Sitecore Experience Platform 8.2

Sitecore Express Migration 2.0 Plugin Guide

How to create a plugin for migrating a custom module

sitecore®
Own the experience™

Express Migration Plugin

The Express Migration plugin subsystem is designed to help you migrate custom Sitecore modules.

The plugin subsystem intercepts data during migration and adjusts it if necessary.

The plugin is an assembly file that contains a class that implements the IModulePlugin interface. You must store every plugin in the `\Sitecore Express Upgrade Tool\plugins` folder. They are automatically discovered when you run the tool.

Extensibility points

When you run the Express Migration Tool, a plugin plays an active role during:

- Configuration

  The plugin generates a list of the module versions that are supported by the source and target instance of Sitecore. The user uses this list to enable module migration and specify the versions of the modules that they want to migrate.

  The plugin checks if the selected version of a module is installed on the source instance.

  The plugin can request additional parameters from the user, and validate that the values entered are correct.

- Analysis

  The Express Migration Tool invokes the relevant plugin before the Analysis step begins.

  The plugin is invoked when the tool analyzes each item and file, and this lets you specify how each particular entity is analyzed.

  The plugin is invoked after the analysis process is complete.

- Migration

  The plugin is invoked before the migration process begins.

  The plugin is invoked when the tool migrates each item and file. The plugin can then modify every item and file when they are read from the source instance and before they are migrated to the target instance.

  The plugin is invoked after the migration process is complete.

The Plugin API

Every plugin must contain a public class that implements the `IModulePlugin` interface. This class should have a public parameterless constructor.

IModulePlugin interface

This interface defines the main communication contract between the Express Migration Tool and a plugin.

Note
When you create a plugin, you must ensure that it is resistant to sudden errors and interruptions

Properties

Property	Description
`Key`	The system name of the plugin. This name must be unique.
`Title`	The user-friendly name of the plugin that is shown in the UI.

Methods

`AppendModuleSpecificReportData`

Provides the information about the module migration that is displayed in the final migration report.

```
void AppendModuleSpecificReportData(IReportSectionDataAppender dataAppender,
IPluginPersistentState persistentState)
```

`CreateMigrationWorker`

Creates an instance of the `IMigrationModuleWorker` interface.

```
IMigrationModuleWorker CreateMigrationWorker(MigrationContext migrationContext,
IPluginPersistentState persistentState)
```

`CreatePopulationWorker`

Creates an instance of the `IPopulationModuleWorker` interface.

```
IPopulationModuleWorker CreatePopulationWorker(MigrationContext migrationContext,
IPluginPersistentState persistentState)
```

`GetSupportedSourceVersions`

Generates a list of the source module versions that the plugin supports.

```
IReadOnlyCollection<SitecoreVersion> GetSupportedSourceVersions(SitecoreVersion
sourceSitecoreVersion)
```

`GetSupportedTargetVersion`

Generates a list of the target module versions that the plugin supports.

```
IReadOnlyCollection<SitecoreVersion> GetSupportedTargetVersion(SitecoreVersion
targetSitecoreVersion)
```

`IsModuleInstalled`

Checks whether the module is installed on the specified instance.

```
bool IsModuleInstalled(IInstance instance, SitecoreVersion pluginVersion)
```

IPopulationModuleWorker interface

During analysis, the population worker performs all the module-specific operations.

Methods:

`OnAfterPopulation`

This method is called after the analysis process is complete.

```
void OnAfterPopulation(IPopulationStatusTracker statusTracker)
```

`OnBeforePopulation`

This method is called before the analysis process is started.

```
void OnBeforePopulation(IPopulationStatusTracker statusTracker)
```

`OnFilePopulating`

This method is called every time a file is analyzed. A plugin can make changes to the FileOperation object and influence the analysis of the file. The method returns a flag that indicates whether each file should be migrated or not.

```
OperationPopulationResult OnFilePopulating(FileOperation operation)
```

`OnItemPopulating`

This method is called every time an item is analyzed. A plugin can make any changes to the ItemOperation object and influence the analysis of the item. The method returns a flag that indicates whether each item should be migrated or not.

```
OperationPopulationResult OnItemPopulating(ItemOperation operation, [CanBeNull] ItemRow
etalonItemRow, [CanBeNull] ItemRow sourceItemRow)
```

IMigrationModuleWorker interface

The migration worker performs all the module-specific operations about converting and transferring data during the migration step.

