Sitecore CMS 6.3 to 6.5 and SIP 3.2

SharePoint Integration Framework API Reference

Tips and techniques for SharePoint Integration Developers

Table of Contents

Chapter 1

API Integration

This chapter contains detailed information for developers who want to create their own custom SharePoint integration functionality.

It includes code samples and reference material to assist developers working with the SharePoint Integration Framework API.

This chapter contains the following sections:

- Developer Pre-requisites and Considerations
- The SharePoint Integration Framework API
- API Reference
- SharePoint Web Services

1.1 Developer Pre-requisites and Considerations

Sitecore developers working with the SharePoint Integration Framework must also have a good working knowledge of SharePoint.

Sitecore

Developers must possess the appropriate level of C# and .NET developer expertise and be comfortable using the Sitecore development environment.

SharePoint

Developers must be able to use SharePoint to create new sites, sub-webs and views and configure security permissions. They should also be familiar with using SharePoint Web services. The SharePoint Integration Framework uses standard Web services to connect to and retrieve lists from the SharePoint database server.

Before working with the SharePoint Integration Framework, developers should consider:

File Size

There is a default 500 MB size limit on files that you can upload to the Sitecore Media Library. Notice that this size is supported for Sharepoint media items, though it can decrease the performance in case of vast amount of big Sharepoint media items. To avoid negative performance issues we recommend you do not use a lot of big Sharepoint items.

See the following setting in the `web.config` file:

```
                    <!--  MEDIA - MAX SIZE IN DATABASE
                     The maximum allowed size of media intended to be stored in a database (binary
blob).
                     This value must be less than the ASP.NET httpRuntime.maxRequestLength
setting.
                     Default value: 500MB
                -->
```

Item Limit

In Sitecore, when you add sub items to an item we recommend that you set an item limit to avoid negative performance issues. You can set this in the SharePoint Integration Wizard. The default limit is 100 items.

SharePoint Views

You can use the SharePoint Integration Framework to display SharePoint views. However, it is not possible by default to display calculated columns from SharePoint views using the sample renderings. Currently the SharePoint Integration Framework does not provide this functionality. To overcome this limitation, you need to recreate this functionality in the custom renderings you create.

1.2 The SharePoint Integration Framework API

The SharePoint Integration Framework is a Visual Studio solution consisting of several different projects. Each project contains a set of classes that enable you to instantiate integration objects to use with the sample controls or the Item Provider. A SharePoint integration object is a .NET class used to connect to and retrieve data from a SharePoint website.

The `Sitecore.Sharepoint.ObjectModel` contains the classes that represent SharePoint objects such as *Server*, *Web and List*. These classes are also in the sample controls and in the `Sitecore.Sharepoint.Data.Providers` project. Developers should use either of these classes to customize the Sharepoint Integration Framework.

All communication between Sitecore and SharePoint uses XML format. In the SharePoint Integration Framework, developers work with objects instead of working directly with the XML data.

Some useful classes in the `Sitecore.Sharepoint.ObjectModel`:

- `SpContext` — handles user authentication and the connection to SharePoint

  `SpUiContext` and `SpDataContext` inherit from `SpContext`.

  `SpUiContext` is used in `Sitecore.Sharepoint.Web`.

  `SpDataContext` is used in `Sitecore.Sharepoint.Data.Providers`.

  They implement two different ways of resolving predefined SharePoint credentials set in the `sharepoint.config` file.
- `Server` — used to point to a specific SharePoint server.
- `Web` — used to point to a specific site.
- `List` — used to point to a specific SharePoint list. There are several methods you can use to manipulate a list.
- `BaseItem` — used to point to specific data contained in a list.
- `ItemCollection` — represents a set of SharePoint items retrieved from the specified SharePoint list using the specified options.

1.3 API Reference

This section contains reference information on the main classes used in the SharePoint Integration Framework. A selection of the most useful classes and methods are included in this document, it is not possible to describe every project, class and method included in the solution.

The SharePoint Integration Framework consists of the following projects:

- `Sitecore.Sharepoint.Common`
- `Sitecore.Sharepoint.Data.Providers`
- `Sitecore.Sharepoint.Data.WebServices`
- `Sitecore.Sharepoint.ObjectModel`
- `Sitecore.Sharepoint.Web`

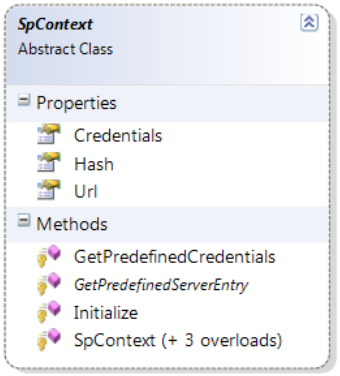
1.3.1 Object Model Classes

Use these classes to customize the sample controls.

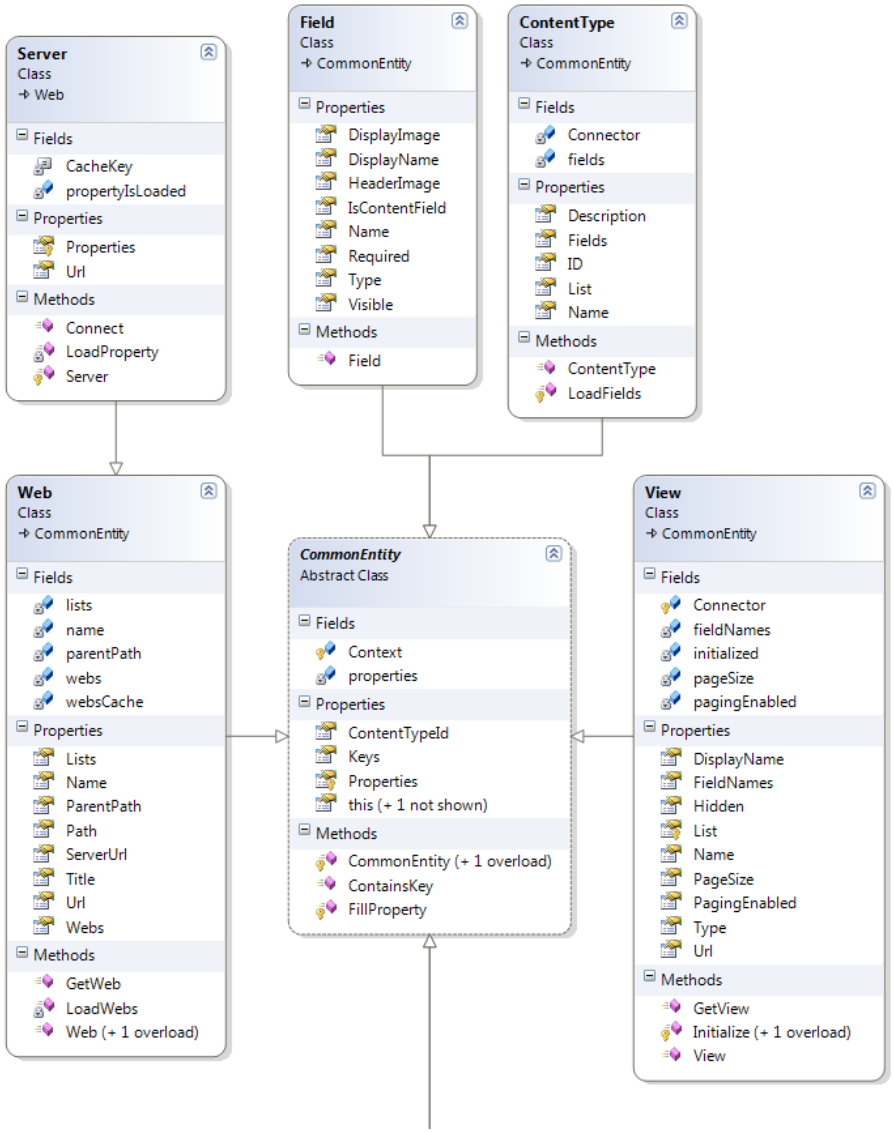
Namespace: `Sitecore.Sharepoint.ObjectModel`

