

# TARGETPROCESS PLUGIN DEVELOPMENT GUIDE



v.2.8

Plugin Development Guide

This document describes plugins in TargetProcess and provides some usage examples.

**PLUG-IN DEVELOPMENT ..... 3**

- CORE ABSTRACTIONS ..... 3
- PLUGIN LIFECYCLE ..... 4
- ERROR HANDLING ..... 5

**DEVELOPING PLUGINS ..... 6**

- PLUGIN STRUCTURE ..... 6
- INSTALLING PLUGIN ..... 7
- LOGGING ..... 8
- APPLICATION SERVICES ..... 8
  - Business Logic* ..... 8
  - Events* ..... 10
  - User Interface* ..... 12
  - Plugin Settings* ..... 14

**WALKTHROUGHS ..... 15**

- Write Minimal Plugin Walkthrough* ..... 15
- Write Plugin Which Installs User Interface Commands Walkthrough* ..... 15
- Write Plugin Which Reacts to Persistence and Business Logic Events Walkthrough* ..... 16

**OTHER ..... 18**

# Plug-in Development

Developers can extend TargetProcess, add new functionality to it by means of plugins. Let's consider some use cases for plugins:

- A plugin which imports bugs and issues from a third party bug tracking system. This plugin would run in background periodically polling third party system, downloading new and changed bugs and importing them in TargetProcess.
- A plugin which integrates with source control system. This plugin would run in background periodically polling source control repository to get commits. A developer in the commit message can provide time spent, bug status, etc. Then plugin would parse this information and import it in TargetProcess.

## Core Abstractions

A plugin is a concrete class with default public constructor which implements interface *IPlugin* and optionally provides static metadata in attached attribute *PluginAttribute*. This interface and attribute is declared in namespace *Tp.Integration.Common.Plugins*.

The *IPlugin* interface source code is provided in the following example:

```
using System;
namespace Tp.Integration.Common.Plugins
{
    public interface IPlugin : IDisposable
    {
        ServiceProvider Site { set; }
    }
}
```

The only writable property *Site* is set by TargetProcess upon application startup when it needs to initialize plugin. Once this property is set, plugin can obtain services from the specified services provider, install event listeners, commands and callbacks. The service provider is the only and primary means of communication between application and plugin. Application provides services in different categories, like persistence services, event listener subscription services, user interface command services, etc. These services are documented below.

Attribute *PluginAttribute* can optionally be attached to a plugin class to give it user friendly name and description.

```
using System;
namespace Tp.Integration.Common.Plugins
{
    [AttributeUsage(AttributeTargets.Class, AllowMultiple = false, Inherited = false)]
    public class PluginAttribute : Attribute
    {
        public PluginAttribute(string name) { ... }
        public string Name { get { ... } }
    }
}
```

```

    public string Description { get { ... } set { ... } }
}

```

Plugin name and description is an arbitrary string. If no attribute is attached class name will be the plugin name, and description will be empty. Plugin name is better to be unique; however, it is not a strict requirement.

The minimal plugin implementation is provided in the following example:

```

using System;

[Plugin("My Plugin", Description = "Simple plugin example")]
public class MyPlugin : IPlugin
{
    private IServiceProvider _site;

    public IServiceProvider Site
    {
        set { _site = value; }
    }

    public void Dispose() {}
}

```

This plugin does nothing, but it gets loaded and listed in the **Admin** → **Plugins** section.

To do something useful, plugin has to get some service from the service provider, than call that service.

```

...
public class MyPlugin : IPlugin
{
    ...
    public IServiceProvider Site
    {
        set
        {
            _site = value;
            IApplicationService service =
                (_site.GetService(typeof(IApplicationService)));
            if (service != null)
            {
                service.Operation();
            }
        }
    }
}

```

The provided services constitute API of the application. Every plugin has its own service provider. Some of the provided services are singletons shared between all plugins; some are unique instances for each plugin. Some services are guaranteed to be provided, some not, so you will need to check if the returned instance is not null. Most of the service interfaces are declared in namespace *Tp.Integration.Common*.

## Plugin Lifecycle

The lifecycle of a plugin is as follows:

1. TargetProcess initializes itself.
2. TargetProcess scans *~/Plugins* directory of web application for plugin DLL files.
3. TargetProcess loads found assemblies and instantiates plugin objects in an arbitrary order. If an assembly cannot be loaded, TargetProcess logs warning and continues loading other plugins. If a plugin cannot be instantiated, or plugin throws exception in constructor, TargetProcess logs warning and continues loading other plugins. This is the plugin's last chance to gracefully unload without crashing whole application.