Methods

`OnAfterMigration`

This method is called after the migration process is completed.

```
void OnAfterMigration()
```

`OnBeforeMigration`

This method is called before the migration process is started.

```
void OnBeforeMigration()
```

`OnFileMigrating`

This method is called every time a file is migrated. It allows you to modify the file content before it is migrated to the target instance.

```
Stream OnFileMigrating(FileOperation operation, Stream fileStream)
```

`OnItemMigrating`

This method is called every time an item is migrated. It allows you to modify the item content before it is migrated to the target instance.

```
void OnItemMigrating(ItemOperation operation, ItemRecord item)
```

IUIConfigurableModulePlugin interface

The Express Migration Tool lets plugins have their own configuration tab, and asks users to specify plugin-specific parameters.

In order to support this feature, the plugin must implement the `IUIConfigurableModulePlugin` interface instead of the `IModulePlugin` interface.

However, this interface requires that you implement an additional method:

```
IUserDataRequest CreateConfigurationRequest(IPluginPersistentState state, MigrationContext
migrationContext);
```

Parameters:

Parameter	Description
`IPluginPersistentState state`	This is storage for the plugin-specific settings and allows the plugin to save the settings and load them when they are necessary.
`MigrationContext migrationContext`	This is an initialized context based on instance-specific parameters. For example, you have access to the file system, databases, and configuration of the source, target, or comparison instances and can use this parameter if your configuration depends on some instance-specific parameters - for example, for validation.

The method should return an instance of the `IUserDataRequest` interface.

That interface should not be implemented by the plugin; it should use one of the predefined classes instead.

UI data requesting framework overview

In order to simplify and consolidate the process of requesting data from the user, the Express Migration Tool has a built-in framework that allows the requested data to be organized hierarchically, and to specify the behavior for each request.

The framework makes two kinds of requests:

- Group requests

- Simple requests

As result, plugins can return a single group request that contains other requests.

All the components related to this framework are located in the `Sitecore.ExpressMigration.Wizard.Steps.Common.UserDataRequesting.Requests` namespace.

Group requests

Group request	Description
VirtualGroup	An invisible wrapper that combines multiple requests. Its main function is to act as the root for other requests or to control the behavior of a set of requests for example, to hide all the requests .
GroupWithLabel	Lets you combine multiple requests in a GroupBox control and give the group a header.
ToggleGroupWithLabel	Lets you specify whether a group of requests is enabled.

Simple requests

Simple requests	Description
LabelBlock	Lets you render some text as a label. That is not actually a data request.
WarningBlock	This is like a LabelBlock, but the text is highlighted in a different way.
StringRequest, BoolRequest, OptionRequest	Allows you to request data of particular kind from the user. These are rendered as a label and require a particular kind of input for example, a checkbox.

Localization

To support localization, when a text is required an `IRequestText` instance is used. If you want to construct an instance, you must use the factory methods from the `RequestText` class.

The methods are:

Method	Description
`AsIs`	The text is not localized and is returned as is.

Method	Description
Loc	The text is localized with the specified localization key. Optionally you can pass a custom `ILocalizationService` to use specific localization key instead of a global one. This is useful for plugins, because their text strings are typically not present in the tool's localized resources.
LocFormatted	The format is localized using the specified localization key, and then the value is formatted using the `string.Format()` method.

Mixins

To extend the behavior of a UI data request, a mixin approach is used. You can attach a set of custom mixins to each request. Some mixins control the appearance of the request, while others control the behavior for example, validation.

You should use extension methods to attach a mixin to a request.

The most commonly used extensions and underlying mixins are:

Mixin methods	Description
WithValueValidation, WithValidation, WithValidator	These methods attach validation to a request. The values are validated before they are saved and if errors are found, the user is asked to correct them. You can also attach validation to a group request. This lets you validate several values together.
WithBehaviorVisibility, WithBehaviorReadOnly	These methods specify whether the request is visible in the UI. You can specify values directly or you can do so in the `IObservable` method, which the tool detects when there are any change in the values.
OnSave, SaveValueTo	These methods specify callbacks which allow you to export the values entered by the user to some external storage. These methods are called after the values have been validated.

Example

```
public IUserDataRequest GetRequest(dynamic dataStorage)
{
  return new VirtualGroup()
    .WithInner
    (
      new LabelBlock(RequestText.AsIs("Welcome to configuration!")),

      //
      new StringRequest(RequestText.AsIs("Your name"), dataStorage.Name)
        .WithValidator(IsNotEmptyValidator)
        .SaveValueTo((string v) => dataStorage.Name = v),

      //
```

```
            new BoolRequest(RequestText.AsIs("Do magic"), true)
              .WithBehaviorReadOnly(true)
        );
    }
```

This example creates the following elements in the UI:

- A label that says “Welcome to configuration!”
- A text field with the “Your name“ label.

  If the field contains a value, the value is displayed in the UI and the field is validated. If the value is empty, the user is asked to enter a value.

  The input is saved to the “dataStorage.Name” variable.
- A checkbox with the “Do magic“ label. The checkbox is read-only and the user it therefore not allowed to modify it.