The following table includes a list of the most useful object model classes that developers can use to create their own solution. There is a description of each class and code examples of some of the most useful methods or properties.

Namespace: `Sitecore.Sharepoint.ObjectModel`

Class Name	Description
SpContext 	This is an abstract class used by connector classes. `SpContext` handles SharePoint login credentials in one of three possible ways: • Prompt user for credentials • Use sharepoint.config file • Use logged in user If there are no credentials in the sharepoint.config file, then it uses the credentials of the logged in user by default. It takes both the URL and login credentials to access the SharePoint server.

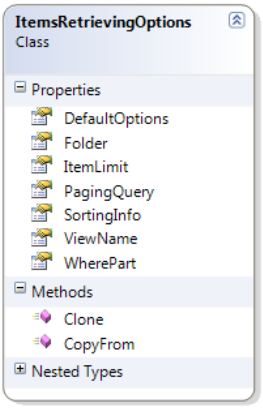
Namespace: `Sitecore.Sharepoint.ObjectModel.Entities`



Class Name	Description
`CommonEntity`	This is the base class for all objects in the Sharepoint Object Model. It encapsulates members that are common to all objects. Members: • `Context` — specifies URL of target SharePoint server and login credentials. • `Properties` — specifies the properties of the current SharePoint object. This is a protected property but you can access it using the indexer.

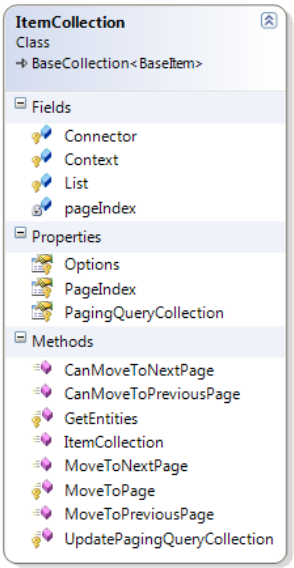
Class Name	Description
`ContentType`	This class represents a Sharepoint content type and uses `ContentTypeID` to show the SharePoint object type. The main purpose of this class is to store a list of fields that an item of this type can contain.
`Field`	Represents a single Sharepoint field for a specific SharePoint content type. Use this class to retrieve properties from fields such as: • Required • Type • Display Name
`Server`	Represents the root web of the SharePoint server. It enables you to perform a search of all webs on the SharePoint server.
`View`	Represents a SharePoint view. If you want to display a specific SharePoint view, use objects of this class to create a list of fields to display. Useful properties include `FieldNames`.
`Web`	This class presents a single SharePoint web. Useful properties include `Lists` and `Webs`. You can use these properties to access child lists or sub webs.

Namespace: `Sitecore.Sharepoint.ObjectModel.Options`

Class Name	Description
`ItemsRetrievingOptions` 	This class represents the settings and options that you can apply to the list items you retrieve from a SharePoint list. Properties: `ViewName` — specifies a SharePoint view for the current SharePoint list. `Folder` — retrieves items from a specific folder for the current SharePoint list. `SortingInfo` — specifies a sort order for the SharePoint list items you retrieve. `WherePart` — specifies a filter (CAML query) that applies to all SharePoint list items retrieved from a specific target destination. `PagingQuery` — specifies which page to retrieve from a SharePoint list item. Applies when the current SharePoint view enables paging for the target SharePoint list.

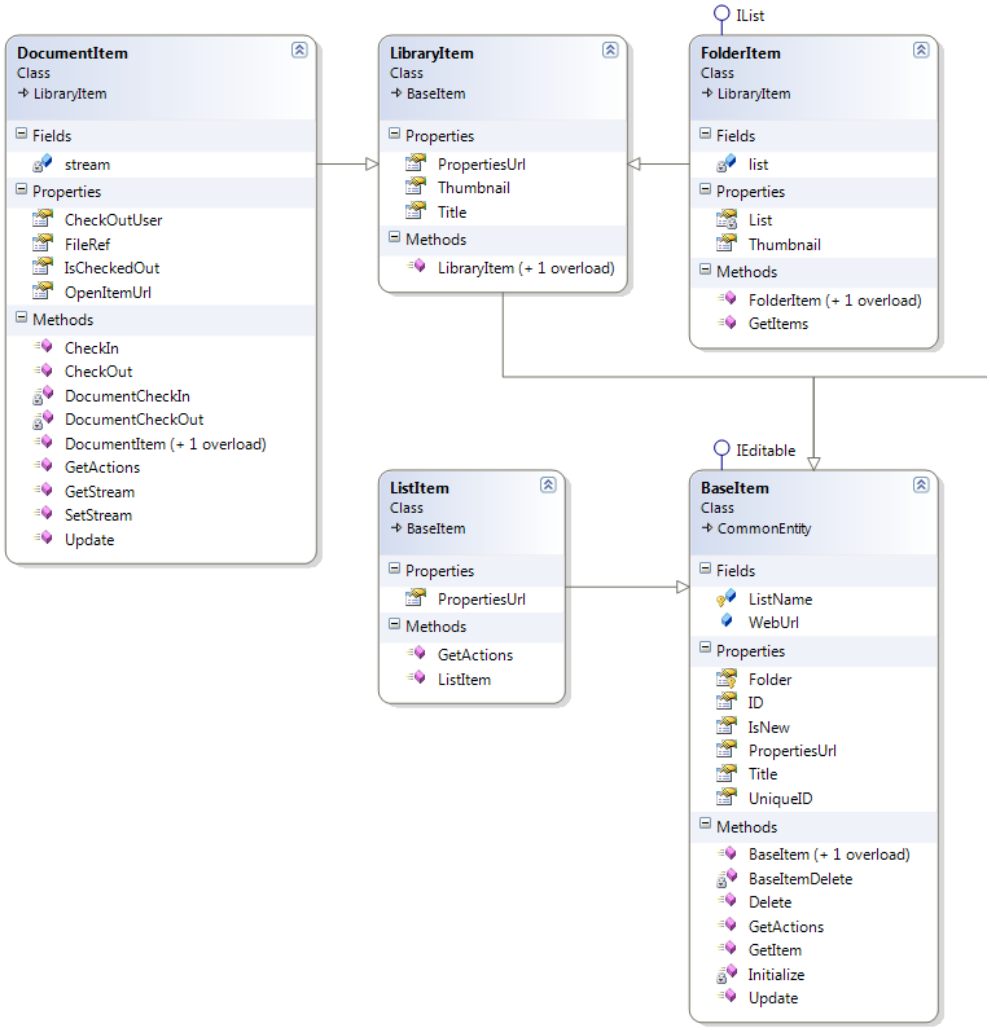
Namespace: Sitecore.Sharepoint.ObjectModel.Entities.Collections