4. For every instantiated plugin, TargetProcess sets *Site*. Plugin gets required services from *Site*, installs event handlers, commands and callback. Now plugin is ready and operational. If plugin throws exception in service provider setter it will crash application.
5. TargetProcess raises events and plugin reacts to them.
6. When TargetProcess is being shut down, it disposes plugins.

## Error Handling

---

TargetProcess handles plugin errors:

1. If plugin DLL cannot be loaded, application logs warning and continues loading other plugins.
2. If plugin object cannot be instantiated or throws exception, application logs warning and continues loading other plugins.
3. If plugin throws exception in *Site* setter, application logs error, the exception pops unhandled, application does not start.
4. If plugin throws exception in a synchronous event handler, application logs error, the exception pops unhandled, request does not complete normally.
5. If plugin throws exception in an asynchronous event handler, the application handles exception and logs error.
6. If plugin throws exception in dispose, application handles exception and logs error.

# Developing Plugins

## Plugin Structure

Plugin is compiled into DLL assembly. Plugin DLL file is an ordinary assembly, it may contain executable code and resources. It does not need to have strong name, you can use any versioning scheme. Plugins assembly may reference arbitrary assemblies, but two references are mandatory: *Tp.BusinessLogic* and *Tp.Integration.Common* from TargetProcess.

Registration information about your assembly, like title, description, company and product can be displayed on the user interface in TargetProcess.

A special attribute *PluginAssemblyAttribute* is used to distinguish plugins assembly from other assemblies. An assembly will be scanned for plugins only if this attribute is present.

Here is an example of *AssemblyInfo.cs* file of a plugins assembly:

```
using System.Reflection;
using Tp.Integration.Common.Plugins;

[assembly: AssemblyTitle("My Plugin")]
[assembly: AssemblyDescription("My Plugin Description")]
[assembly: AssemblyCompany("My Company")]
[assembly: AssemblyProduct("My Product")]
[assembly: AssemblyCopyright("Copyright © 2008 My Company")]

[assembly: AssemblyVersion("1.2.3.4")]
[assembly: AssemblyFileVersion("1.2.3.4")]

//
// Apply this attribute so that TargetProcess will recognize this assembly as plugin.
// Specify plugin category name. Any name will do. Examples are "Bug Tracking",
// "Source Control", etc.
//
[assembly: PluginAssembly("Bug Tracking")]
```

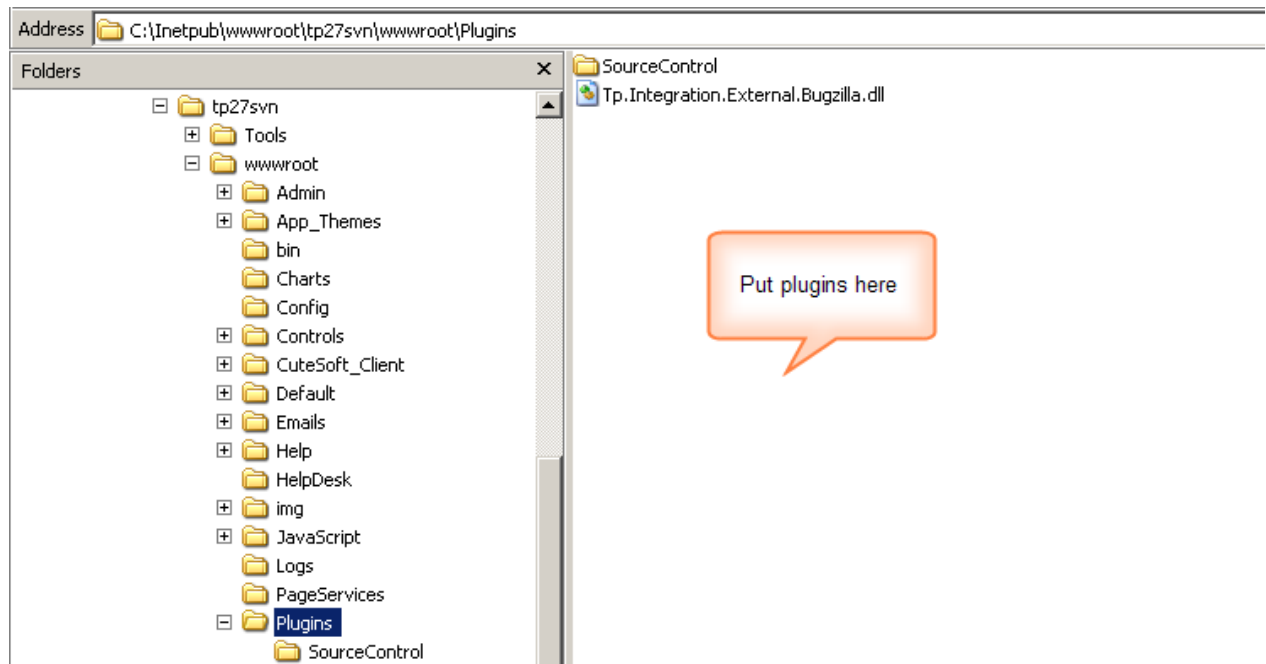
There may be several plugins in the same assembly, but they all will be of the same category. In the example above, the category is *"Bug Tracking"*. Category name groups similar plugins in one place, so it is easier to find them in [Admin](#) -> [Plugins](#) section UI.

