# Property Collectors


- What are Property Collectors?
- How are Property Collectors Used?
  - Actions
    - Manually Providing a Tool Path
  - Configuration
    - Actions
    - Stage Options
  - Repositories
- Why Should I Use Property Collectors?
- How Do I Know When My Tool Has Registered With Continua?
- Property Collector Types
- Agent and Server Property Collectors
- Viewing Property Collectors
- Editing Property Collectors
  - Namespace
  - Run On
  - Type
  - Property Name
  - Executable
  - Search Paths
  - Register folder property
  - Extract version info
  - Extra File Properties
- Creating Custom Property Collectors
  - Namespace
  - Run On
  - Type
- Restoring Default Property Collectors
- Continua Cannot Find My Build Tool
- Continua Does Not have a Property Collector for my Tool
  - What is a Property Collector Type?
- How to access properties gathered from Property Collectors
  - FinalBuilder
  - DotNetFramework
  - VisualStudio
  - MSTest
  - File Version
  - Path Access Plugin
  - Path Finder Plugin
  - Registry Key Finder
  - Environment Variable
  - Environment Variables
  - Operating System
  - Operating System
- Agent/Server Properties and Property Collectors
- Actions/Repositories and Property Collector Namespaces
  - Actions
  - Repositories
- Why it's a good idea to use a property collector.


## What are Property Collectors?

Property Collectors gather information on the build tools that are installed on your Continua server and agents. Property Collectors run every time the Continua service starts and they search your server and agents for any build tools that are installed on these machines. If a build tool is found then Continua will create properties on either the server or the agent which will provide Continua information regarding that tool (i.e. path to the tool, tool version, etc.).

There are several types of property collectors, however the Path property collector is the most widely used. The path property collector is used to find where a certain build tool is located. Path property collectors are designed to look in several common directories when trying to find a particular build tool.

For example, lets look at the default Git property collector for the server.

| | | | | |
|---|---|---|---|---|
| Git.Default | Path | Server | Find the executable 'git.exe' in any of the following locations '%PATH%, %PROGRAMFILES%\Git\bin, %PROGRAMFILES(x86)%\Git\bin'. | [Edit] [Delete] |

This property collector is called Git.Default and it attempts to find Git.exe in the following locations on the server:

- %PATH% (This refers to the 'path' environment variable)
- %PROGRAMFILES%\Git\bin (%PROGRAMFILES% refers to the 'programfiles' environment variable)
- %PROGRAMFILES(x86)%\Git\bin (%PROGRAMFILES(x86)% refers to the 'programfiles(x86)' environment variable)

When the property collectors run, our Git property collector will check the above directories for the Git executable and if Continua can find it then a server property called Git.Default.Path which will point to the location of the Git executable. Continua then uses this property to check whether Git is installed on your server.

By default, Continua comes with an extensive list of property collectors that will try to locate the majority of the tools that you will use during your builds. These collectors will search standard install locations where these tools are usually installed. If you have installed any tools in a non-standard directory then you may need to modify these collectors.

## How are Property Collectors Used?

Property Collectors are used every time you use a build tool or access a repository through Continua. Most of the time you will be unaware property collectors are in use due to the extensive list of default property collectors that are included automatically in Continua.

Property Collectors are used in the following areas:

### Actions

When creating or editing an action you may notice the **Using** field at the bottom of the dialog. This **Using** property lists every path property collector that points to the current action tool. For example, if you create an NUnit action, its **Using** property will list every NUnit path property collector (as shown below). By selecting a property collector, you are telling that action to use a specific version of the build tool. For the NUnit example, if we select NUnit 2.6.2 then this action will only ever execute using the NUnit version 2.6.2. This also means that this action (and its parent's stage) will only run on an agent that has NUnit 2.6.2 installed and the executable is locatable by the property collector.

As all actions are executed on an agent, the **Using** property **only includes property collectors that are set to run on agents**. If you create a custom property collector for an action, make sure it is set to run on agents.

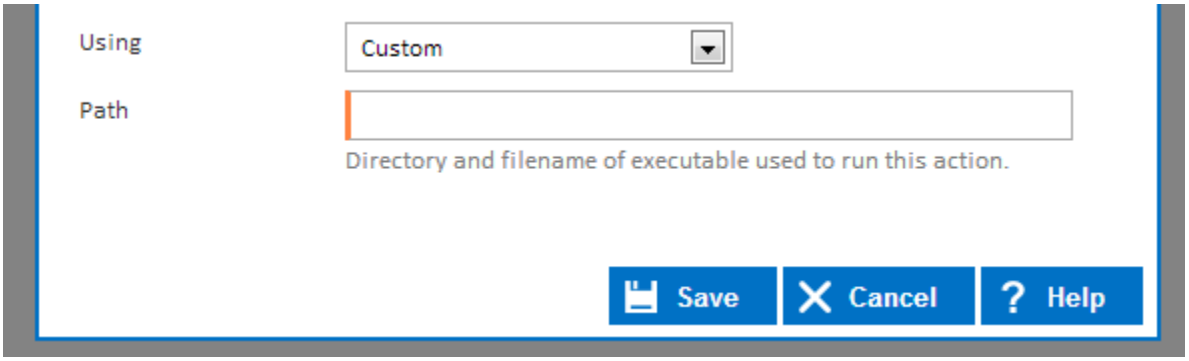
Note: By default, all property collectors listed in the **Using** property are default property collectors and they all have their "Run On" attribute set to Agent. Since they're all set as Agent property collectors, we construct an Agent Compatibility Matrix which shows why an Agent may not be compatible with a stage. Continua determines the compatibility of an agent by checking the Using property of all the actions in that stage. If the selected property collector matches its corresponding property on the agent then that agent is compatible with the current stage. For example, with the NUnit 2.6.2 property collector selected, the compatibility matrix will check which agents have the nunit.2.6.2.path property (This property was automatically generated by the NUnit.2.6.2 property collector). If the agent includes this property then Continua found the NUnit 2.6.2 executable on that agent.