Class Name	Description

ItemCollection
 
 	
	This class represents a set of SharePoint list items. It is necessary to specify a target list and options when you create ItemCollection. After you have created an ItemCollection object it is possible to access all SharePoint list items of a specified SharePoint list filtered using the specified options. Use the indexer of the current ItemCollection to get access to the appropriate SharePoint list items.
	
	Properties:
	PageIndex — specifies the index of a SharePoint list page. This only applies if the SharePoint view used enables paging.
	
	Methods:
	The methods described here also only apply if the specified SharePoint view enables paging. Otherwise each method will return false and the PageIndex property will equal 1.
	
	MoveToNextPage()
	
	MoveToPreviousPage()
	
	CanMoveToNextPage() — verifies that a next page is available.
	
	CanMoveToPreviousPage() — verifies that a previous page is available.
	
	GetEntities() — gets items in a SharePoint list.
	
	UpdatePagingQueryCollection(string newNextPagingQuery)
	
	MoveToPage() — try to move to the next page.
	
	If the next page is available, the PageIndex property will increase. SharePoint list items from the next page of the current SharePoint list (folder) are loaded and the method returns *True*. Otherwise the method will return *False*.
	
	Syntax:
	`protected virtual bool MoveToPage(int pageIndexToRetrieve)`
	`{`
	`Add logic here`
	`}`
	
	Types / **Parameters**
	int / pageIndexToRetrieve
	
	Return value: Bool

Class Name	Description
	Note All the other collection classes represent the sets of the appropriate objects. They do not implement paging. Paging is only available on the item set of a SharePoint list.

Namespace: Sitecore.Sharepoint.ObjectModel.Entities.Items



Class Name	Description
`ListItem`	This class inherits from `BaseItem`. It represents most SharePoint list item types such as announcements or tasks but not SharePoint document libraries types. Syntax: `public ListItem([NotNull] EntityProperties property, [NotNull] string listName, [NotNull] Uri webUrl, [NotNull] SpContext context)`

<table>
<tr><th>Class Name</th><th>Description</th></tr>
<tr><td></td><td>

```
            : base(property, listName, webUrl, context)
        {
    Add logic here
        }
```

Types	Parameters
EntityProperties	property
string	listName
Uri	webUrl
SpContext	context

</td></tr>
<tr><td>LibraryItem</td><td>

Represents SharePoint library list items. This class inherits from BaseItem.

Syntax:

```
    public LibraryItem([NotNull] EntityProperties property,
[NotNull] string listName, [NotNull] Uri webUrl, [NotNull] SpContext
context)
            : base(property, listName, webUrl, context)
        {
    Add logic here
        }
```

Types	Parameters
EntityProperties	property
string	listName
Uri	webUrl
SpContext	context

</td></tr>
<tr><td>FolderItem</td><td>

Represents a SharePoint folder in a SharePoint library that contains multiple list items. This class inherits from LibraryItem and implements the IList interface.

Method:
GetItems(ItemsRetrievingOptions options)- Get SharePoint library items which are located in the current SharePoint folder. The specified options apply.

Syntax:

```
    public ItemCollection GetItems([NotNull]ItemsRetrievingOptions
options)
        {
    Add logic here
        }
```

Types	Parameters
ItemsRetrievingOptions	options

Return value: ItemCollection

</td></tr>
</table>

<table>
<tr><th>Class Name</th><th>Description</th></tr>
<tr><td>DocumentItem</td><td>
Represents SharePoint document list items. This class inherits from LibraryItem. The main feature of these items is that they contain a BLOB as part of the item.

Methods:
CheckIn(string comment) — executes Check In command for the current SharePoint document list item.

Syntax:
<pre>
public bool CheckIn([NotNull] string comment)
{
Add logic here
}
</pre>
<table>
<tr><th>Types</th><th>Parameters</th></tr>
<tr><td>string</td><td>comment</td></tr>
</table>
Return value: bool

CheckOut(bool localCheckout) — executes Check Out for the current SharePoint document list item.

Syntax:
<pre>
public bool CheckOut(bool localCheckout)
{
Add logic here
}
</pre>
<table>
<tr><th>Types</th><th>Parameters</th></tr>
<tr><td>bool</td><td>localCheckout</td></tr>
</table>
Return value: bool

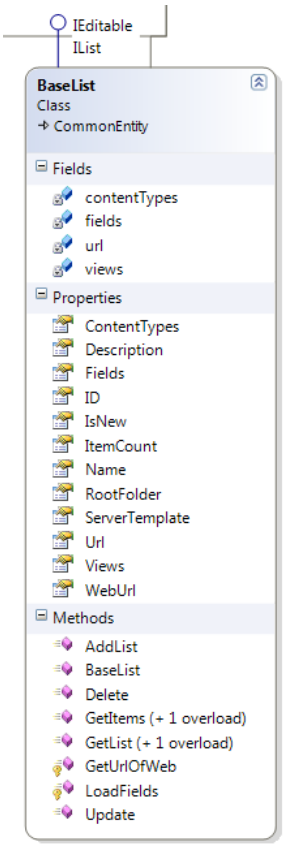
SetStream(Stream streamData) — sets steam of the current SharePoint document item.

