# Plugin Documentation

A Plugin consists of a zip-archive file containing plugin.xml and upgrade.xml files; optionally, the Plugin may also contain a plugin.sig and/or any other scripts/resources needed by the Plugin.

The plugin.xml file is the Plugin descriptor file. It is an xml file conforming to the PluginXMLSchema.xsd schema definition. The plugin.xml document has two attributes on the root element: 'id' and 'type.' The 'id' attribute specifies an unchanging value which identifies the Plugin, and the 'type' attribute specifies the type of Plugin (currently supported types are SOURCE and INTEGRATION). These two attributes can't be changed across versions of the Plugin. The root element has several required child elements: 'version', 'name', and 'description.' Beyond these basic attributes and elements, a Plugin can also have some additional structures which may vary in number and specific attributes, depending upon the type of Plugin being implemented. The common components are 'property-groups' and 'step-types'.

# Plugin Properties

The property-group and properties elements (see Plugin Step Types) define what user-configured properties will be passed to the step, as well as how those properties will be presented to the end-user.

The property-group element is different from the properties element, in that it has type attribute as well as description and validation elements. The number and types of property-groups present in a Plugin are determined by the type of the Plugin (see Plugin Types).

There are a variety of possible property elements. Each property element has a name, label, description and is optionally required; it may also be marked to be passed as a native environment variables:

- **property-text and property-textarea.** These properties differ only in how they are presented to the user: both accept arbitrary text (with property-textarea providing a larger input area than property-text).
- **property-secure.** Similar to property-text, except that it is represented as a password-style field. Its value is redacted from all output from the step.
- **property-checkbox.** Displays a checkbox to the user. If the user checks the box, then the value of 'true' will be used for the property; otherwise the property is not set.
- **property-select.** Requires a list of one or more child values that the user may select among; renders as a dropdown.
- **property-group-ref.** A select property that is populated with other configured property-groups from this Plugin of the given type.

The 'validation' script element has a 'file' and a 'lang' attribute. The file attribute is the path for the validation script within the Plugin file. The 'lang' attribute denotes the language of the validation script. The validation script is used to perform both validation at configuration time and at runtime for the step (see Validation Script).

# Plugin Step Types

The 'step-type' elements describe the steps which the Plugin is providing. Step-types have a name, description, one or more tags, any number of properties, an optional custom property-validation script, an optional interpreter, and a script.

The tag is used to determine where the step will show up in the step-folder-tree when users are adding steps to a job. A tag can denote sub-directories by incorporating the '/' character for folder-boundaries.

The properties element defines what properties users will configure to be passed to the step, as well as how those properties will be present to user (see Plugin Properties).

The 'validation' script element has a 'file' and a 'lang' attributes. The file attribute is the path for the validation script within the Plugin file. The 'lang' attribute denotes the language of the validation script. The validation script is used to perform both validation at configuration time and at runtime for the step (see Validation Script).

The interpreter element (if present) contains the name of executable to use to run the script (see script element below).

The script element has a 'file' attribute which denotes the path to the script or executable within the Plugin file which will be run for this step.

# Plugin Sub-types

StepTypes may have one of a set of sub-types: All of the 'scm.*' sub-types must have a property which is a hidden property-group-reference named "source" to a source-type property-value-group:

- **scm.populate.** Must have a property 'date' which can receive a date to base the checkout upon
- **scm.cleanup.** <no specific requirements>
- **scm.changelog.** Must have properties 'startDate' and 'endDate' which can receive dates to define the interval for which a changelog should be produced
- **scm.label.** Must have properties 'label' and 'message' which can receive the label to create and the message to use during the action
- **scm.changelog.quietperiod.** Must have properties 'startDate' and 'endDate' which can receive dates to define the interval to examine for changes. The latest change date in this interval should be set as a property 'latest-change' on the current step via ahptool.

# Plugin Types

There are currently two types of Plugins supported:

- **INTEGRATION:** Integration Plugins are a catch-all type for bundling together a set of related steps which are not related to Source Control Management systems. An integration Plugin may have up to 1 property-group element which must be of type 'integration'. This property group element is used to enable users define Integration objects in AnthillPro at the System level which may be reused by the steps within this Plugin.
- **SOURCE:** Source Plugins are designed to supply new SCM integrations for AnthillPro. A source Plugin must have exactly one 'source' type and exactly one 'repo' property-group. Also, the 'source' type property-value group must have a property-group-reference named 'repo' which is hidden; it must also reference a 'repo' type property-value group. Additionally, the source Plugin must also supply step-types with sub-type 'scm.populate', 'scm.cleanup', 'scm.changelog', 'scm.label', and 'scm.changelog.quietperiod' (see Plugin Sub-types).