Some simple actions do not have the Using property. These actions run either natively within Continua or Windows and do not require an external tool to run. These actions include the Delay action, Tag Build action action and most File operation actions.

If your build tool (or build tool version) does not appear in the **Using** property (i.e. does not have a property collector in Continua), then you can create a custom property collector that points to your build tool. Creating custom property collectors is described further down this page in the **Create Custom Property Collectors** section.

#### Manually Providing a Tool Path

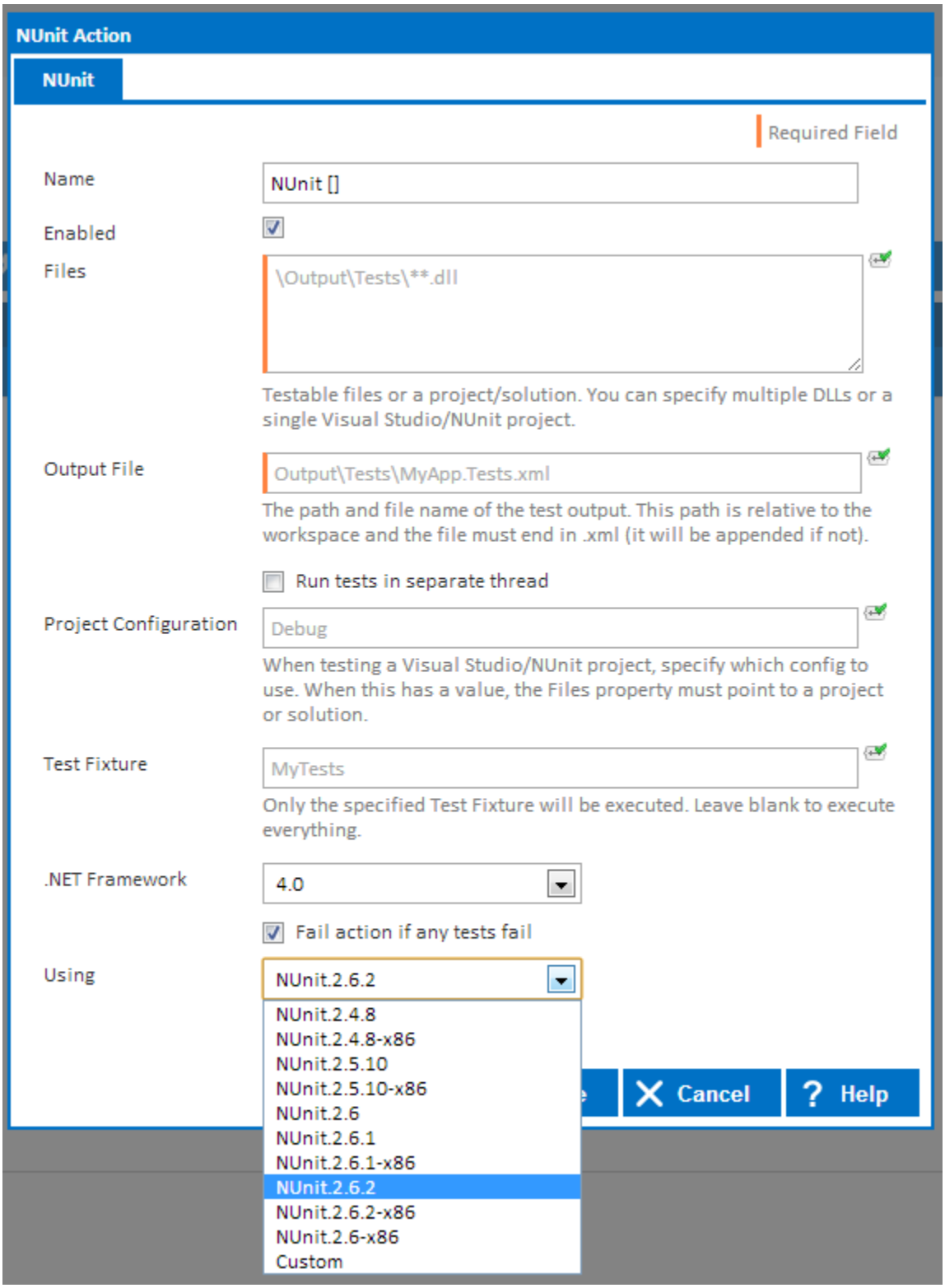
If your build tool does not appear in the **Using** property or you are defining a unique executable for an action, then you can manually set the build tool path of that action. By selecting custom in the **Using** property you will then see the **Path** property appear at the bottom of the dialog (as shown below).