Syntax:
<pre>
public void SetStream([NotNull] Stream streamData)
{
Add logic here
}
</pre>
<table>
<tr><th>Types</th><th>Parameters</th></tr>
<tr><td>Stream</td><td>streamData</td></tr>
</table>
Stream GetStream() — gets steam of the current SharePoint document item.

Syntax:
<pre>
public Stream GetStream()
{
Add logic here
}
</pre>
Return Value: Stream
</td></tr>
</table>

Namespace: Sitecore.Sharepoint.ObjectModel.Entities.Lists

Class Name	Description

BaseList
 
 	
	Represents a SharePoint list and list items, views, content types and fields.
	
	Properties:
	• ServerTemplate — represents the type of current SharePoint list.
	• Views — represents all SharePoint views which are available for current the SharePoint list.
	• ContentTypes — represents all types of SharePoint list items that are available for the current SharePoint list.
	• Fields — represents all the fields in a specific list item type which are available for the current SharePoint list.
	
	Method:
	• GetList(string webUrl, string listName, SpContext context) — This is a static method that retrieves a SharePoint list with a specified name from a specific SharePoint web and server.
	
	Syntax:
	`public static BaseList GetList([NotNull] Uri webUrl, [NotNull] string listName, [NotNull] SpContext context)`
	`{`
	`Add logic here`
	`}`
	
	Types / **Parameters**
	Uri — webUrl
	string — listName
	SpContext — context
	
	Return value: BaseList
	
	Other useful methods:
	• GetItems() — retrieves list items from the current SharePoint list. The default SharePoint view is used.
	• GetItems(ItemsRetrievingOptions options) — retrieves list items from the current SharePoint list. Any specific options are used.
	
	All other list objects inherit from the BaseList class and share the same functionality: Sitecore.Sharepoint.ObjectModel.Entities.Lists.BaseList

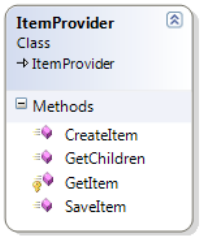
1.3.2 Item Provider Classes

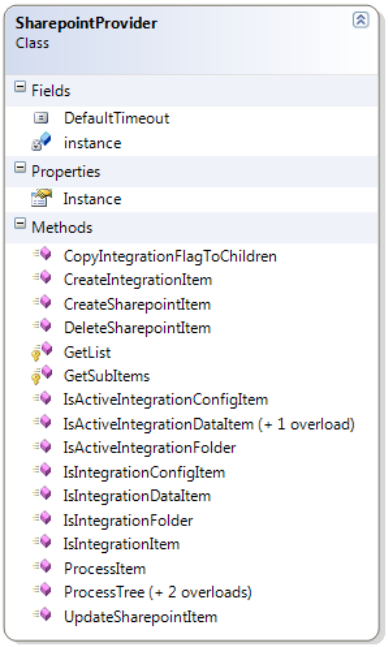
The classes in this section come from the following namespace:

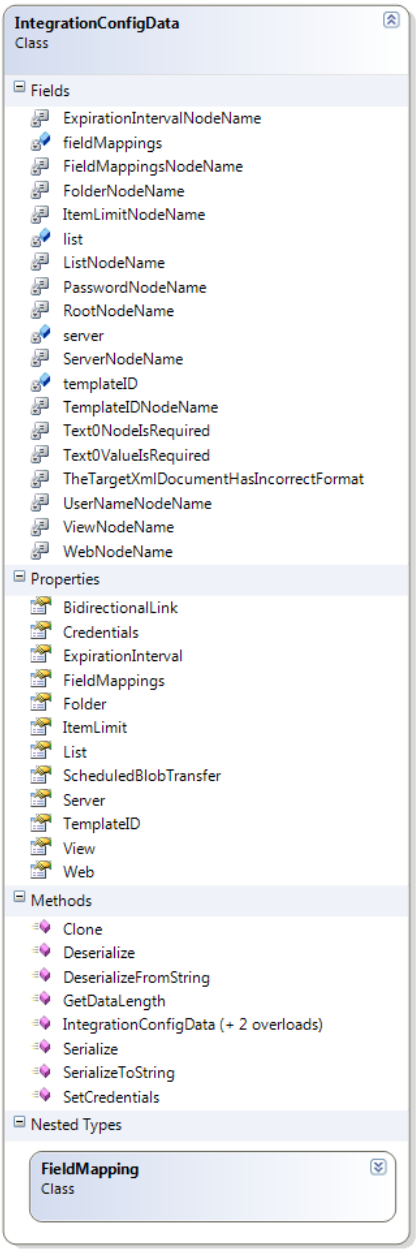
- `Sitecore.Sharepoint.Data.Providers`

Use these classes to customize item level integration.

Diagram showing fields, properties and methods included in these classes.







<table>
<tr><th>Class Name</th><th>Description</th></tr>
<tr><td>ItemProvider</td><td>
This class inherits from the Sitecore.Data.Managers.ItemProvider class.
It calls other methods from the SharepointProvider class to handle Sharepoint specific items:

Sharepoint Integration Definition item
SharepointFolder item

Method: CreateItem
<pre>
           public override Item CreateItem([NotNull] string
itemName, [NotNull] Item destination, [NotNull] ID templateId,
[NotNull] ID newId, SecurityCheck securityCheck)
            {
       Add logic here
            }
</pre>
<table>
<tr><th>Types</th><th>Parameters</th></tr>
<tr><td>string</td><td>itemName</td></tr>
<tr><td>Item</td><td>destination</td></tr>
<tr><td>ID</td><td>templateID</td></tr>
<tr><td>ID</td><td>newId</td></tr>
<tr><td>SecurityCheck</td><td>securityCheck</td></tr>
</table>
Return value: Item
</td></tr>
<tr><td>SharepointProvider</td><td>
This class populates the Sitecore tree with SharePoint data.

Method: ProcessTree
This method passes ProcessIntegrationItemsOptions and Item as parameters. ProcessIntegrationItemsOptions implements how to handle different types of Sharepoint integration items.
It represents the Sharepoint integration definition item or Sharepoint folder item to be processed.
Other important methods used in this class:

ProcessItem
CreateSharepointItem
DeleteSharepointItem

ProcessTree also ensures the synchronization of Sitecore and SharePoint items.

It does not change any items itself but calls the methods of the specified behaviour to make the changes.

