <variable name="resource.bundle.system.property" value="BUNDLE_NAME" />
```

```
</variables> ... <resources> <
```

```
Unable to render embedded object: File (-- bundle 1 --> <bundle id="bundle1" default="yes"> <res id="Heading.image" src="path1/HeadingImage.png" /> <res id="installer.langsel.img" src="path1/langsellimage.png" /> </bundle> <) not found.
```

```
-- bundle 2 --><bundle id="bundle2"><res id="Heading.image" src="path2/HeadingImage.png" /><res id="installer.langsel.img" src="path2/langsellimage.png" /></bundle><!-- common resources --> <res id="resource.id" src="path.to.resource" /> ...
```

Resources will be then be loaded in the following order:

```
/res/<bundleName>/<resourceName>_<iso3>  
/res/<bundleName>/<resourceName>  
/res/<resourceName>_<iso3>  
/res/<resourceName>
```

The `panels` Element `<panels>` h3. Here you tell the compiler which panels you want to use. They will appear in the installer in the order in which they are listed in your XML installation file. Take a look at the different panels in order to find the ones you need. The `<panel>` markup takes the following attributes:

- `classname`: which is the classname of the panel.
- `id`: an identifier for a panel which can be used e.g. for referencing in `userinput` panel definitions.
- `condition`: an id of a condition which has to be fulfilled to show this panel
- `jar`: jar file where the classes for this panel can be found. This attribute is optional. If it is empty (`jar=""`) the classes for this panel must be merged using the `<jar>` tag.

Here is a sample :

```

<panels>
  <panel classname="HelloPanel" />
  <panel classname="LicencePanel" />
  <panel classname="TargetPanel" />
  <panel classname="InstallPanel" />
  <panel classname="UserInputPanel"
id="myuserinput" condition="pack2selected"
/>
  <panel classname="FinishPanel"
jar="MyFinishPanel.jar" />
</panels>

```

The following sections describe the tags available for a `<panel>` section. `<help>` - optional file for a help h3. The content of the help file is shown in a small window on the panel, when User clicks on the help button. The button is only shown, when a help in the language exists.

The `<help>` takes the following attributes :

- `iso3`: iso3 representation of the language the help is written
- `src`: path to the help file to display

Here's a sample :

```

<panel classname="HelloPanel">
  <help iso3="deu"
src="HelloPanelHelp_deu.html" />
  <help iso3="eng"
src="HelloPanelHelp_eng.html" />
</panel>

```

`<validator>` - optional validation on idata h3. This validation is done, when going on for the next panel. It is also done in case of an automatic installation. The class must implement the interface `com.izforge.izpack.api.installer.DataValidator`.

The `<validator>` takes the following attributes :

- `classname`: The class implementing `com.izforge.izpack.api.installer.DataValidator`

Here's a sample :

```
<panel classname="UserInputPanel"
id="jdbc.connection.parameters">
  <validator
classname="JdbcConnectionValidator" />
</panel>
```

`<actions>` - optional actions for the panel h3. Here you can define multiple actions that are done during the lifetime of the panel. The class must implement the interface `com.izforge.izpack.installer.PanelAction`. The actions are also called during an automated installation.

The `<actions>` tag has no attributes but has `<action>` markups with the following attributes :

- `stage`: The stage when the action should be triggered. Possible values are `preconstruct`, `preactivate`, `prevalidate` or `postvalidate`.
- `classname`: The class implementing `com.izforge.izpack.installer.PanelAction`

Here's a sample :