While it is highly recommended that you create a custom property collector, there are some scenarios where you will just want to provide the path to your build tool. Read the **Why Should I Use Property Collectors** section below for reasons why custom property collectors are a good idea.

In addition to the **Using** property, you can manually specify property collector properties by using the Query Syntax in any action.

In addition to the "Using" drop down list, you can manually specify property collector properties by using the Query Syntax in any action. Chances are at some point you will need to use an executable in your workflow that Continua doesn't have an action form, for example FTP. To do this, you would add a property collector for your FTP executable, then use the Execute Program action and specify the property using the Query Syntax. Accessing properties from property collectors using the query syntax is described below. Please note, when specifying properties manually there's no way for Continua to check if an Agent is compatible until the action is executed (at run time). If you want to manually specify a path and have the compatibility of an Agent detected before run time, then you would need to add a Stage Option that checks of the path exists. As an example, your stage option would look something like this: ***$Agent.MyFtpExecutable.Path$*** with the condition drop down set to ***Exists***.




## Configuration

### Actions

When creating an action in the Stage Editor, you may notice a "Using" field at the bottom of the dialog on the first tab for all actions that are backed by an executable. Obviously actions like Delay and Tag Build action don't require executables to perform their job so they won't have a "Using" drop down list.  The "Using" list contains the property collectors that are of use to the action. Once a property collector is selected from this list and the action is saved to the Stage and subsequently the Configuration, a requirement is then set on the Configuration for that property needing to be available before the Configuration can execute. One common theme for all property collectors found in the "Using" drop down in actions is they all use a property collector type that returns a ***Path*** property. By default, all actions will take the property collector assigned to them and only use the ***Path*** property since that's all the action needs... the path to the executable to run the action.

**Note**: Currently all property collectors listed in the "Using" drop down in actions are all default property collectors and all have their "Run On" attribute set to Agent. Since they're all set as Agent property collectors, we can construct an Agent Compatibility Matrix which allows the user to see why an Agent may not be compatible with a Configuration. Determining the compatibility of an Agent is as simple as scanning all actions in a Configuration and seeing if the property collectors they specified to use can find those properties on an Agent.

**Note**: To see how to get your own property collectors listed in the "Using" drop down, a full explanation can be found below.

In addition to the "Using" drop down list, you can manually specify property collector properties by using the Query Syntax in any action. Chances are at some point you will need to use an executable in your workflow that Continua doesn't have an action form, for example FTP. To do this, you would add a property collector for your FTP executable, then use the Execute Program action and specify the property using the Query Syntax. Accessing properties from property collectors using the query syntax is described below. Please note, when specifying properties manually there's no way for Continua to check if an Agent is compatible until the action is executed (at run time). If you want to manually specify a path and have the compatibility of an Agent detected before run time, then you would need to add a Stage Option that checks of the path exists. As an example, your stage option would look something like this: ***$Agent.MyFtpExecutable.Path$*** with the condition drop down set to ***Exists***.

#### Stage Options

Stage Options are one place where you can directly access properties provided by property collectors. By using the Query Syntax, you can use access properties such as the Environment Variables of an Agent or the version of a file if you've setup a File Version property collector for that file. Accessing properties from property collectors using the query syntax is described below.

### Repositories

Repositories are much like actions with their "Using" field containing a list of property collectors. Like actions, Repository property collectors must have a Path property, currently only the Path Finder Plugin property collector type is used for Repository property collectors. What's important to note with Repository property collectors is they're required to run on the Server but not the Agent. The reason for this is, Continua manages your repository from the Continua Server and not Agents which means the Server is the only one that absolutely needs access to the executable that manages the Repository.