Syntax:
<pre>
       public virtual void ProcessTree([NotNull]
ProcessIntegrationItemsOptions processIntegrationItemsOptions,
[NotNull] Item integrationConfigDataSource)
            {
       Add logic here
</pre>
</td></tr>
</table>

<table>
<tr><th>Class Name</th><th>Description</th></tr>
<tr><td></td><td>
<pre>}</pre>
<table>
<tr><th>Types</th><th>Parameters</th></tr>
<tr><td>ProcessIntegrationItemsOptions</td><td>processIntegration ItemsOptions</td></tr>
<tr><td>Item</td><td>integrationConfig DataSource</td></tr>
</table>
</td></tr>
<tr><td>IntegrationConfigData</td><td>This class represents the SharePoint integration configuration data needed to retrieve list items from a SharePoint list. Configuration information is stored in the IntegrationConfigData field of the SharePoint integration definition item. This item uses the Sharepoint Integration Configuration template.

Properties:
Server — specifies the target SharePoint server.

Web — specifies the target SharePoint site.

List — specifies the target SharePoint list.

Folder — Specify a folder fir integration items.

View — specifies which view to use when retrieving list items from the target SharePoint list.

ItemLimit — specifies an item limit. Use this if you want to restrict the number of items that you can integrate. Default limit = 100.

ExpirationInterval — The minimum amount of time between requests to the SharePoint server for updated list information.

TemplateID — ID of Sitecore template assigned to integration items.

FieldMappings — enables you to specify field mappings between SharePoint list items and Sitecore items.

ScheduledBlobTransfer — Schedule downloading of large files at a pre-defined time. For example BLOB files.

BidirectionalLink — Enable updates to items from either SharePoint or Sitecore.</td></tr>
</table>

1.3.3 Connector Classes

In the SharePoint Integration object model, use the following path to locate the connector classes in the code solution:

`Sitecore.Sharepoint.ObjectModel\Connectors`

Entities use connector classes to communicate directly with SharePoint Web services. They transform the XML that SharePoint Web services return to key-value pairs and create CAML queries.

How connectors create CAML queries:

- The `Update` method of the `BaseItem` class updates the current SharePoint list item. To do this it runs the `UpdateItem` method which is part of the `ItemConnector` class.
- It then passes properties or key-value pairs to the current item and the target SharePoint list.
- The `UpdateItem` method of `ItemConnector` class creates a CAML query using values in the properties and the target list.
- It passes these values to the `UpdateListItems` method of `SharepointLists` Web service.

1.3.4 Pipelines

Item level integration hooks into a series of pipelines in the sharepoint.config file to integrate SharePoint lists with Sitecore items and to perform additional actions such as update and delete. Each pipeline consists of a series of processors that execute in a specific order. Pipelines are easy to create and to customize and you can add your own custom pipelines and processors to the sharepoint.config file.

Example pipeline: `createIntegrationItem`

```
<createIntegrationItem>
         <processor type="Sitecore.Sharepoint.Pipelines.CreateIntegrationItem.GetTemplate,
Sitecore.Sharepoint.Data.Providers" />
         <processor type="Sitecore.Sharepoint.Pipelines.CreateIntegrationItem.CreateItem,
Sitecore.Sharepoint.Data.Providers" />
         <processor type="Sitecore.Sharepoint.Pipelines.ProcessIntegrationItem.UpdateFields,
Sitecore.Sharepoint.Data.Providers" />
         <processor type="Sitecore.Sharepoint.Pipelines.ProcessIntegrationItem.UpdateBlob,
Sitecore.Sharepoint.Data.Providers" />
        </createIntegrationItem>
```

This pipeline creates a new integrated Sitecore item. You invoke four processors when you use the SharePoint Integration wizard to map a SharePoint list with Sitecore:

- GetTemplate
- CreateItem
- UpdateFields
- UpdateBlob

Each processor contains a series of steps that to be executed in a specific order. The logic that implements each processor action is contained in a C# class file stored in the Pipelines folder of the SharePoint Integration Framework code solution.

This table describes the purpose of each of the default integration pipelines in the sharepoint.config file.

Pipeline Name	Description
`createIntegrationItem`	Use this pipeline to integrate SharePoint list items with Sitecore. This pipeline creates a new Sitecore item and adds it to the content tree. This pipeline runs when you use the wizard or any time a new item is added to SharePoint. This pipeline has four processors:

Pipeline Name	Description
	• *GetTemplate* — Retrieves the appropriate Sitecore template. • *CreateItem* — Creates a new item in the Sitecore content tree. • *UpdateFields* — Adds fields as defined in mappings in the SharePoint Integration wizard. • *UpdateBlob* — If the item contains a Blob then a Blob field is added to the Sitecore item.
`ProcessIntegrationItem`	This pipeline processes any changes made to a SharePoint list and updates the corresponding integrated Sitecore items. For example, if you edit or create a new list item in SharePoint, this pipeline updates the existing integrated Sitecore item to reflect the changes made in SharePoint. This pipeline also runs when the expiration interval has expired or when Sitecore starts. This pipeline has four processors: • *GetItem* — Retrieves the appropriate Sitecore item. • *IsLocked* — Checks to see if there is a lock on the Sitecore item. • *UpdateFields* — If the item is not locked then it updates the fields in the Sitecore item. • *UpdateBlob* — If the item contains a Blob then it updates the Blob field.
`deleteIntegrationItem`	When you delete list items from SharePoint, this pipeline runs and deletes the corresponding integrated item from the Sitecore content tree. This pipeline as three processors: • *GetItem* — Retrieves the Sitecore item. • *IsLocked* — Checks to see if there is a lock on the Sitecore item. • *DeleteItem* — If the item is not locked it is deleted.
`createSharepointItem`	If you add a new document to the Sitecore Media library, for example, a Word document or an image file, this pipeline runs and creates a new SharePoint list item. The pipeline has two processors: • *IsBidirectionalLink* — This enables you to create the item from either Sitecore or SharePoint. • *CreateItem* — This creates a new item in SharePoint.
`updateSharepointItem`	This pipeline processes any changes made to Sitecore integrated items and updates the corresponding SharePoint

Pipeline Name	Description
	list. For example, if you make any changes to a Sitecore item that has fields mapped to a SharePoint list when you click save, this pipeline runs and updates the corresponding SharePoint fields. This pipeline also runs when the expiration interval has expired or when Sitecore starts. This pipeline has four processors: • *IsBidirectionalLink* — This enables you to change the item from either Sitecore or SharePoint. • *GetItem* — Retrieves the SharePoint list item. • *IsCheckedOut* — Checks to see if the item is checked out. If it is checked out, the pipeline is aborted. • *UpdateItem* — If it is not checked out, it updates the SharePoint list item.
`deleteSharepointItem`	When you delete an integrated item from the Sitecore content tree this pipeline runs and deletes the corresponding list item from SharePoint. This pipeline has four processors: • *IsBidirectionalLink* — This enables you to delete the item from either Sitecore or SharePoint. • *GetItem* — Retrieves the list item from SharePoint. • *IsCheckedOut* — Checks to see if the item is checked out. If it is checked out, the pipeline is aborted. • *DeleteItem* — If the item is not checked out, it is deleted.
`translateSharepointValue`	This pipeline translates incompatible field formats found in a SharePoint list to a format compatible with Sitecore. For example, a SharePoint field that has an incompatible date format. This pipeline has two processors: • *CopySourceValue* — Copies the value in the SharePoint field. • *TranslateDateToIsoDate* — Converts the value in the SharePoint field to the ISO Date format.