We predefined two standard categories, which are *"Bug Tracking"* and *"Source Control"*, but you can use any name in your plugins.

Every plugin is a separate entity; it does not share any state or functionality with other plugins. Plugins are loaded and initialized in an arbitrary order, you must not rely in your plugin on services from other plugins. It is only guaranteed that application will be fully initialized by the time when plugin initializes. The plugin initialization routine is described in details later in this specification.

## Installing Plugin

To install plugin, copy its assembly file and any referenced assemblies to folder `~/Plugins` of TargetProcess web application. You can group plugins in subfolders, like `~/Plugins/SourceControl` or `~/Plugins/Integration`, Target Process will scan plugins folder recursively for plugins.



TargetProcess cannot detect new plugins on the fly, so you will need to restart it to let it read and install your plugins.

When working in development mode, you probably would want to enable shadow copying plugins to a temporary folder to avoid file locks. To do so, set diagnostic switch `shadowCopyPluginFiles` in file `~/web.config` to `true` as in the example below:

```
<?xml version="1.0"?>
<configuration xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
  <system.diagnostics>
    <switches>
      <!-- Specifies whether to copy plugin files to a temporary directory to avoid file locks. -->
      <add name="shadowCopyPluginFiles" value="true"/>
    </switches>
  </system.diagnostics>
  ...
</configuration>
```

If this switch is set, TargetProcess will shadow copy assemblies to a temporary folder prior to loading them, so that you can override original files at any time. There is no need to set this switch in production environment.

## Logging

As plugins often run in background and do not have any user interface there must be a way to notify administrator about any errors happened in plugins and normal plugin activities. Use *Commons.Logging* library in your plugins for logging purposes as in the following example:

```
using System;
using Common.Logging;
using Tp.Integration.Common.Plugins;

namespace Tp.Uni tTests.Extensibi lity
{
    public class MyPlugin : IPlugin
    {
        private readonly ILog log = LogManager.GetLogger("MyPlugin");
        private IServiceProvider site;

        public IServiceProvider Site
        {
            set
            {
                site = value;
                log.Info("Plugin initialized");
            }
        }

        public void Dispose()
        {
            log.Info("Plugin disposed");
        }
    }
}
```

Any logging entries made by plugins will appear in application logs (**Admin** -> **Log**) or ~/Log folder.

## Application Services

You should use application services for all persistence and business operations. Complete API references may be download from TargetProcess web site at <http://www.targetprocess.com/support.asp>.

### Business Logic

*IEntityService* interface provides basic functionality for CRUD operations with entities.

Name	Description
<b>Create</b>	Creates the specified entity.
<b>Delete</b>	Deletes the entity with the specified id.
<b>GetByID</b>	Gets the entity by ID.
<b>GetIDs</b>	Retrieves the IDs of entities by specified HQL.
<b>Retrieve</b>	Retrieves the list if entities by specified HQL.
<b>RetrieveAll</b>	Retrieves all.
<b>RetrieveCount</b>	Retrieves the count of entities.
<b>RetrievePage</b>	Retrieves the page (the range) of entities.
<b>Update</b>	Updates the specified entity.