```
<panel classname="UserInputPanel"
id="jdbc.connection.parameters">
  <actions>
    <action stage="preconstruct"
classname="ConnectionPreConstructAction" />
    <action stage="preactivate"
classname="ConnectionPreActivateAction" />
    <action stage="prevalidate"
classname="ConnectionPreValidateAction" />
    <action stage="postvalidate"
classname="ConnectionPostValidateAction" />
  </actions>
</panel>
```

The Packs Element `<packs>` h3. This is a crucial section as it is used to specify the files that need to be installed. The `<packs>` section consists of several `<pack>`, `<refpack>` and `<refpackset>` tags.

The `<pack>` takes the following attributes :

- `name`: the pack name
- `required`: takes `yes` or `no` and specifies whether the pack is optional or not.
- `os`: optional attribute that lets you make the pack targeted to a specific *operating system*, for instance `unix`, `mac` and so on.
- `preselected`: optional attribute that lets you choose whether the pack is by default selected for installation or not. Possible values are `yes` and `no`. A pack which is not preselected needs to be explicitly selected by the user during installation to get installed.
- `loose`: can be used so that the files are not located in the installer Jar. The possible values are `true` or `false`, the default being `false`. The author of this feature needed to put his application on a CD so that the users could run it directly from this media. However, he also wanted to offer them the possibility to install the software locally. Enabling this feature will make IzPack take the files on disk instead of from the installer. *Please make sure that your relative files paths are correct !*
- `id`: this attribute is used to give a unique id to the pack to be used for internationalization via `packsLang.xml` file.
- `packImgId`: this attribute is used to reference a unique resource that represents the pack's image for the `ImgPacksPanel`. The resource should be defined in the `<resources>` element of the installation XML using the same value for the `id` attribute of the `<res>` element.
- `condition`: an id of a condition which has to be fulfilled to install this package.
- `hidden`: takes `true` or `false` and specifies whether the pack shall be shown in the packs panel. The bytes of such a hidden pack will be used to calculate the required space, but the pack itself won't be shown. A hidden pack can be selected conditionally. So you have to specify a condition to enable it for installation. The default for this attribute is `false`

The `<refpack>` takes only one attribute `file`, which contains the relative path (from the installation compiler) to an externally defined packs-definition. This external packs definition is a regular IzPack installation XML. However the only elements that are used from that XML file are the `<packs>` and the `<resources>` elements. This enables a model in which a single developer is responsible for maintaining the packs and resources (e.g. separate `packsLang.xml_xyz` files providing internationalization; see 'Internationalization of the PacksPanel') related to the development-package assigned to him. The main install XML references these xml-files to avoid synchronization efforts between the central installation XML and the developer-maintained installer XMLs.

The `<refpackset>` tag can be used in situations where there is no predefined set of `<refpack>` files, but a given directory should be scanned for `<refpack>` files to be included instead. This element takes the following parameters:

- `dir`: the base directory for the `refpackset` (relative path)
- `includes`: a pattern of files in `<refpack>` format that must be included

An example:

```
<refpackset dir="" includes="**/refpack.xml" />
```

Internationalization of the PacksPanel h3. In order to provide internationalization for the `PacksPanel`, so that your

users can be presented with a different name and description for each language you support, you have to create a file named `packsLang.xml_xyz` where `xyz` is the ISO3 code of the language in lowercase. Please be aware that case is significant. This file has to be inserted in the resources section of `install.xml` with the `id` and `src` attributes set at the name of the file. The format of these files is identical with the distribution langpack files located at `$IZPACK_HOME/bin/langpacks/installer`. For the name of the panel you just use the pack id as the `txt id`. For the description you use the pack id suffixed with `.description`.

An example:

```
<resources>
  <res id="packsLang.xml_eng"
  src="i18n/myPacksLang.xml_eng"/>
</resources>
```

The `packsLang.xml_eng` file: file:

```
<langpack>
  <str id="myApplication" txt="Main
  Application"/>
  <str id="myApplication.description"
  txt="A description of my main application"/>
  [...]
</langpack>
```

The following sections describe the tags available for a `<pack>` section. `<description>` - pack description h3. The contents of the `<description>` tag describe the pack contents. This description is displayed if the user highlights the pack during installation. `<depends>` - pack dependencies h3. This can be used to make this pack selectable only to be installed only if some other is selected to be installed. The pack can depend on more than one by specifying more than one `<depends>` elements. Circular dependencies are not supported and the compiler reports an error if one occurs.

This tag takes the following attribute:

- `packname`: The name of the pack that it depends on `<os>` - OS restrictions h3. It is possible to restrict a panel to a certain list of operating systems. This tag takes the following attributes:
  - `family`: `unix`, `windows` or `mac`
  - `name`: the exact OS name (ie `Windows`, `Linux`, ...)
  - `version`: the exact OS version (see the JVM `os.version` property)
  - `arch`: the machine architecture (see the JVM `os.arch` property). `<updatecheck>` h3. This feature can update an already installed package, therefore removing superfluous files after installation. Here's how this

feature author (Tino Schwarze) described it on the IzPack development mailing-list:

> Each pack can now specify an `<updatecheck>` tag. It supports a subset of ant fileset syntax, e.g.:

```
<updatecheck>
  <include name="lib/**" />
  <exclude name="config/local/**" />
</updatecheck>
```

> If the paths are relative, they will be matched relative to `$INSTALL_PATH`. Update checks are only enabled if at least one `<include>` is specified. See `com.izforge.izpack.installer.Unpacker` for details. `<file>` - add files or directories h3. The `<file>` tag specifies a file (a directory is a file too) to include into the pack. It takes the following attributes:

- `src`: the file location (relative path) - if this is a directory its content will be added recursively. It may contain previously defined static variables (see `<variables>`).
- `targetdir`: the destination directory, could be something like `$INSTALL_PATH/subdirX`
- `os`: can optionally specify a target operating system (`unix`, `windows`, `mac`) - this means that the file will only be installed on its target operating system
- `override`: if `true` then if the file is already installed, it will be overwritten (use `false` otherwise). Alternative values: `asktrue` and `askfalse` -ask the user what to do and supply default value for non-interactive use. Another possible values is `update`. It means that the new file is only installed if it's modification time is newer than the modification time of the already existing file (note that this is not a reliable mechanism for updates - you cannot detect whether a file was altered after installation this way.) By default it is set to `update`.
- `blockable` : Defines whether and how blocked target files on Windows should be recognized. This might result in pending file operations which require a system reboot. The reboot behavior at the end of an installation for pending file operations can be set using the nested `rebootaction_` in the `info` element. See above for possible reboot options. Possible values:
  - `none` (default): No recognition of blocked target files will be done at all, this is the default behavior of previous IzPack versions.
  - `auto`: Automatic recognition of a blocked target file by the operating system, resulting in leaving a pending file operation to be finished after system reboot. Using `auto` this applies only for files that are really blocked, the other files are copied normally, which can result in mixed, old and new target files at the end of the installation, unless the system won't be really rebooted.
  - `force`: Forces target file to be always assumed a blocked, resulting in leaving a pending file operation to be finished after system reboot. Using `force` this applies for each file, regardless whether it is really blocked during installation. This makes sense if you don't want to mix files old and new files at the end of the installation to not disturbing a running process, but having the complete set of target files installed after system reboot.Notes:
  1. Using `blockable` does not necessarily force you to limit such files on Windows systems. For multi-platform installations there is a compiler warning thrown that `blockable` will be ignored on non-Windows systems.
  2. The native library `WinSetupAPI` must be explicitly included using this feature. For setting up the according reboot behaviour see the `rebootaction_` element. For getting more information see the chapter *Advanced features*.

- `unpack`: if `true` and the file is an archive then its content will be unpacked and added as individual files
- `condition`: an id of a condition which has to be fulfilled to install this file `<additionaldata> h3`. This tag can also be specified in order to pass additional data related to a file tag for customizing.
- `<key>`: key to identify the data
- `<value>`: value which can be used by a custom action `<singlefile>` - add a single file h3. Specifies a single file to include. The difference to `<file>` is that this tag allows the file to be renamed, therefore it has a `target` attribute instead of `targetdir`.
- `src`: the file location (relative path). It may contain previously defined static variables (see `<variables>`).
- `target`: the destination file name, could be something like `$INSTALL_PATH/subdirX/fileY`
- `os`: can optionally specify a target operating system (`unix`, `windows`, `mac`) - this means that the file will only be installed on its target operating system
- `override`: see `<file>` for description
- `blockable`: see `<file>` for description
- `condition`: an id of a condition which has to be fulfilled to install this file

A `<additionaldata>` tag can also be specified for customizing. `<fileset>`: add a fileset h3. The `<fileset>` tag allows files to be specified using the powerful Jakarta Ant set syntax. It takes the following parameters:

- `dir`: the base directory for the fileset (relative path)
- `targetdir`: the destination path, works like for `<file>`
- `casesensitive`: optionally lets you specify if the names are case-sensitive or not - takes `yes` or `no`
- `defaultexcludes`: optionally lets you specify if the default excludes will be used - takes `yes` or `no`.
- `os`: specifies the operating system, works like for `<file>`
- `override`: see `<file>` for description
- `blockable`: see `<file>` for description (applied for all files in the fileset)
- `includes`: comma- or space-separated list of patterns of files that must be included; all files are included when omitted. This is an alternative for multiple include tags.
- `excludes`: comma- or space-separated list of patterns of files that must be excluded; no files (except default excludes) are excluded when omitted. This is an alternative for multiple exclude tags.
- `condition`: an id of a condition which has to be fulfilled to install the files in this fileset

You specify the files with `<include>` and `<exclude>` tags that take the `name` parameter to specify the Ant-like pattern :

- `*` : means any subdirectory
- `?` : used as a wildcard.

Here are some examples of Ant patterns :

- `<include name="lib"/>` : will include `lib` and the subdirectories of `lib`