Pipeline Name	Description
translateIntegrationValue	This pipeline translates incompatible field formats found in a Sitecore item to a format compatible with SharePoint lists. For example, a Sitecore field with an incompatible date format. This pipeline has one processor: • *CopySourceValue* — Copies the value in the Sitecore item.

Pipeline Arguments

`ProcessIntegrationItemArgs.cs`

This class contains a series of arguments or properties that are passed to the processors contained in the SharePoint Integration pipelines.

When one of the processors in a pipeline is invoked, a C# class such as `GetItems` is called which contains references to arguments in the `ProcessIntegrationItemArgs` class.

These arguments are:

```
public class ProcessIntegrationItemArgs : PipelineArgs
{
  public Item IntegrationItem { get; set; }
  public ID IntegrationItemID { get; set; }
  public ID IntegrationItemTemplateID { get; set; }
  public SharepointBaseItem SourceSharepointItem { get; set; }
  public SynchContext SynchContext { get; set; }
  public ProcessIntegrationItemsOptions Options { get; set; }
  public EventSender EventSender { get; set; }
}
```

Argument Name	Description
IntegrationItem	The name of the Sitecore item integrated with SharePoint.
IntegrationItemID	The ID of the Sitecore item integrated with SharePoint.
IntegrationItemTemplateID	The ID of the template assigned to the Sitecore item integrated with SharePoint.
SourceSharepointItem	The name of the SharePoint list item integrated with Sitecore. Sitecore item fields are mapped to this SharePoint list item.
SynchContext	This enables synchronization of data between mapped fields.
Options	Additional settings on the item, For example, expiration interval, BLOB transfer or item limit.

Custom Processors

You can create your own custom pipeline processors that hook into the pipeline arguments. For example, when you integrate a SharePoint list item with Sitecore you could create a processor with logic for choosing the templates that a Sitecore item is based on. To do this, reference the `IntegrationItemTemplateID` argument in the `ProcessIntegrationItemArgs` class to get all

template IDs. This would enable you to choose a different template rather than the default template normally used for that item.

Each argument enables you to retrieve information about the Sitecore items and SharePoint lists that you want to integrate and can be used in different ways.

1.4 SharePoint Web Services

The SharePoint Integration Framework uses SharePoint 2010 Web services to connect to a SharePoint SQL Server database. The Sitecore SharePoint Integration Object model includes the following classes that enable you to communicate with Microsoft SharePoint Web services.

Note
You can use earlier versions of SharePoint with this module; however you will get the best results if you use SharePoint 2010.

When you request a list from SharePoint, Web services retrieve the appropriate data from your SharePoint SQL database.

Some key Web services used by the SharePoint Integration Framework:

Web Service Name	Description
`SharepointCopy`	Methods for copying files between SharePoint sites.
`SharepointLists`	Methods for working with lists and list data.
`SharepointSearch`	Entry point for Enterprise search.
`SharepointViews`	Methods for working with views of lists.
`SharepointWebs`	Methods for working with sites and sub sites.

Chapter 2

Using the API

This chapter contains use cases and tutorials that show how you can use the SPIF API to solve business problems.

This chapter contains the following sections:

2.1 API Use Cases

This section contains the SPIF API use cases. These use cases are designed to illustrate how you can overcome some particular business challenges.

2.1.1 How to Protect a SharePoint Revision

The Problem and the Expected Behavior

An organization has configured item level integration between Sitecore and SharePoint.

They have configured a considerably long expiration interval, for instance, one hour.

On some occasions, editors update items in SharePoint within the expiration interval. If you edit the corresponding item in Sitecore within the same interval, you don't see the latest revision from SharePoint and Sitecore overwrites the revision on SharePoint with your changes when you save the item in Sitecore.

Sitecore must therefore check whether there is a conflict between the revisions in the Sitecore item and the SharePoint item. If there is a conflict, Sitecore should:

- Create a log entry in the Sitecore log file that indicates that there is a conflict.
- Keep the item in SharePoint intact.

Sitecore's Solution

The following list outlines the main points that we perform in our solution:

- plug into the `updateSharepointItem` pipeline. Sitecore executes this pipeline when it updates SharePoint items and you want to control this process.
- add a custom processor to the `updateSharepointItem` pipeline that aborts the pipeline and adds a log entry if it detects a revision conflict.

  The custom processor uses the `Modified` property of the integrated item. This property contains the time when the integrated item in Sitecore was last updated from SharePoint.

  The custom processor must run before the `UpdateItem` processor.

to solve this task:

1. Create a Visual Studio web application project for the existing SIP solution.
2. In Visual Studio, add a reference to the following assemblies:

```
Sitecore.Kernel
Sitecore.Sharepoint.Common
Sitecore.Sharepoint.Data.Providers
Sitecore.Sharepoint.ObjectModel
```

   For information about how to add a reference, see *section 2.2.1, Adding a Reference to a Sitecore Library in Visual Studio*.
3. In your project, create a code file.
4. In the code file, enter the code from the following *Code Sample* section .
5. Build the project and put the compiled DLL file in the `\bin\` folder of your SIP solution.
6. In the `sharepoint.config` file, in the `updateSharepointItem` pipeline, insert the reference to your custom processor.

Insert the reference to your processor before the `UpdateItem` processor:

```
        <updateSharepointItem>
        ...
                <processor type="SPIF_Customization.DoNotOverrideSharepointRevision,
SPIF_Customization" />
                <processor type="Sitecore.Sharepoint.Pipelines.UpdateSharepointItem.UpdateItem,
Sitecore.Sharepoint.Data.Providers" />
        ...
              </updateSharepointItem>
```

The custom processor is configured.

