Umbraco Courier 2.0

Developer Documentation

Per Ploug Hansen

3/18/2011

Table of Contents

Introduction

This document will show how to work with courier 2 from a developer standpoint and show how to use the courier api. The document will contain step-by-step samples, as well as fully implemented provider samples, plus a full reference section.

Revision History

- Version 1.2 23/5/2011 – Adjusting to RC
  - Adding EventManager
  - ResolutionManager
  - DataResolvers
- Version 1.1 18/3 /2011 – changes in namespaces and class names
  - Adding intro config description
- Version 1, 1/31/2011 – initial api samples

Packaging Manager

Engine for converting Umbraco objects into serialized objects and there-after saved to file on disk. By default the manager generates readable XML

The steps to package a site is:

1. Select the items you want to package, this is either done by knowing the ID and provider GUID, or by querying the available Item providers and selecting from those.
2. Tell the packager where to save
3. Load the items into the engine queue
4. Package all queued items

Get available items from the item providers

Get items from the item providers

Courier uses a provider model to access all data. So to get access to all possible items we can transfer, we need to query the provider model.

```
List<ItemIdentifier> itemsToPackage = new List<ItemIdentifier>();
//get all item providers available
foreach (var provider in Umbraco.Courier.Core.ProviderModel.ItemProviderCollection.Instance.GetProviders()
) {

//get all available system items from the provider
foreach (var item in provider.AvailableSystemItems()) {
  //each item contains meta data (name, description, icon etc), and an ItemID
  //this id is the unique identifier of this specific umbraco object

  itemsToPackage.Add(item.ItemId);

  //a system item can have children, so you can load these into the queue as well
  if (item.HasChildren) {

    foreach (var child in provider.AvailableSystemItems(item.ItemId)) {
      //etc
    }
  }
}
}

//we now have a collection of some items
```

Package the items

To package a set of items, simply tell the engine where to save the files, tell it the id's of the items and then package the queued items.

```
//get the package engine instance
var engine = PackagingManager.Instance;

//Tells the package engine where to package items to
//Foldername = /app_data/courier/revisions/Foldername
engine.Load("foldername");

//clear any existing items queued
engine.ClearQueue();

//Start adding item IDs to package
engine.AddToQueue(new ItemIdentifier("item", ProviderIDCollection.dataTypeItemProviderGuid));
//or
engine.AddToQueue(new ItemIdentifierCollection());

//optional tell the engine to package the items from a remote courier instance
engine.RemotePackagerUrl = "domain.com";
engine.RemotePackagerUser = "admin";
engine.RemotePackagerPassword = "meh";

//Package all items queued
engine.PackageQueue();
```

The items have now been converted into xml files representing the objects you want to move. Courier might have altered the contents of the objects to make it transportable, for instance replacing interger ID's with GUID's – these will be replaced when the items are extracted again.

A typical courier item as XML

When converted, an item is stored as XML, this is how a sample document looks. Notice that the xml does not describe it's property data. That data is handled by the propertyData item provider

```xml
<?xml version="1.0"?>
<Document xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
  <CourierFileName>documents\About_1083.courier</CourierFileName>
  <ItemId>
    <ProviderId>d8e6ad83-e73a-11df-9492-0800200c9a66</ProviderId>
    <Id>150582cc-6e20-4493-8e0a-d3f3b94e25fd</Id>
  </ItemId>
  <Dependencies>
    <Dependency>
      <ItemId>
        <ProviderId>d8e6ad83-e73a-11df-9492-0800200c9a66</ProviderId>
        <Id>626abc7d-6862-4cd9-a249-e62203ef213e</Id>
      </ItemId>
      <Name>Parent Document: Home</Name>
      <IsChild>false</IsChild>
    </Dependency>
  </Dependencies>
  <Resources />
  <Name>About</Name>
  <Id>1083</Id>
  <UniqueId>150582cc-6e20-4493-8e0a-d3f3b94e25fd</UniqueId>
  <ParentId>1082</ParentId>
  <ParentUniqueId>626abc7d-6862-4cd9-a249-e62203ef213e</ParentUniqueId>
  <DocumentTypeAlias>CWS_Textpage</DocumentTypeAlias>
  <Template>CWS_Textpage</Template>
  <Published>true</Published>
  <Properties />
</Document>
```

The above file is stored in:
/app_data/courier/revisions/**{revisionName}**/revision/**{providerfolder}**/**{itemId}**.courier

Extraction Manager

The extraction manager loads xml files, analyses them and places them in a dependency graph. This graph makes sure that the items are extracted to umbraco in the right order.