  Plugins also support global repository triggers for the source-type integrations. This allows you set up one trigger in your SCM and have AnthillPro determine which projects need to build based on the information passed along with the trigger request. To use a global repository trigger, you must define it in your Plugin. See Global Repository Triggers for Plugins.

# Plugins and AHPTool Schemas

When writing a Plugin, you will most likely be using AHPTool to control the communication between the server and the agent in the context of a running workflow. Because AHPTool can look up or set properties at the system, step, request, job, Build Life and agent levels -- as well as upload Test, Coverage, Analytics, or Issue data -- it provides an excellent integration point for writing Plugins. The available Plugin steps listed above can use AHPTool to communicate between the agent and the server, and follow the schemas below.

## Issues Schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:complexType name="Issues">
     <xsd:sequence>
         <xsd:element name="issue" type="Issue" minOccurs="0" maxOccurs="unbounded"/>
     </xsd:sequence>
     <xsd:attribute name="date" type="xsd:string" use="required"/>
 </xsd:complexType>
 <xsd:complexType name="Issue">
     <xsd:sequence>
         <xsd:element name="name" type="xsd:string" minOccurs="1"/>
         <xsd:element name="type" type="xsd:string" minOccurs="0"/>
         <xsd:element name="status" type="xsd:string" minOccurs="0"/>
         <xsd:element name="description" type="xsd:string" minOccurs="0"/>
     </xsd:sequence>
     <xsd:attribute name="issue-tracker" type="xsd:string" use="required"/>
     <xsd:attribute name="id" type="xsd:string" use="required"/>
     <xsd:attribute name="change-id" type="xsd:long" use="optional"/>
 </xsd:complexType>
 <xsd:element name="issues" type="Issues"/>
</xsd:schema>
```

## Properties Schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <xsd:complexType name="Properties">
     <xsd:sequence>
         <xsd:element name="property" type="Property" minOccurs="0"
              maxOccurs="unbounded"/>
     </xsd:sequence>
 </xsd:complexType>

 <xsd:complexType name="Property">
     <xsd:simpleContent>
         <xsd:extension base="xsd:string">
             <xsd:attribute name="name" type="xsd:string" use="required"/>
             <xsd:attribute name="secure" type="xsd:boolean"/>
         </xsd:extension>
     </xsd:simpleContent>
 </xsd:complexType>

 <xsd:element name="properties" type="Properties"/>

</xsd:schema>
```

## SCM Change Log Schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
<xsd:complexType name="ChangeLog">
    <xsd:sequence>
        <xsd:element name="log-info"   type="ChangeLogInfo" minOccurs="0"
             maxOccurs="1"/>
        <xsd:element name="change-set" type="ChangeSet" minOccurs="0"
             maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ChangeLogInfo">
    <xsd:sequence>
        <xsd:element name="build-life" type="xsd:long" minOccurs="1"
             maxOccurs="1"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="ChangeSet">
    <xsd:sequence>
        <xsd:element name="repository-id" type="xsd:string" minOccurs="1"
             maxOccurs="1"/>
        <xsd:element name="repository-type" type="xsd:string" minOccurs="1"
             maxOccurs="1"/>
        <xsd:element name="anthill-id" type="xsd:long" minOccurs="0"
             maxOccurs="1"/>
        <xsd:element name="id" type="xsd:string" minOccurs="0"
             maxOccurs="1"/>
        <xsd:element name="user" type="xsd:string" minOccurs="1"
             maxOccurs="1"/>
        <xsd:element name="module" type="xsd:string" minOccurs="0"
             maxOccurs="1"/>
        <xsd:element name="branch" type="xsd:string" minOccurs="0"
             maxOccurs="1"/>
        <xsd:element name="date" type="date_time" minOccurs="1"
             maxOccurs="1"/>
        <xsd:element name="file-set" type="FileSet" minOccurs="1"
             maxOccurs="1"/>
        <xsd:element name="properties" type="Properties" minOccurs="1"
             maxOccurs="1"/>
        <xsd:element name="comment" type="xsd:string" minOccurs="0"
             maxOccurs="1"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="FileSet">
    <xsd:sequence>
        <xsd:element name="file" type="Change" minOccurs="1"
             maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="Change">
    <xsd:simpleContent>
        <xsd:extension base="xsd:string">
            <xsd:attribute name="change-type" type="change_type"/>
            <xsd:attribute name="revision-number" type="xsd:string"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="Properties">
    <xsd:sequence>
        <xsd:element name="property" type="Property" minOccurs="0"
             maxOccurs="unbounded"/>
```