```

<exclude name="/.java"/> : will exclude any file in any directory starting from the base path
ending by .java
<include name="lib/*.jar"/> : will include all the files ending by .jar in lib
<exclude name="lib/*/FOO"/> : will exclude any file in any subdirectory starting from lib w
hose name contains FOO.

```

There are a set of definitions that are excluded by default file-sets, just as in Ant. IzPack defaults to the Ant list of default excludes. There is currently no equivalent to the `<defaultexcludes>` task. Default excludes are:

```

**/*\~{ }
**/\##*\#
**/. \##*
**/%*%
**/. \_ *
**/CVS
**/CVS/**
**/.cvsignore
**/SCCS
**/SCCS/**
**/vssver.scc
**/.svn
**/.svn/**
**/.DS\_Store

```

A `<additionaldata>` tag can also be specified for customizing. `<parsable>` - parse file(s) after installation h3. Files specified by `<parsable>` are parsed after installation and may have variables substituted.

- `targetfile` : the file to parse, could be something like `$INSTALL_PATH/bin/launch-script.sh` A slash will be changed to the system dependant path separator (e.g. to a backslash on Windows) only if no backslash masks the slash. No longer mandatory as a fileset of files can be used.
- `type` : specifies the type (same as for the resources) - the default is `plain`
- `encoding` : specifies the file encoding
- `os` : specifies the operating system, works like for `<file>`
- `condition` : an id of a condition which has to be fulfilled to parse this file

One or more fileset tags can be used inside `parsable` to specify multiple files at once. `<executable>` - mark file executable or execute it h3. The `<executable>` tag is a very useful thing if you need to execute something during the installation process. It can also be used to set the executable flag on Unix-like systems. Here are the attributes :

- `targetfile` : the file to run, could be something like `$INSTALL_PATH/bin/launch-script.sh` Slashes are handled special (see attribute `targetfile` of tag `<parsable>`).
- `class` : If the executable is a jar file, this is the class to run for a Java™ program
- `type` : `bin` or `jar` (the default is `bin`)
- `stage` : specifies when to launch : `postinstall` is just after the installation is done, `never` will never launch it (useful to set the `+x` flag on Unix). `uninstall` will launch the executable when the application is uninstalled. The executable is executed before any files are deleted. `never` is the default value.
- `failure` : specifies what to do when an error occurs : `abort` will abort the installation process, `ask` (default) will ask the user what to do and `warn` will just tell the user that something is wrong
- `os` : specifies the operating system, works like for `<file>`

- `keep` : specifies whether the file will be kept after execution. The default is to delete the file after it has been executed. This can be changed by specifying `keep="true"`.
- `condition`: an id of a condition which has to be fulfilled to execute this file

A `<args>` tag can also be specified in order to pass arguments to the executable:

- `<arg>`: passes the argument specified in the `value` attribute. Slashes are handled special (see attribute `tar` `getfile` of tag `<parsable>`). `<os>` - make a file OS-dependent. The `<os>` tag can be used inside the `<file>`, `<fileset>`, `<singlefile>`, `<parsable>`, `<executable>` tags to restrict its effect to a specific operating system family, architecture or version using the following attributes:
  - `family`: `unix`, `windows`, `mac` to specify the operating system family
  - `name`: the operating system name
  - `version`: the operating system version
  - `arch`: the operating system architecture (for instance the Linux kernel can run on `i386`, `sparc`, and so on)

Here's an example installation file fragment:

```

<packs>

(...)

  <pack name="Core" required="yes">

    (...)

      <executable
targetfile="$INSTALL_PATH/bin/compile"
stage="never">
        <os family="unix"/>
      </executable>

    (...)

  </pack>

(...)