Code Sample

```
        namespace SPIF_Customization
        {
            using System;
            using Sitecore.Diagnostics;
            using Sitecore.Sharepoint.Pipelines.ProcessSharepointItem;
            using Sitecore;

            /// <summary>
            /// Sitecore must check whether there is a conflict between revisions in the Sitecore
integrated item and the SharePoint item, and if there is a conflict, it should:
            /// * Create a log entry in the Sitecore log file that indicates that there is a
conflict.
            /// * Not overwrite the SharePoint item.
            /// </summary>
            public class DoNotOverrideSharepointRevision
            {
                //The updateSharepointItem pipline requires that the class we create contains a
method called Process.
                //The updateSharepointItem pipline changes SharePoint items, we therefore use the
ProcessSharepointItemArgs type for the args parameter.
                public virtual void Process(ProcessSharepointItemArgs args)
                {
                    //Getting the modification time of the given SharePoint item.
                    string lastModified = args.SharepointItem["ows_Modified"];
                    //Getting the time when the integrated item in Sitecore was last updated from
SharePoint.
                    string updated = args.SourceIntegrationItem["Modified"];
                    //Converting the time to universal format
                    DateTime time1 = DateTime.Parse(lastModified).ToUniversalTime();
                    DateTime time2 = DateTime.Parse(updated);

                    if (time1.CompareTo(time2) == 1)
                    {
                        //If there is a conflict between revisions in the Sitecore integrated
item and the SharePoint item, create a log entry in the Sitecore log file and not change item in
SharePoint.
                        Log.Error(string.Format("SharePoint item {0} and the corresponding
integrated item in Sitecore are in conflict. Wait until the item in Sitecore is updated and then
make your changes.", args.SharepointItem.Title), this);
                        args.AbortPipeline();
                    }
                }
            }
        }
```

2.1.2 How to Prevent New Items from Being Deleted

The Problem and the Expected Behavior

An organization requires:

- That new items that were created in SharePoint within a specified period of time (for example, eight hours ago or less) must not be deleted from SharePoint even if a user deletes the corresponding integration items in Sitecore. This is because an editor must review new items first.
- That when a user deletes an integrated item, Sitecore must check whether this item was created within a given interval and if it was, Sitecore must abort the deletion pipeline and add a message to the log file which explains why the SharePoint item cannot be deleted.

Sitecore's Solution

The following list outlines the main points that we perform in our solution:

- plug into the `deleteSharepointItem` pipeline. Sitecore executes this pipeline when it deletes SharePoint items and you want to control this process.
- In the pipeline, add a custom processor that checks whether the item was created within the given interval. If it was, the custom processor adds a message to the log file and aborts the pipeline.

to solve this problem:

1. Create a Visual Studio web application project for the existing SIP solution.
2. In Visual Studio, add a reference to the following assemblies:

```
Sitecore.Kernel
Sitecore.Sharepoint.Common
Sitecore.Sharepoint.Data.Providers
Sitecore.Sharepoint.ObjectModel
```

   For information about how to add a reference, see *section 2.2.1, Adding a Reference to a Sitecore Library in Visual Studio*.

3. In your project, create a code file. In the code file, enter the code from the following *Code Sample* section.
4. Build the project and put the compiled DLL file in the `\bin\` folder of your SIP solution.
5. In the `\App_Config\Include\` folder, create a configuration file called `interval.config` and put the following code in it:

```
<configuration xmlns:patch="http://www.sitecore.net/xmlconfig/">
  <sitecore>
      <settings>
         <setting name="MyTimeInterval" value="08:00:00"/>
      </settings>
  </sitecore>
</configuration>
```

6. In the `sharepoint.config` file, in the `deleteSharepointItem` pipeline, insert the reference to the custom processor. Insert the reference to the processor before the `DeleteItem` processor:

```
<deleteSharepointItem>
...
        <processor type="SPIF_Customization.KeepNewItems, SPIF_Customization" />
        <processor type="Sitecore.Sharepoint.Pipelines.DeleteSharepointItem.DeleteItem,
Sitecore.Sharepoint.Data.Providers" />
</deleteSharepointItem>
```

The custom processor is configured.

Code Sample

```
namespace SPIF_Customization
```

```
        {
            using System;
            using Sitecore.Diagnostics;
            using Sitecore.Sharepoint.Pipelines.ProcessSharepointItem;

            public class KeepNewItems
            {
                //The deleteSharepointItem pipline requires that the class contains a method
called Process.
                //The deleteSharepointItem pipline changes SharePoint items, we therefore use the
ProcessSharepointItemArgs type for the args parameter.
                public virtual void Process(ProcessSharepointItemArgs args)
                {
                    //Getting the creation time of the given SharePoint item
                    string created = args.SharepointItem["ows_Created"];
                    string now = DateTime.Now.ToString();
                    //Converting the time to universal format
                    DateTime time1 = DateTime.Parse(created).ToUniversalTime();
                    DateTime time2 = DateTime.Parse(now).ToUniversalTime();
                    //Calculating how much time has passed since the given item was created
                    var peroid = time2 - time1;
                    //Getting the interval variable from the configuration file. If there is no
setting called MyTimeInterval, the GetTime SpanSetting method sets the time interval as defined
by the "new TimeSpan" expression.
                    var interval =
Sitecore.Configuration.Settings.GetTimeSpanSetting("MyTimeInterval", new TimeSpan(2,0,0));

                    if (peroid < interval)
                    {
                        //If the item was created within the given interval, add a message to the
log file and abort the pipeline.
                        Log.Info(string.Format("Integration item \"{0}\" was created within this
interval (HH:MM:SS): {1}. It has not been reviewed yet and cannot be deleted!",
args.SharepointItem.Title, interval), this);
                        args.AbortPipeline();
                    }
                }
            }
        }
```

2.1.3 How to Monitor Delete Operations

The Problem and the Expected Behavior

An organization requires:

- That when you delete an item in SharePoint, Sitecore must add an entry to the log file when a corresponding integrated item is deleted in Sitecore.

Sitecore's Solution

The following list outlines the main points that we perform in our solution:

- plug into the `deleteIntegrationItem` pipeline. Sitecore executes this pipeline when it deletes integrated items and you want to control this process.
- In the pipeline, add a custom processor that monitors the delete operations. The custom processor must run before the `DeleteItem` processor.

to solve this problem:

1. Create a Visual Studio web application project for the existing SIP solution.
2. In Visual Studio, add a reference to the following assemblies:

```
        Sitecore.Kernel
```

```
        Sitecore.Sharepoint.Common
        Sitecore.Sharepoint.Data.Providers
        Sitecore.Sharepoint.ObjectModel
```

For information about how to add a reference, see *section 2.2.1, Adding a Reference to a Sitecore Library in Visual Studio*.

3. In your project, create a code file. In the code file, enter the code from the following *Code Sample* section.
4. Build the project and put the compiled DLL file in the `\bin\` folder of your SIP solution.
5. In the `sharepoint.config` file, in the `deleteIntegrationItem` pipeline, insert the reference to the custom processor. Insert the reference to the custom processor before the `DeleteItem` processor:

```
        <deleteIntegrationItem>
        ...
                <processor type="SPIF_Customization.MonitorDeleteOperation, SPIF_Customization"
/>
                <processor type="Sitecore.Sharepoint.Pipelines.DeleteIntegrationItem.DeleteItem,
Sitecore.Sharepoint.Data.Providers" />
             </deleteIntegrationItem>
```

The custom processor is configured.