```
    </xsd:sequence>
 </xsd:complexType>

 <xsd:complexType name="Property">
    <xsd:sequence>
        <xsd:element name="name" type="xsd:string" minOccurs="1" maxOccurs="1"/>
        <xsd:element name="value" type="xsd:string" minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
 </xsd:complexType>

 <xsd:simpleType name="change_type">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="A"/>
        <xsd:enumeration value="M"/>
        <xsd:enumeration value="D"/>
    </xsd:restriction>
 </xsd:simpleType>

 <!-- "yyyy-MM-dd HH:mm:ss Z" -->
 <xsd:simpleType name="date_time">
    <xsd:restriction base="xsd:string">
        <xsd:pattern value="\d{4}-([0][1-9]|[1][0-2])-([0][1-9]|[2][0-9]|[3][0-1]) (
            [0-1][0-9]|[2][0-3]):[0-5][0-9]:[0-5][0-9].[0-9]+ .+"/>
    </xsd:restriction>
 </xsd:simpleType>

 <xsd:element name="change-log" type="ChangeLog"/>

</xsd:schema>
```

# Source-analytics Schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <xsd:complexType name="SourceAnalytics">
    <xsd:sequence>
        <xsd:element name="finding" type="Finding" minOccurs="0"
            maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="type" type="xsd:string" use="required"/>
    <xsd:attribute name="buildLifeId" type="xsd:long" use="required"/>
    <xsd:attribute name="urlLink" type="xsd:string" use="optional"/>
    <xsd:attribute name="findingUrlLink" type="xsd:string" use="optional"/>
 </xsd:complexType>

 <xsd:complexType name="Finding">
    <xsd:sequence>
        <xsd:element name="id" type="xsd:string" minOccurs="0" maxOccurs="1"/>
        <xsd:element name="file" type="xsd:string" minOccurs="0" maxOccurs="1"/>
        <xsd:element name="line" type="xsd:long" minOccurs="0" maxOccurs="1"/>
        <xsd:element name="name" type="xsd:string" minOccurs="0" maxOccurs="1"/>
        <xsd:element name="severity" type="xsd:string" minOccurs="0"
            maxOccurs="1"/>
        <xsd:element name="description" type="xsd:string" minOccurs="0"
            maxOccurs="1"/>
        <xsd:element name="status" type="xsd:string" minOccurs="0" maxOccurs="1"/>
    </xsd:sequence>
 </xsd:complexType>

 <xsd:element name="analytics" type="SourceAnalytics"/>
```

```
</xsd:schema>
```

# Test-coverage Report Schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <xsd:complexType name="CoverageReport">
     <xsd:sequence>
         <xsd:element name="coverage-groups" type="CoverageGroup" minOccurs="0"
              maxOccurs="unbounded"/>
     </xsd:sequence>
     <xsd:attribute name="name" type="xsd:string" use="required"/>
     <xsd:attribute name="type" type="xsd:string" use="required"/>
     <xsd:attribute name="job-id" type="xsd:long" use="required"/>
     <xsd:attribute name="line-percentage" type="xsd:double" use="optional"/>
     <xsd:attribute name="method-percentage" type="xsd:double" use="optional"/>
     <xsd:attribute name="branch-percentage" type="xsd:double" use="optional"/>
 </xsd:complexType>

 <xsd:complexType name="CoverageGroup">
     <xsd:attribute name="name" type="xsd:string" use="required"/>
     <xsd:attribute name="line-percentage" type="xsd:double" use="optional"/>
     <xsd:attribute name="method-percentage" type="xsd:double" use="optional"/>
     <xsd:attribute name="branch-percentage" type="xsd:double" use="optional"/>
     <xsd:attribute name="complexity" type="xsd:double" use="optional"/>
 </xsd:complexType>