## Why Should I Use Property Collectors?

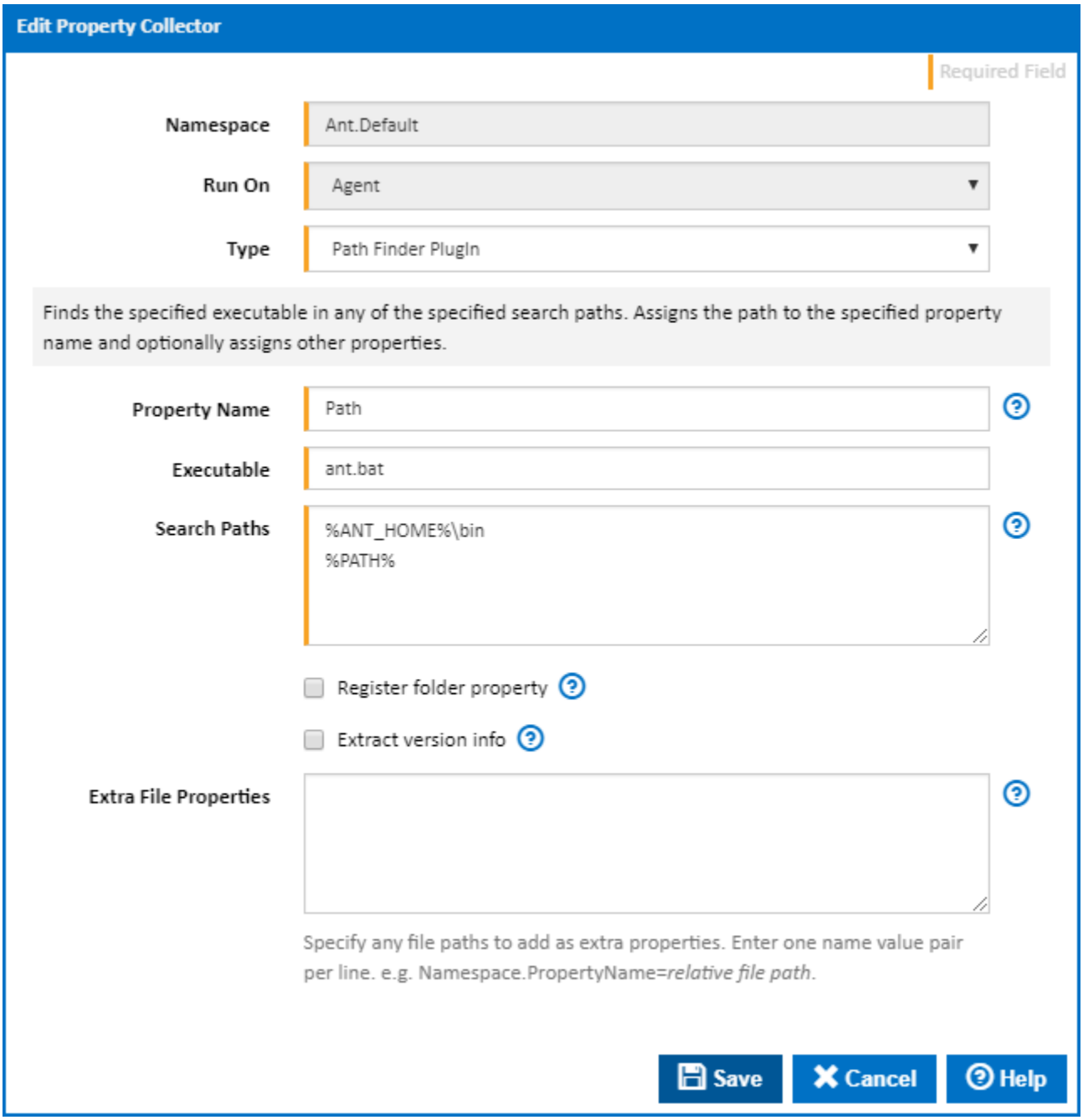
## How Do I Know When My Tool Has Registered With Continua?

## Property Collector Types

## Agent and Server Property Collectors

## Viewing Property Collectors

## Editing Property Collectors




### Namespace

The namespace of the property collector.

### Run On

Property collectors can be run on either the server or an agent.

### Type

The type of the property collector.

### Property Name

The name of the property to assign the full path to.

### Executable

The executable of the property collector.

### Search Paths

One path per line.

**Note:** Environment variables e.g. %HOME% can be used and will be expanded.

### Register folder property

Tick to assign the path to the folder where the executable is found to a property.

### Extract version info

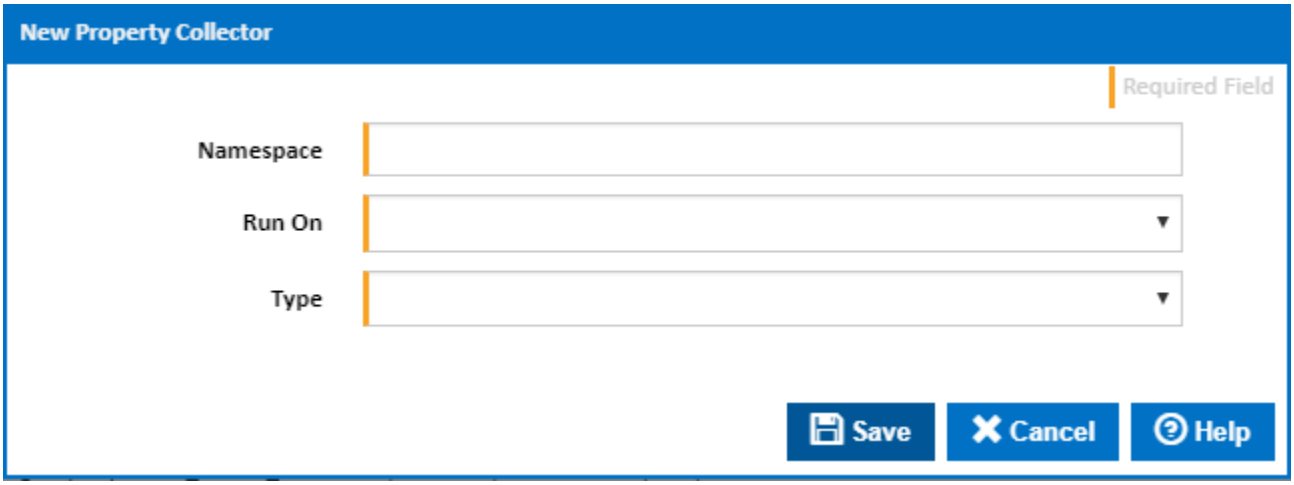
Tick to assign version details for the executable to a Version property.