</packs>

```

The Native Element `<native>` h3. Use this if you want to use a feature that requires a native library. The native libraries are placed under `bin/native/...`. There are 2 kinds of native libraries : the iZPACK libraries and the third-party ones. The IzPack libraries are located at `bin/native/izpack`, you can place your own libraries at `bin/native/3rdparty`. It is possible to place a native library also into the uninstaller.

It is usable from CustomActions. If one or more are referenced for it, the needed support classes are automatically placed into the uninstaller. To place it only on operating systems for which they are build, it is possible to define an OS restriction. This restriction will only be performed for the uninstaller. The markup takes the following attributes :

- `type` : `izpack` or `3rdparty`
- `name` : the library filename
- `stage` : stage where to use the library (`install|uninstall|both`)
- `src` : source directory where to find the library to build the installer

Note for developers: `com.izforge.izpack.util.Librarian.loadLibrary()` must be used to load custom native libraries, as it performs some housekeeping. `<os>` - make a library OS-dependent h3. The `<os>` tag can be used to restrict the inclusion into the uninstaller to a specific operating system family, architecture or version. The inclusion into the installer will be always done.

Here's a sample :

```
<native type="izpack" name="ShellLink.dll">
  <os family="windows" />
</native>
```

The Jar Merging Element `<jar>` h3. If you adapt iZPACK for your own needs, you might need to merge the content of another jar file into the jar installer. For instance, this could be a library that you need to merge. The `<jar>` markup allows you to merge the raw content of another jar file into the installer and the uninstaller. It is necessary that the paths in the jars are unique because only the contained files of the jar are added to the installer jar, not the jar file self. The attributes are:

- `src` : the path at compile time
- `stage`: stage where to use the contents of the additional jar file (install|uninstall|both)

A sample :

```
<jar src=" ../nicelibrary.jar" />
```

## XInclude-style constructs

The `xi:include` element is used to include xml or text documents in your configuration files.

The `xi:include` element can be used anywhere in pretty much any of the xml files used by IzPack. It is supported by the `javax.DocumentBuilder`. It follows the XInclude recommendation produced by the W3C (<http://www.w3.org/TR/xinclude/>) and should be able to be used as described in that document.

To use the `xi:include`, you have to specify the namespace `xmlns:xi="http://www.w3.org/2001/XInclude"` in your xml.

If a logical error appear in an included file, the line number shown will indicate the `xinclude` element in the main xml file.

The `xi:include` element has the following attributes:

- `href`  
The location of the file to include. If this is a relative path (e.g. `href='../bob.xml'`) then the file is resolved relative to the document that includes the `xinclude` element. The `href` element can be a remote url (e.g. `href='http://example.com/file.txt'`) but this is not recommended.
- `parse`  
Indicates that the included file should be treated as xml, forcing the included file to be parsed (which results in all `xinclude` elements in the included file to also be included), or as text which includes the specified file as a

text node. It can have a value of 'xml' or 'text' and defaults to 'xml' if omitted.

- xpointer

In case in parse="xml", you can use xpointer to get a part of the included xml. The details of the xpointer framework can be found <http://www.w3.org/TR/2003/REC-xptr-framework-20030325/>.

- encoding

This is used when the parse attribute is 'text' to specify the character encoding of the included text document. It has no effect if parse is 'xml'.

- accept

Used if the href attribute specifies a url accessible via HTTP. The value of this attribute will be added as the accept header on the HTTP request. The fallback element h3. The xi:fallback element is used to provide a fallback if the xi:include element fails. It can be empty or can contain a valid xml. If it is empty then a failure to include the specified document is suppressed. If it contains a xml then a failure to include the specified document causes the xml specified in the fallback is parsed (evaluating nested xinclude elements) and inserted into the document in its place.

For example the following use of xi:include

```
<?xml version="1.0" encoding="iso-8859-1"
standalone="yes" ?>
<aaa
xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="does-not-exist.xml">
    <xi:fallback>
      <bbb/>
    </xi:fallback>
  </xi:include>
</aaa>
```

will result in

```
<?xml version="1.0" encoding="iso-8859-1"
standalone="yes" ?>
<aaa>
  <bbb/>
</aaa>
```

if the file 'does-not-exist.xml' does indeed not exist. The xfragment element h1.

The xfragment element allows document fragments (xml documents without a single top level element) to be included. Simply use the xfragment as the top level element in the included document. It will be removed when the document is included.

e.g. If I want to use xi:include to include a file containing the following fragments:

```
<ffff>
  <gggg>hello</gggg>
</ffff>
<hhhh>
  <iiii>there</iiii>
</hhhh>
```

I could use the xfragment element to enable it

```
<?xml version="1.0" encoding="iso-8859-1"
standalone="yes" ?>
<xfragment>
  <ffff>
    <gggg>hello</gggg>
  </ffff>
  <hhhh>
    <iiii>there</iiii>
  </hhhh>
</xfragment>
```