 <xsd:element name="coverage-report" type="CoverageReport"/>

</xsd:schema>
```

# Test Report Schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <xsd:complexType name="TestReport">
     <xsd:sequence>
         <xsd:element name="test-suite" type="TestSuite" minOccurs="0"
              maxOccurs="unbounded"/>
     </xsd:sequence>
     <xsd:attribute name="name" type="xsd:string" use="required"/>
     <xsd:attribute name="type" type="xsd:string" use="required"/>
     <xsd:attribute name="job-id" type="xsd:long" use="required"/>
     <xsd:attribute name="successes" type="xsd:long" use="optional"/>
     <xsd:attribute name="failures" type="xsd:long" use="optional"/>
 </xsd:complexType>

 <xsd:complexType name="TestSuite">
     <xsd:sequence>
         <xsd:element name="test" type="Test" minOccurs="0" maxOccurs="unbounded"/>
     </xsd:sequence>
     <xsd:attribute name="name" type="xsd:string" use="required"/>
     <xsd:attribute name="successes" type="xsd:long" use="required"/>
     <xsd:attribute name="failures" type="xsd:long" use="required"/>
 </xsd:complexType>

 <xsd:complexType name="Test">
     <xsd:sequence>
         <xsd:element name="message" type="xsd:string" minOccurs="0"/>
     </xsd:sequence>
```

```
    <xsd:attribute name="name" type="xsd:string" use="required"/>
    <xsd:attribute name="class-name" type="xsd:string" use="optional"/>
    <xsd:attribute name="result" type="TestResult" use="required"/>
    <xsd:attribute name="time" type="xsd:unsignedInt" use="optional"/>
 </xsd:complexType>

 <xsd:simpleType name="TestResult">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="success"/>
        <xsd:enumeration value="failure"/>
        <xsd:enumeration value="error"/>
    </xsd:restriction>
 </xsd:simpleType>

 <xsd:element name="test-report" type="TestReport"/>

</xsd:schema>
```

## Working Directory Schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <xsd:complexType name="WorkDir">
   <xsd:simpleContent>
     <xsd:extension base="xsd:string">
       <xsd:attribute name="scope" type="Scope" use="required"/>
     </xsd:extension>
   </xsd:simpleContent>
 </xsd:complexType>

 <xsd:simpleType name="Scope">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="job"/>
        <xsd:enumeration value="workflow"/>
    </xsd:restriction>
 </xsd:simpleType>

 <xsd:element name="work-dir" type="WorkDir"/>

</xsd:schema>
```

# Plugin Signing

To load a Plugin into a production AnthillPro server, the Plugin must be signed by Urbancode (denoted by a plugin.sig file). Non-production licensed systems may load unsigned Plugins to facilitate testing prior to submitting the Plugin for signing.

# Plugin Runtime

**Validation Script.** The property validation scripts make use of any supported BSF language. The script is a BSF language based script which is passed all the property values bound to their property names as well as a map called 'errors'. For any values that the validation script determines are invalid, the script will then place a String message into the errors map with the key equal to the invalid property's name. These messages are then displayed to the user asking the user to correct them before proceeding.

**Step Execution.** Steps are executed by invoking the given interpreter with the script file as the last argument. If no interpreter is specified then the script file is launched directly by its canonical path.

# Global Repository Triggers for Plugins

Plugins support global repository triggers for the source-type integrations. This allows you set up one trigger in your SCM and have AnthillPro determine which projects need to build based on the information passed along with the trigger request.

To use a global repository trigger, you must define it in your Plugin. Note that global triggers are supported for Plugin Schema 5 or greater, so if you are using an older schema you will not be able to use a global trigger.

To get started:

1. **Ensure that you have a Plugin Schema 5 (or higher) element in your `plugin.xml`**. It should look something like this:

   ```
   xmlns="http://www.anthillpro.com/PluginXMLSchema_v5"
   ```

   Example:

   ```
   <plugin id="com.dev.urbancode.anthill3.plugin.Git" type="SOURCE"
   xmlns="http://www.anthillpro.com/PluginXMLSchema_v5"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   ```

2. **Add property-script element to repo property group.** The property-script element needs to have a file attribute that points to a script in the Plugin -- this script is responsible for processing any repository trigger requests.

   The name of the element must be `trigger` -- this is how the Plugin system recognize the special type of element. You will also need to specify the scripting language used in the script as part of the lang attribute (groovy, beanshell and javascript are currently supported). For example:

   ```
   <property-script name="trigger" file="git_repo_trigger.groovy" lang="groovy"
   hidden="true">
   ```

   Keep in mind that this script will execute on the server and could have negative effects on the system if the script is using a lot of resources. Use a hidden attribute to make sure the property is not exposed in the repository configuration, thus reducing the chances a user may change the value and break the trigger.

   You can also provide a description element that provides examples of what the SCM trigger scripts could be or what parameters they should pass in order for the trigger script to function correctly. The contents of the description element will be displayed under the trigger tab for each repository, so make sure you use plain text or valid html. In the description you can use `${triggerUrl}` and `${triggerCode}` to have AnthillPro provide the trigger URL and the trigger code for the repository and not have to hard-code it in the Plugin.