### Extra File Properties

Specify any file paths to add as extra properties. Enter one name value pair per line. e.g. Namespace.PropertyName=relative file path.

**Note:** The relative file path can be a file name or a file path relative to main executable path. If the file path is omitted then the main executable path will be added to the extra property name.

## Creating Custom Property Collectors




### Namespace

The namespace of the property collector.

### Run On

Property collectors can be run on either the server or an agent.

### Type

The type of the property collector.

## Restoring Default Property Collectors

## Continua Cannot Find My Build Tool

# Continua Does Not have a Property Collector for my Tool

Property Collectors tell Continua to gather certain properties on either an Agent or on the Server. Once a property collector has been defined, it will be sent out to all interested parties (Agents/Server, as defined in the Run On attribute of the collector). The property collector then gathers information on the target and reports those results to the server for later use. The type of property to gather is defined when the property collector is created.

### What is a Property Collector Type?

Currently in Continua, there's 11 property collector types. Each type returns a value or a set of values. The table below outlines the property collector types and their return values.

| Property Type | Description | Returns (Properties) |
|---|---|---|
| FinalBuilder | Enter the version number of Finalbuilder to check for. | Version, MajorVersion, MinorVersion, RelaseVersion, BuildVersion, Path. |
| DotNetFramework | Select the Framework Version of .NET. | FrameworkPath, FrameworkPathX86,FrameworkPathX64, Path, PathX86, PathX64 |
| VisualStudio | Select the version of Visual Studio to check for. | Path |
| File Version | Check the File/Product Version of the file specified. | Version, MajorVersion, MinorVersion, RelaseVersion, BuildVersion |
| Path Access Plugin | Enter a path to check for its existence. | HasAccess (boolean value) |
| Path Finder Plugin | Enter a file and paths where it can be found. | Path |
| Registry Key Finder | Search for a Registry Key and its Value. | Value of Registry Key Name. |
| Environment Variable | Enter an environment variable to store. | <*EnvironmentVariableName*> |
| Environment Variables | Stores a list of all environment variables. | All environment variables. |
| Operating System | Stores a list of important Operating System information. | Platform, Runtime, Name, Arch, ServicePack, HostName<br><br>IsWindowsOS (bool), IsMacOS (bool), IsLinuxOS (bool) |
| ASP.NET MVC | Checks for a certain version of ASP.NET MVC libraries. | IsInstalled (boolean value) |

## How to access properties gathered from Property Collectors

In the above section, it was mentioned that properties gathered from Property Collectors could be accessed using the Query Syntax. The first step to accessing these properties is to specify the source of the property, that being ***Server*** or ***Agent***. The sections below contain an example of each property collector type, some made up values and how to access the resulting properties. The resulting properties match up with the return values from the table above.

**Note**: All expression examples below assume the property collector was setup to run on an Agent, simply replace Agent with Server if you created a server property collector.

FinalBuilder

Assuming the namespace when creating the property collector was ***FB***.

| Properties |
|---|

| |
|---|
| $Agent.FB.Version$ |
| $Agent.FB.MajorVersion$ |
| $Agent.FB.MinorVersion$ |
| $Agent.FB.RelaseVersion$ |
| $Agent.FB.BuildVersion$ |
| $Agent.FB.Path$ |

### DotNetFramework

Assuming the namespace when creating the property collector was ***NET***.

| Properties |
|---|
| $Agent.NET.FrameworkPath$ (default) |
| $Agent.NET.FrameworkPathX86$ |
| $Agent.NET.FrameworkPathX64$ |
| $Agent.NET.Path$ (deprecated, retained for backwards compatibility only) |
| $Agent.NET.PathX86$ |
| $Agent.NET.PathX64$ |

### VisualStudio

Assuming the namespace when creating the property collector was ***VisualStudio***.

| Properties |
|---|
| $Agent.VisualStudio.Path$ |

### MSTest

Assuming the namespace when creating the property collector was ***MSTest***.

| Properties |
|---|
| $Agent.MSTest.Path$ |

### File Version

Assuming the namespace when creating the property collector was ***MyFile***.

| Properties |
|---|
| $Agent.MyFile.Version$ |
| $Agent.MyFile.MajorVersion$ |
| $Agent.MyFile.MinorVersion$ |
| $Agent.MyFile.RelaseVersion$ |
| $Agent.MyFile.BuildVersion$ |

### Path Access Plugin

Assuming the namespace when creating the property collector was ***MyFile***.

| Properties |
| --- |
| $Agent.MyFile.HasAccess$ |

### Path Finder Plugin

Assuming the namespace when creating the property collector was ***Programs*** and the property name was ***MyFile***.

| Properties |
| --- |
| $Agent.Programs.MyFile$ |

### Registry Key Finder

Assuming the namespace when creating the property collector was ***Registry*** and the property name was ***MyRegVal***.

| Properties |
| --- |
| $Agent.Registry.MyRegVal$ |

### Environment Variable

Assuming the namespace when creating the property collector was ***Env*** and the property name was ***BINPATH***.

| Properties |
| --- |
| $Agent.Env.BINPATH$ |

### Environment Variables

Assuming the namespace when creating the property collector was ***Env***.

| Properties |
| --- |
| $Agent.Env.<*environment_variable*>$ |

The results for this property collector are the environment variables so they're different for every machine.