Code Sample

```
        namespace SPIF_Customization
        {
          using Sitecore.Diagnostics;
          using Sitecore.Sharepoint.Pipelines.ProcessIntegrationItem;

          public class MonitorDeleteOperation
          {
            //The deleteIntegrationItem pipline requires that the class you create contains a
method called Process.
            //The deleteIntegrationItem pipline makes changes to Sitecore items, we therefore use
the ProcessIntegrationItemArgs type for the args parameter.
            public virtual void Process(ProcessIntegrationItemArgs args)
            {
              Log.Info(string.Format("Integration item \"{0}\" is going to be deleted!",
args.IntegrationItem.Paths.FullPath), this);
            }
          }
        }
```

2.1.4 How to Fill in a Field when It is Empty in SharePoint

The Problem and the Expected Behavior

An organization has a SharePoint repository and all items contain the **Author** field.

In Sitecore, some integration items contain the **Author** field and some do not.

The organization wants:

- Sitecore to check whether or not items contain the **Author** field.
  - If the item contains the **Author** field and it is empty, Sitecore must insert the string “An author is not specified”.
  - If the item does not contain the **Author** field, Sitecore must add the field mapping that maps the **Author** field to the current integration definition item.

Sitecore Solution

The following list outlines the main points that we perform in our solution:

- plug into the `updateIntegrationItem` pipeline. since Sitecore executes this pipeline when updating integrated items and we want to check whether those items have the Author field in them. If there is no Author field, then the processor adds this field in the template and adds the mapping between the new field and the corresponding field in SharePoint.
- plug into the `translateSharepointValue` pipeline to check the value of the **Author** field in SharePoint and to change this value in the integrated item, . Sitecore runs this pipeline for every field in a SharePoint item.

to solve this problem:

1. Create a Visual Studio web application project for the existing SIP solution.
2. In Visual Studio, add a reference to the following assemblies:

```
Sitecore.Kernel
Sitecore.Sharepoint.Common
Sitecore.Sharepoint.Data.Providers
Sitecore.Sharepoint.ObjectModel
```

   For information about how to add a reference, see *section 2.2.1, Adding a Reference to a Sitecore Library in Visual Studio*.

3. In your project, create a code file. In the code file, enter the code from the following *Code Sample* section.
4. Build the project and put the compiled DLL file in the `\bin\` folder of your SIP solution.
5. In the `sharepoint.config` file, in the `updateIntegrationItem` pipeline, insert the reference to the custom processor that adds the author mapping. Insert the reference to the custom processor before the `UpdateFields` processor:

```
<updateIntegrationItem>
...
        <processor type="SPIF_Customization.AddAuthorMapping, SPIF_Customization" />
        <processor
type="Sitecore.Sharepoint.Pipelines.ProcessIntegrationItem.UpdateFields,
Sitecore.Sharepoint.Data.Providers" />
        <processor type="Sitecore.Sharepoint.Pipelines.ProcessIntegrationItem.UpdateBlob,
Sitecore.Sharepoint.Data.Providers" />
</updateIntegrationItem>
```

6. In the `sharepoint.config` file, in the `translateSharepointValue` pipeline, insert the reference to the custom processor that fills in the author field when it is empty:

```
<translateSharepointValue>
...
        <processor type="SPIF_Customization.FillAuthorField, SPIF_Customization" />
</translateSharepointValue>
```

the custom processor is configured.

Code Sample

```
namespace SPIF Customization
{
    using Sitecore.Sharepoint.Data.Providers.IntegrationConfig;
    using Sitecore.Sharepoint.Pipelines.ProcessIntegrationItem;
    using Sitecore.Sharepoint.Pipelines.TranslateSharepointValue;

    class AddAuthorMapping
    {
```

```
                //The updateIntegrationItem pipline requires that the class contains a method
called Process.
                //The updateIntegrationItem pipline makes changes to Sitecore items, we therefore
use the ProcessIntegrationItemArgs type for the args parameter.
                public virtual void Process(ProcessIntegrationItemArgs args)
                {
                    if (args.IntegrationItem.Fields["Author"] == null)
                    {
                        args.IntegrationItem.Template.AddField("Author", "SharePoint Data");
                        IntegrationConfigData.FieldMapping fieldMapping = new
IntegrationConfigData.FieldMapping("ows_Author", "Author");
                        args.SynchContext.IntegrationConfigData.FieldMappings.Add(fieldMapping);
                        //In the previous line the method adds the field mapping to the current
contextual configuration. To make the mapping work the next time the pipeline runs, Sitecore must
save the configuration in the integration definition item.

IntegrationConfigDataProvider.SaveToItem(args.SynchContext.IntegrationConfigData,
args.SynchContext.ParentItem);
                    }
                }
            }

            class FillAuthorField
            {
                //The translateSharepointValue pipline requires that the class we create contains
a method called Process.
                //The translateSharepointValue pipline processes SharePoint items, thus we use
the TranslateSharepointValueArgs type for the args parameter.
                public virtual void Process(TranslateSharepointValueArgs args)
                {
                    if (args.SourceFieldName != "ows_Author")
                    {
                        return;
                    }

                    if (string.IsNullOrEmpty(args.SourceSharepointItem["ows_Author"]))
                    {
                        args.TranslatedValue = "Author is not specified.";
                    }

                }
            }
        }
```

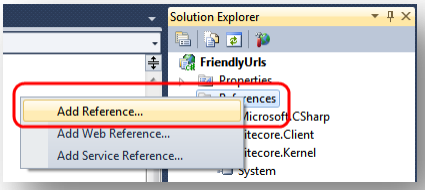
2.2 Tips and Tricks

This section contains some tips and tricks for developers.

2.2.1 Adding a Reference to a Sitecore Library in Visual Studio

To add a reference to a library In Visual Studio:

1. In the **Visual Studio Solution Explorer**, right-click **References**, and then click **Add Reference**.



2. In the **Add Reference** dialog box, select the **Browse** tab.
3. Navigate to the `\bin` folder within the Sitecore solution, for example `C:\inetpub\siotecore\MyWebSite\WebSite\bin` and select the required libraries.

2.2.2 Creating a Visual Studio Web Application Project

For information about creating a Visual Studio web application project for an existing Sitecore solution, see the section *How to Create a Visual Studio Web Application Project* in the following document: http://sdn.sitecore.net/upload/sitecore6/64/presentation_component_cookbook-a4.pdf