   **Example Repo Property Group:**

   ```
   <property-group type="repo">
         <description>A placeholder for using the git scm system.
          The remote repository is configure on a per workflow basis.
         </description>
         <property-script name="trigger" file="git_repo_trigger.groovy"
          lang="groovy" hidden="true">
           <description><![CDATA[
     <span class="bold">Using wget (Unix)</span>
     <pre class="code">
     #!/bin/bash

     TRIGGER_URL="${triggerUrl}"
     CODE="${triggerCode}"
   ```

```
REPO="[insert repo FQN here]"`pwd`
BRANCH=$1
wget -O /dev/null --quiet --no-check-certificate
   --post-data="code=$CODE&repo=$REPO&branch=$BRANCH" "$URL"

# To use curl instead of wget, use this command
# curl --retry 1 -k -d "code=$CODE&repo=$REPO&branch=$BRANCH"
  -o /dev/null "$TRIGGER_URL"
</pre>
]]>
      </description>
    </property-script>
  </property-group>
```

3. **You will also need a script in your Plugin that will run on every trigger request, based on information passed by the SCM.** The script is responsible for providing a list of source configs that match the changes that need to be built. The script is part of the Plugin, preferably in the root. In the script you have access to several objects:

   - **availableSourceConfigSet** -- A hashset of all source configs that reference repositories for this Plugin. The set will not include source configs that ignore repository triggers or are part of inactive workflows or projects.
   - **matchingSourceConfigSet** -- An empty hashset that the script needs to populate with all source configs from the `availableSourceConfigSet` that would be affected by the changes that caused the trigger. In other words, all the source configs that would check-out the changed files.
   - **triggerPropertyMap** -- A map of all the parameters that were part of the HTTP POST request made from the SCM. This would be the place where you would look for data about what's changed if the SCM trigger is configured correctly.
   - **requestPropertyMap** -- An empty map that allows you to specify additional properties to be passed on the request created from the trigger. This gives you additional control over the build and allows you to put any property on the request that can later be used by your workflows/jobs.
   - **log** -- this is an `org.apache.log4j.Logger` instance that allows you to log messages to the server log. The only place those messages would be visible is in **System >Server Settings > Log/Error** tabs.

   The following example of a trigger script assumes the git repo passes repo and branch parameters that contain the path of the repo where the change occurred and which branch (take a look at the description of the example in 2 above):

```
if (repo && branch) {
    for (sc in availableSourceConfigSet) {
        for (module in sc.getPropertyValueGroups()) {
            def scRepo = ParameterResolver.resolveForSourceConfig
            (module.getPropertyValue("remoteUrl")?.getValue(),
             sc)?.toLowerCase()
            def scBranch = ParameterResolver.resolveForSourceConfig
            (module.getPropertyValue("branch")?.getValue()?:'master', sc)
            if (scRepo && scRepo.size() > 0) {
                // strip any trailing \ or / the users might have specified
                while (scRepo[-1] == '/' || scRepo[-1] == '\\') {
                    scRepo = scRepo[0..-1]
                }

                if (scRepo.endsWith(repo) && branch.equalsIgnoreCase
                 (scBranch)) {
                    matchingSourceConfigSet.add(sc)
                    log.debug("Found matching source config from project
                    $sc.project.name with values $scRepo:$scBranch")
```

```
                }
                else {
                    log.debug("Trigger did not match source config from
                    project $sc.project.name with values $scRepo:$scBranch")
                }
            }
        }
    }
}
else {
    log.warn("Could not process Git repo trigger data:
    $triggerPropertyMap - invalid repo and/or branch parameters")
}
```

# Upgrading/Updating Plugins

A Plugin can be updated in one of two ways: Either as an unversioned update or as a fully versioned upgrade. In both cases, the user updating/upgrading the Plugin simply loads the new Plugin file. The user is then presented with an upgrade/update assessment page which will show what changes have been made to the Plugin, and what configuration will be impacted and will no longer validate. The user is then given a final option to accept the upgrade/update or to cancel the process.

## Versioned Plugin Upgrades

To create a versioned upgrade of a Plugin, increment the number in the version element in plugin.xml and create a "migrate" element in the upgrade.xml with the corresponding 'to-version' attribute. This element will then contain the property and step-type structure matching your updated plugin.xml.

The advantage of the versioned upgrade is that you can also incorporate "old" and "default" attributes. The old attribute is the name of a step/property which should be renamed to the current step/property name. The default attribute on properties allows new properties to be introduced with the given default during the upgrade process. Together, the versioned upgrade facilitates more elaborate and robust upgrades to a Plugin.

## Unversioned Plugin Updates

An unversioned update of a Plugin is a simple mechanism for making small changes to a Plugin. There is no special configuration to create a unversioned update. When uploading an unversioned update, any properties or steps that are missing in the new version will be dropped from current configuration. This mechanism is most useful for development of a Plugin and for minor bug-fixes/updates.