### Operating System

Assuming the namespace when creating the property collector was ***OS***.

| Properties |
| --- |
| $Agent.OS.Platform$ |
| $Agent.OS.Runtime$ |
| $Agent.OS.Name$ |
| $Agent.OS.Arch$ |
| $Agent.OS.ServicePack$ |
| $Agent.OS.HostName$ |
| $Agent.OS.IsWindowsOS$ |
| $Agent.OS.IsMacOS $ |
| $Agent.OS.IsLinuxOS$ |

Operating System

Assuming the namespace when creating the property collector was ***MVC***.

| Properties |
| --- |
| $Agent.MVC.IsInstalled$ |

## Agent/Server Properties and Property Collectors

After adding a property collector, the resulting property will show up on the Agent or Server's properties list with the actual value. Things like Operating System version will show up and any kind of check for paths/files or file access will display the true values. You won't see the changes in the UI immediately after entering a property collector. The server properties have a refresh button to force the properties to get updated and the Agent will update after polling the server which usually happens every minute or so. If a property doesn't show up, there's a chance you either configured the property collector incorrectly or the Server or that particular Agent simply couldn't find what you told it to look for.

## Actions/Repositories and Property Collector Namespaces

Property Collector Namespaces may seem a bit redundant or verbose but they serve an important purpose. Each action and Repository which use property collectors also define a namespace pattern that is used to determine which property collectors it's interested in. When you create a property collector and give it a namespace which matches the pattern in an action/repository, it will show up in the "Using" list of that action/repository.

The tables below show the patterns for each action/repository and namespace examples which show up in the 'Using' drop down list of actions /repositories.

**Note:** Patterns are case insensitive.

### Actions

| Plugin | Pattern | Namespace Examples |
| --- | --- | --- |
| 7-Zip Create<br>7-Zip Extract | `^7-Zip.*` | 7-Zip.Ver2.1<br>7-Zipper<br>7-zip-8.09 |
| Ant | `^Ant.*` | Ant.9.11<br>Antler<br>ant.V-10.6 |

| | | |
|---|---|---|
| Control Azure Web App<br>Create Azure App Service Plan<br>Create Azure Directory<br>Create Azure File Share<br>Create Azure Function<br>Create Azure Resource Group<br>Create Azure Storage Account<br>Create Azure Storage Container<br>Create Azure Web App<br>Delete Azure App Service Plan<br>Delete Azure Blob<br>Delete Azure Directory<br>Delete Azure File<br>Delete Azure File Share<br>Delete Azure Function<br>Delete Azure Resource Group<br>Delete Azure Storage Account<br>Delete Azure Storage Container<br>Delete Azure Web App<br>Deploy Azure Function<br>Deploy Azure Web App<br>Get Azure Storage Account Keys<br>Upload Azure Blob<br>Upload Azure File<br>Upload Azure Web App | `^Azure.Cli.*` | |
| Bower Install<br>Bower Update | `^Bower.*` | |
| Cake | `^Cake\..*` | |
| Docker Build<br>Docker Command<br>Docker Commit<br>Docker Inspect<br>Docker Pull<br>Docker Push<br>Docker Run<br>Docker Stop<br>Docker Tag | ^Docker.* | |

| | | |
|---|---|---|
| DotNet Add<br>DotNet Build<br>DotNet Pack<br>DotNet Publish<br>DotNet Remove<br>DotNet Restore<br>DotNet Run<br>DotNet Test | `^DotNet.Cli.*` | |
| Fake | `^Fake\..*` | |
| FinalBuilder | `FinalBuilder\..*` | FinalBuilder.8<br>FianlBuilder.8.09<br>finalbuilder.600 |
| Grunt | `^Grunt\..*` | |
| Gulp | `^Gulp\..*` | |
| Karma | `^Karma\..*` | |
| Maven | `Maven\..*` | |
| Mocha | `^Mocha\..*` | |
| MSBuild | `^msbuild\..*` | MSBuild.2.0<br>MSBuild.4.0<br>MSBuild.12.0 |
| MSTest | `^MSTest.*` | MSTest.1.22<br>MSTest44<br>mstest3 |
| NAnt | `^NAnt.*` | NAnt10.3<br>nant.9.33<br>nantnant4 |
| NCover 3 | `^NCover\.Console.*` | NCover.Console.3.22<br>NCover.Console9<br>ncover.console1 |
| NCover Reporting | `^NCover\.Reporting.*` | NCover.Reporting.1.33<br>NCover.Reporting4.99<br>ncover.reporting2 |
| NPM Install<br>NPM Pack<br>NPM Publish<br>NPM Update | `^NPM.*` | |