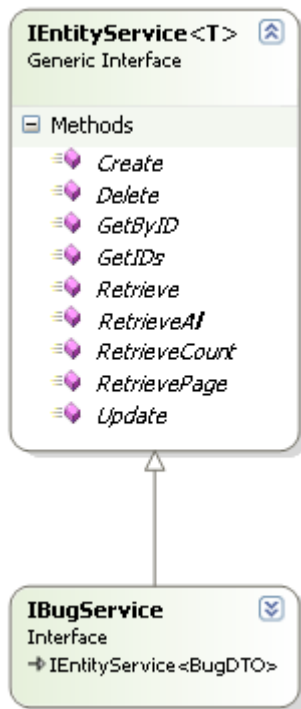
Services usage is quite straightforward. You instantiate service provider, invoke required service and invoke required methods.

In this example we load bug with ID = 5 and change its state to Fixed.

```

bugFacade = (IBugService) _serviceProvider.GetService(typeof(IBugService));
BugDTO dto = _bugFacade.GetByID(5);
dto.EntityStateName = "Fixed";
_bugFacade.Update(dto);
    
```

Each entity has own service with additional methods.



As an example, below you may find all members of *BugService*:

Name	Description
<b>AddAttachmentToBug</b>	Adds the attachment to the specified Bug. Note: The filename is the name of existing file which should be uploaded to upload directory using FileService or manually.
<b>AddBugWithAttachment</b>	Adds the bug with attachment.
<b>AddCommentToBug</b>	Adds Comment to the specified Bug
<b>AddRequestGeneralToBug</b>	Adds Request General to the specified Bug
<b>AddRevisionAssignableToBug</b>	Adds Revision Assignable to the specified Bug
<b>AddTeamToBug</b>	Adds Team to the specified Bug
<b>AssignUser</b>	Assigns the user by id to the specified Bug.
<b>AssignUserAsActor</b>	Assigns the user by id as actor to the specified Bug.
<b>AssignUserByEmailOrLogin</b>	Assigns the user by email or login to the specified Bug.
<b>AssignUserByEmailOrLoginAsActor</b>	Assigns the user by email or login as actor to the specified Bug.
<b>ChangeEffort</b>	Changes the effort of specified Bug entity.
<b>ChangeState</b>	Changes state of Bug entity to the specified state.

<b>Create</b>	Creates the specified entity.
<b>Delete</b>	Deletes the entity with the specified id.
<b>GetByID</b>	Gets the entity by ID.
<b>GetIDs</b>	Retrieves the IDs of entities by specified HQL.
<b>GetPriorities</b>	Loads available priorities for Bug.
<b>GetSeverities</b>	Gets the severities.
<b>RemoveAttachmentFromBug</b>	Removes Attachment from specified Bug.
<b>RemoveCommentFromBug</b>	Removes Comment from specified Bug.
<b>RemoveRequestGeneralFromBug</b>	Removes Request General from specified Bug.
<b>RemoveRevisionAssignableFromBug</b>	Removes Revision Assignable from specified Bug.
<b>RemoveTeamFromBug</b>	Removes Team from specified Bug.
<b>Retrieve</b>	Retrieves the list if entities by specified HQL.
<b>RetrieveActorEffortsForBug</b>	Loads the child collection of Actor Effort entities for specified Bug.
<b>RetrieveAll</b>	Retrieves all.
<b>RetrieveAllForBuild</b>	Loads Bug entities by specified Build.
<b>RetrieveAllForEntityState</b>	Loads Bug entities by specified Entity State.
<b>RetrieveAllForIteration</b>	Loads Bug entities by specified Iteration.
<b>RetrieveAllForLastCommentUser</b>	Loads Bug entities by specified General User.
<b>RetrieveAllForOwner</b>	Loads Bug entities by specified General User.
<b>RetrieveAllForPriority</b>	Loads Bug entities by specified Priority.
<b>RetrieveAllForProject</b>	Loads Bug entities by specified Project.
<b>RetrieveAllForRelease</b>	Loads Bug entities by specified Release.
<b>RetrieveAllForSeverity</b>	Loads Bug entities by specified Severity.
<b>RetrieveAllForUserStory</b>	Loads Bug entities by specified User Story.
<b>RetrieveAttachedRequestsForBug</b>	Loads the child collection of Request General entities for specified Bug.
<b>RetrieveAttachmentsForBug</b>	Loads the child collection of Attachment entities for specified Bug.
<b>RetrieveCommentsForBug</b>	Loads the child collection of Comment entities for specified Bug.
<b>RetrieveCount</b>	Retrieves the count.
<b>RetrieveOpenForMe</b>	Loads open Bug entities for the currently logged in user.
<b>RetrieveOpenForUser</b>	Loads open Bug entities for specified user
<b>RetrievePage</b>	Retrieves the page (the range) of entities.
<b>RetrieveRevisionAssignablesForBug</b>	Loads the child collection of Revision Assignable entities for specified Bug.
<b>RetrieveTeamsForBug</b>	Loads the child collection of Team entities for specified Bug.
<b>Update</b>	Updates the specified entity.