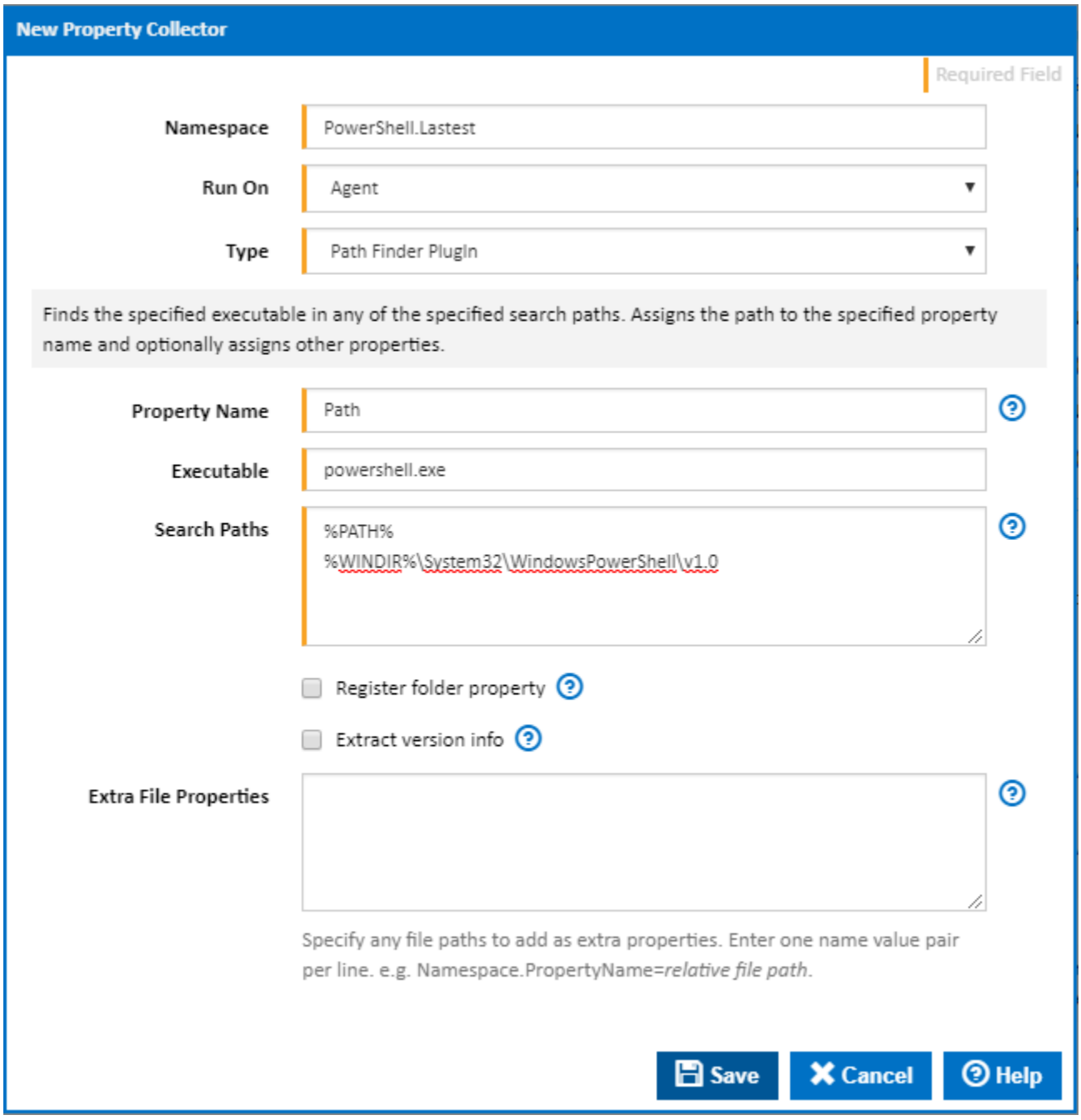
| | | |
|---|---|---|
| NuGet Delete<br>NuGet Install<br>NuGet Pack<br>NuGet Push<br>NuGet Spec<br>NuGet Update | `^NuGet.*` | NuGet.1.99<br>NuGet88<br>nuget23 |
| NUnit | `^NUnit.*` | NUnit.1.22<br>NUnit44<br>nunit3 |
| Octo Pack<br>Octo Push | `^Octo.*` | |
| OpenCover | `^OpenCover\.Console.*` | |
| PowerShell | `^PowerShell.*` | PowerShell.1.6<br>PowerShell99<br>powershell2 |
| Rake | `ruby\.runtime\..*` | |
| Report Generator | `^ReportGenerator\.*` | |
| SQL Package Export<br>SQL Package Extract<br>SQL Package Import<br>SQL Package Publish<br>SQL Package Script | `^SqlPackage\..*` | |
| Visual Studio | `^VisualStudio\..*` | VisualStudio.10<br>VisualStudio.2012<br>visualstudio.win |
| VSTest | `^VSTest.Console*` | |
| XUnit | `^XUnit\..*` | XUnit.1.22<br>XUnit.123<br>XUnit.4.5.6 |

## Repositories

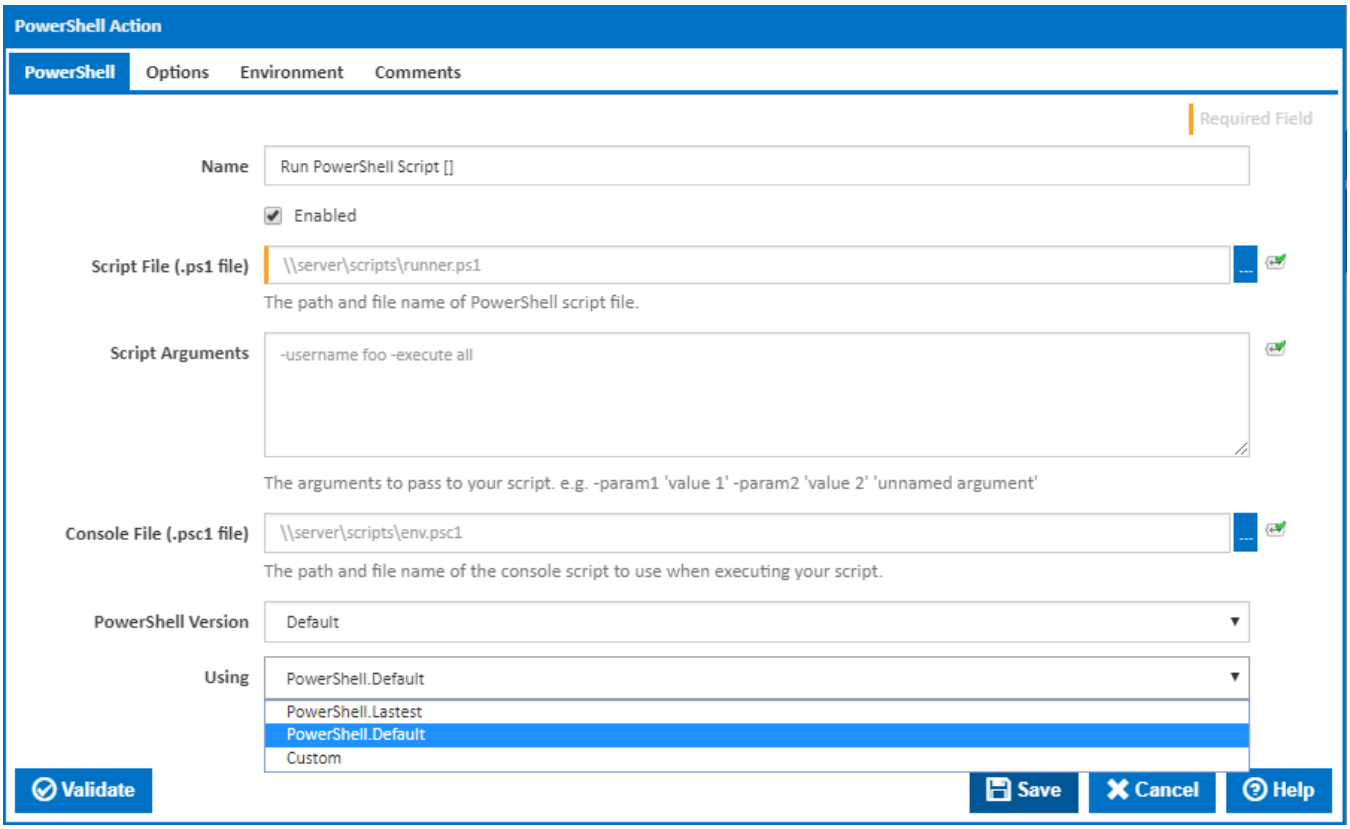
| Plugin | Pattern | Namespace Examples |
|---|---|---|
| Bazaar | `^bazaar\..*` | Bazaar.2.4<br>Bazaar.1<br>bazaar-old |
| Git | `^git\..*` | Git.1.6<br>Git.9<br>git.newest |

| | | |
|---|---|---|
| Mercurial | `^mercurial\..*` | Mercurial.9.4<br>Mercurial.Latest<br>mercurial.2 |
| Perforce | `^perforce\..*` | Perforce.3.6<br>Perforce.old<br>perforce.1 |
| Plastic SCM | `^plasticscm\..*` | |
| Subversion | `^subversion\..*` | Subversion.9.77<br>Subversion.newest<br>subversion.1 |
| Surround SCM | `^surroundscm\..*` | SurroundSCM.1.2<br>SurroundSCM.testing<br>surroundscm.9 |
| Vault | `^vault\..*` | Vault.4.9<br>Vault.WIN<br>vault.8 |

The picture below shows a PowerShell property collector being created. Notice how the namespace value matches the PowerShell pattern in the actions table above.




The next picture shows the property collector that was just created in the "Using" list of the PowerShell action.




## Why it's a good idea to use a property collector.

By not using property collectors you take away a lot of the benefits Continua offers. Unless you take additional precautions you will inevitably end up breaking a build or hindering the performance of builds. The best example of a property collector's benefits is when using an action which requires an executable be installed on the agent. Lets take NUnit for example and assume you haven't used a property collector for your NUnit Action and instead used a custom defined path. Here is where things can go wrong.

1. Continua won't check if the path you provided exists **before the build starts** (with a property collector it will), therefore it won't stop you from running a build. When this happens the build will fail if it can't find the path.
2. Since Continua doesn't check the path before the build starts, it will send the stage to execute on the best available agent which could be an agent where you forgot to install the NUnit executables.
3. The Agent Compatibility Matrix will report agents as compatible when there's a chance they aren't if you didn't install the NUnit executables.