## Events

You may subscribe to events when new entity added into TargetProcess or existing entity updated. *IEntityEventsSource* interface hold these events:

<b>OnEntityPostCreate</b>	Fired when new entity added into TargetProcess
<b>OnEntityPostUpdate</b>	Fired when existing entity updated in TargetProcess

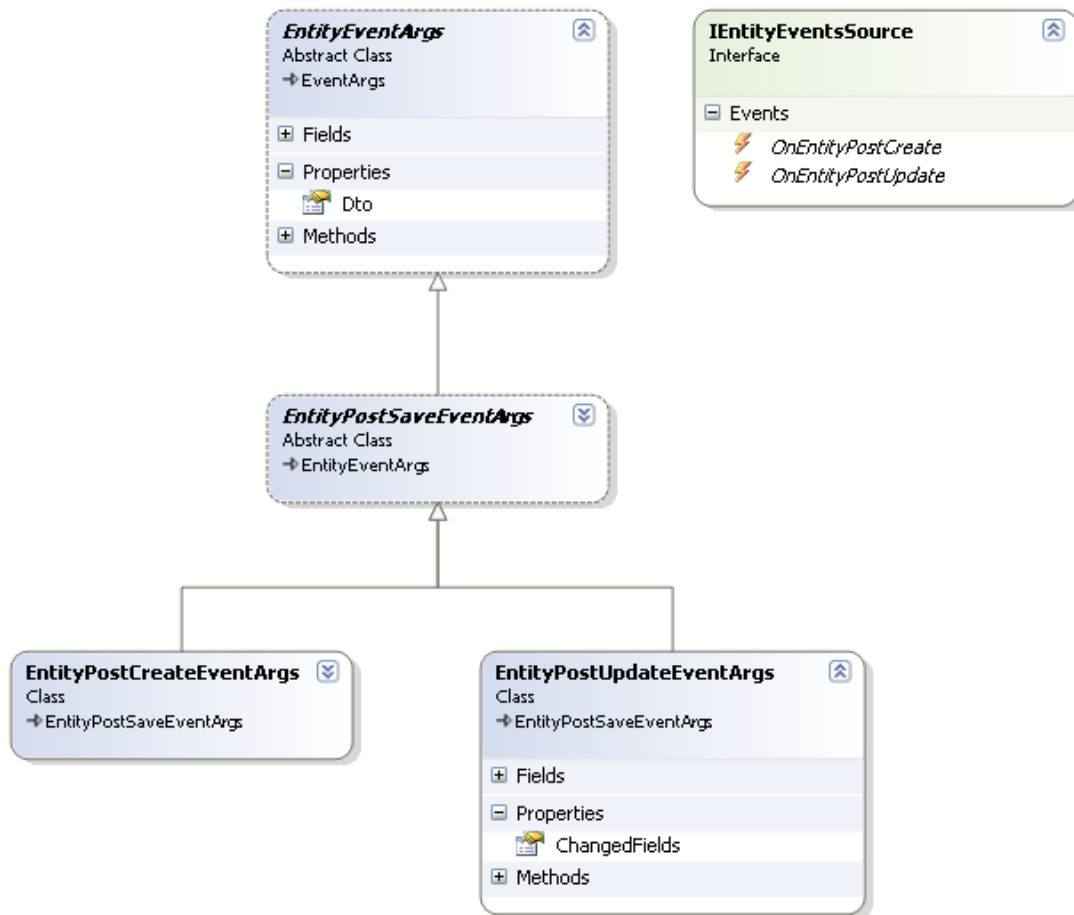
You should *IEntityEventsSource* interface service and subscribe to these events and show in the code below.

```
// Subscribe to persistence events.
IEntityEventsSource eventsSource =
    (IEntityEventsSource) _serviceProvider.GetService(typeof(IEntityEventsSource));
eventsSource.OnEntityPostCreate += OnEntityPostSave;
eventsSource.OnEntityPostUpdate += OnEntityPostSave;
```

*OnEntityPostSave* event handles two events. It extracts *BugDTO* from *EntityEventArgs*.

```
private void OnEntityPostSave(object sender, EntityEventArgs ev)
{
    BugDTO dto = ev.Dto as BugDTO;
    if (dto == null)
    {
        return;
    }
    // do something
}
```

Here is events arguments hierarchy. Note that *EntityPostUpdateEventArgs* has *ChangedFields* property that contains array of all changed fields. Each entity has enum that contains all fields (*BugField*, *UserStoryField*, etc.) These enumerations located in namespace *Tp.Integration.Common*.



You may use *IsFieldChanged* method to decide whether to do something with updated entity or not. For example, if a bug state is not changed you may skip execution.

```

EntityPostUpdateEventArgs ev = (EntityPostUpdateEventArgs) state;
BugDTO dto = ev.Dto as BugDTO;

// Skip processing if state is not changed.
if (!ev.IsFieldChanged(BugField.EntityStateID))
{
    return;
}
    
```

## User Interface

So far TargetProcess does not provide any means for plugins to extend its interface with sophisticated custom components, but plugins can put some buttons on the user interface, and listen for their events. Those buttons called commands; TargetProcess uses abstractions from namespace *System.ComponentModel.Design* to manage them. Refer to classes *CommandID*, *MenuCommand* and *MenuCommandService* from this namespace for more information.

Every command has unique id within plugin instance scope, which consists of menu group GUID and numeric command identifier. It is OK if several plugins have the same command identifiers. So far TargetProcess recognizes only one menu group `{0541E42B-ED30-4b34-B4F7-FE856F7B826E}` which is

GUID of plugin profile commands group. Use this GUID to associate commands with plugin profile, your commands will be available for every plugin profile. TargetProcess will not display commands which menu group GUID does not match the one specified above.

Numeric command identifiers in range 0–99 are reserved by TargetProcess. Your plugin can install commands with identifiers in this range, but these commands will be handled first by TargetProcess itself, then by your plugin.

Installing new command is easy. First, make unique numeric command identifier in the range above 99. Then write command callback method which TargetProcess will invoke if the command button is clicked. Then use service *IMenuCommandService* to install your command in TargetProcess. The following code fragment demonstrates this in details:

```
[Plugin("My Plugin")]
public class MyPlugin : IPlugin
{
    private static readonly CommandID CheckConnectionCommandID =
        new CommandID(new Guid("{0541E42B-ED30-4b34-B4F7-FE856F7B826E}"), 101);

    private MenuCommand checkConnectionCommand =
        new MenuCommand(OnCheckConnection, CheckConnectionCommandID);

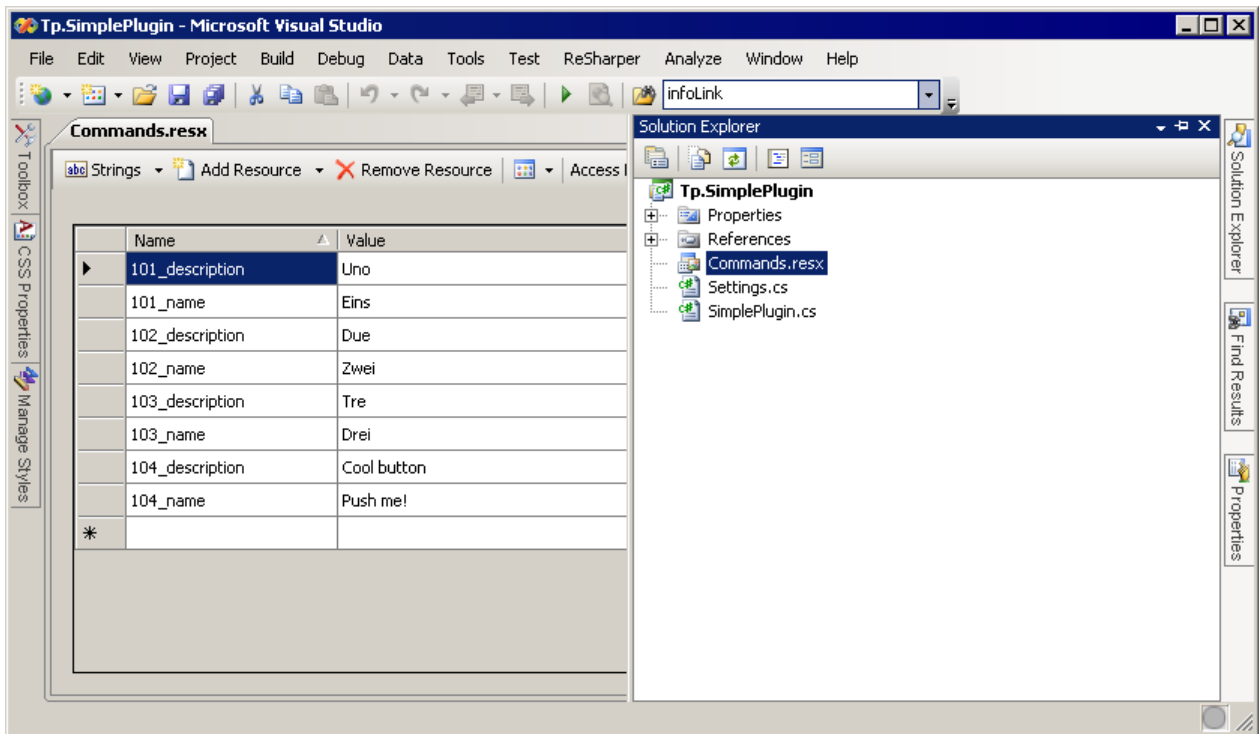
    public IServiceProvider Site
    {
        set
        {
            _serviceProvider = value;
            IMenuCommandService menuCommandService =
                (IMenuCommandService) _serviceProvider.GetService(typeof(IMenuCommandService));
            if (menuCommandService != null)
            {
                checkConnectionCommand.Supported = null;
                checkConnectionCommand.Enabled = null;
                menuCommandService.AddCommand(checkConnectionCommand);
            }
        }
    }

    private void OnCheckConnection(object sender, EventArgs ev)
    {
        // ... handle command button clicked event ...
        // optionally support/unsupported, enable/disable command
        // checkConnectionCommand.Supported = true;
        // checkConnectionCommand.Enabled = true;
    }
}
```

But as you see we did not specified name for the command in the code above. This is because command names and descriptions are provided as string resources. TargetProcess finds command names and descriptions using the following convention. There must be a resources file named *Commands.resx* in the same namespace where your plugin class is declared.

For example, if your plugin class fully qualified name is *Foo.Bar.MyPlugin*, then full path to resources file must be *Foo/Bar/Commands.resx*. In the resources file you must provide collection of strings to give every command name and description. Command name and description string keys follow pattern *{commandID}\_name* and *{commandID}\_description*, where *{commandID}* is command numeric identifier.

For example, *101\_name* and *101\_description* are keys for name and descriptions strings of command id 101.



Your command callbacks will be invoked synchronously, blocking user requests, so write fast and efficient event handlers.

You can throw any exceptions from command event handlers. These exceptions will be logged.

## Plugin Settings

TBD

# Walkthroughs

The topics in this section include several walkthroughs that demonstrates plugin development concepts. The walkthroughs illustrate best practices for performing various plugin development related tasks.

## Write Minimal Plugin Walkthrough

This walkthrough demonstrates steps required to write the simplest possible plugin. The plugin does not do anything, but it gets loaded and listed by the TargetProcess.

1. Create class library project.
2. Reference assembly *Tp.BusinessObjects*, *Tp.Integration.Common*.
3. Create class *MyPlugin*, implement interface *IPlugin* in it.
4. Implement property *Site* and method *Dispose*.
5. Optionally attach attribute *PluginAttribute* to specify custom plugin name and description. The complete source code of plugin might look like:

```
using System;
using System.Collections.Generic;
using System.Text;
using Tp.Integration.Common.Plugins;

namespace Foo.Bar
{
    [Plugin("Foo Bar Plugin", Description = "Foo Bar Plugin Description")]
    public class FooBarPlugin : IPlugin
    {
        public IServiceProvider Site
        {
            set {}
        }
        public void Dispose() {}
    }
}
```

6. Compile assembly.
7. Copy DLL file to folder *~/Plugins* of TargetProcess web application.
8. Restart *TargetProcess*, watch log file to see that your plugin has been loaded and is running. Go to the [Admin](#) → [Plugins](#) section, see you plugin has appeared in the list of installed plugins.

## Write Plugin Which Installs User Interface Commands Walkthrough

This walkthrough demonstrates steps required for plugin to install command button in the application user interface.

1. Create minimal plugin as described in the previous walkthrough.
2. Write command callback method.

```
private void OnCheckConnecti on(object sender, EventArgs ev)
{
    ...
}
```

3. In the *Site* setter, obtain menu command service, install command and its callback. Make sure that the obtained menu command service is not null.

```
public IServiceProvider Site
{
    set
    {
        IServiceProvider serviceProvider = value;
        IMenuCommandService menuCommandService =
            (IMenuCommandService) serviceProvider.GetService(typeof (IMenuCommandService));
        if (menuCommandService != null)
        {
            CommandID checkConnectionCommandID =
                new CommandID(Guid("{0541E42B-ED30-4b34-B4F7-FE856F7B826E}"), 101);
            MenuCommand checkConnectionMenuItem =
                new MenuCommand(OnCheckConnection, checkConnectionCommandID);
            menuCommandService.AddCommand(checkConnectionMenuItem);
        }
    }
}
```

4. Create resources file *Comands.resx* in the same namespace where the plugin class resides. Remove any custom tool from this resources file.
5. Add string resource *101\_name* with value *My Command* in the resource file. Add string resource *101\_description* with value *My Command Description* in the resource file.
6. Compile assembly and copy DLL file as described in the previous walkthrough.
7. Restart *TargetProcess*, watch log file to see that your plugin has been loaded and is running. Go to the **Admin** → **Plugins** section, see you plugin has appeared in the list of installed plugins. In the same section create new plugin profile. See your plugin command has appeared for the created profile.

## Write Plugin Which Reacts to Persistence and Business Logic Events Walkthrough

The plugin below do very useful task, it changes states of all just added or updated bugs to Fixed. Important sections marked bold.

```
using System;
using System.Collections.Generic;
using System.ComponentModel.Design;
using System.Text;
using Tp.Integration.Common;
using Tp.Integration.Common.Events;
using Tp.Integration.Common.Plugins;

namespace Tp.Integration.SimplePlugin
{
    /// <summary>
    /// Simple plugin which changes bug state to fixed.
    /// </summary>
    [Plugin("Bug Free Project", Description = "Just a demo of a simple plugin")]
    public class SimplePlugin : IPlugin
    {
        private IServiceProvider _serviceProvider;

        private IBugService _bugFacade;

        private readonly IPluginMenuCommand _command;

        public SimplePlugin()
        {
            _command = new IPluginMenuCommand(OnCommand,
                new CommandID(Commands.StandardGroups.PluginProfileGroup, 104));
        }

        public IServiceProvider Site
        {
            set
```

```

        {
            _serviceProvider = value;
            Initialize();
        }
    }

    public Type SettingsType
    {
        get { return typeof (Settings); }
    }

    public void Dispose() {}

    private void Initialize()
    {
        _bugFacade = (IBugService) _serviceProvider.GetService(typeof (IBugService));

        // Subscribe to persistence events.
        IEntityEventsSource eventsSource =
            (IEntityEventsSource) _serviceProvider.GetService(typeof (IEntityEventsSource));

        eventsSource.OnEntityPostCreate += OnEntityPostSave;
        eventsSource.OnEntityPostUpdate += OnEntityPostSave;

        IMenuCommandService menuCommandService =
            (IMenuCommandService) _serviceProvider.GetService(typeof (IMenuCommandService));
        if (menuCommandService != null)
            InstallCommands(menuCommandService);
    }

    private void InstallCommands(IMenuCommandService menuCommandService)
    {
        _command.Supported = true;
        menuCommandService.AddCommand(_command);
    }

    private void OnEntityPostSave(object sender, EntityEventArgs ev)
    {
        BugDTO dto = ev.Dto as BugDTO;

        if (dto == null)
            return;

        dto.EntityStateName = "Fixed";
        _bugFacade.Update(dto);
    }

    private void OnCommand(object sender, EventArgs ev)
    {
        PluginCommandEventArgs commandEventArgs = ev as PluginCommandEventArgs;
        if (commandEventArgs != null)
        {
            commandEventArgs.Message = "I am alive!";
        }
    }
}
}

```

## Other

You may find source code with sample plugins and Bugzilla Integration plugin at <http://www.targetprocess.com/support.asp>.

Please, email to [support@targetprocess.com](mailto:support@targetprocess.com) if you have questions about plugins development.

Visit support forum at <http://support.targetprocess.com> or TargetProcess Help Desk at <http://helpdesk.targetprocess.com>

If you have any questions about TargetProcess, just contact us:

Email: [info@targetprocess.com](mailto:info@targetprocess.com)

Web: <http://www.targetprocess.com>