The steps to extract a collection of items are:

1. Point the extraction engine to a revision location (a folder)
2. Load all revision and resource items in the folder into the engine
3. Build the dependency graph
4. Extract all resources
5. Extract all revision items

The above might seem like tedious but it is indeed also a very complicated problem it tries to solve. The collection of XML files represent 100s of connections between items, and if these items are installed in the wrong order, everything will break.

Extracting items

```
//Get the Extraction Engine Instance
var engine = ExtractionManager.Instance;

//Load a folder of items for ex: c:\inet\wwwroot\app_data\courier\revisionFolder
engine.Load("Folder");

//Build the graph
engine.BuildGraph();

//save the graph as an xml dump
engine.ExtractionGraph.ToXml().Save(System.IO.Path.Combine(path, "graph.xml"));

//save as a mindmap file (blumind)
engine.ExtractionGraph.ToMindmap().Save(System.IO.Path.Combine(path, "mindmap.bmd"));

//if we want feedback while processing it....
engine.ExtractedItem += new EventHandler<ItemEventArgs>(engine_ExtractedItem);
engine.Extracted += new EventHandler<ExtractionEventArgs>(engine_Extracted);

//optional, tell the engine what it can and cannot overwrite
engine.OverwriteExistingDependencies = true;
engine.OverwriteExistingitems = true;
engine.OverwriteExistingResources = true;

//Extract resource files
engine.ExtractResources();

//Extract revision items
engine.ExtractRevisions();
```

Transfer items to a repository

Courier supports setting up repositories where you can pull and push updates to and from. Repositories supports a provider model so any kind of storage can be supported. By default Courier supports using a Courier instance via a webservice as storage or using a folder on a (network)drive.

Notice this uses a slightly different naming, so is subject to change

Courier provides a simplified wrapper around transferring a revision to a repository. So all you need to know is the alias of the repository and the folder name of the revision you want to transfer.

Using the transfer wrapper

```
Umbraco.Courier.Core.Services.Transfering t = new Core.Services.Transfering();
t.CommitRevision( "pathToRevisionFolder" , "RepoAlias");
t.Dispose();
```

This of course does not fit all scenarios, so the below code shows what the wrapper actually covers. Accessing the repository and revisions storage classes to fetch the right objects, loading settings in to the repo provider and finally transferring the item

Using the individual storage classes and repo providers

```
//Get all local revisions available
RevisionStorage rs = new RevisionStorage();
var revisions = rs.GetAllLocalRevisions();
//get a single revision
var revision = rs.GetFromDirectory("MyRevision", "folder" );
rs.Dispose();

//Get possible repositories to send it to
RepositoryStorage repoStore = new RepositoryStorage();
var repositories = repoStore.GetAll();

//a repositories can also be called via it's alias
var repo = repoStore.GetByAlias("live");
repoStore.Dispose();

//We can now tell the repository to transfer the revision through it's provider

//if the provider needs settings we to load these first
repo.Provider.LoadSettings(settings);

//then commit
repo.Provider.CommitRevision(revision);
```

Handle 3rd party datatypes

Because of the many many different ways to store data in umbraco, and different ways to reolve this data, Courier cannot possibly handle every single available datatype. So to remedy this, courier comes with an managed handler to intercept items before they are packaged or extracted, so we're able to register additional files, dependencies and replace values.

Such a mechanism is in Courier called a `ItemDataResolverProvider`, and is available on the namespace **Umbraco.Courier.Core.ItemDataResolverProvider**.

Implement the ItemDataResolverProvider Abstract class. This will give you 2 things to implement:

`List<Type> ResolvableTypes`
List defining what type of items the resolver should consider processing

`Bool ShouldExecute`
Method that can quickly determine if the resolver should run, based on the data in the specifc item being processed.

Besides those 2, there are virtual methods for handling each possible event:

- `Packaging(item)`
- `Packaged(item)`
- `Extracting(item)`
- `Extracted(item)`
- `PostProcessing(item)`
- `PostProcessed(item)`

Sample implementation

First let's define what type we want to resolve data on:

Set the resolvable types

```
public override List<Type> ResolvableTypes {
    get {return new List<Type>(){typeof(Umbraco.Courier.ItemProviders ContentPropertyData)};}
}
```

All build-in types are under Umbraco.Courier.ItemProviders.*

When should we run the resolver?

After that, we need to implement a quick way to determine if the resolver should execute, in this case, I only want to execute the resolver if, one of the properties in the `ContentPropertyData.Data` collection contains a DataType Editor with a specific GUID:

```
//this refers to the Guid of the actual Data Type Editor the Data type is using
private static Guid myDataTypeEditorGuid = new Guid("e0474ca5-e73b-11df-9492-0800200c9a66");

//this is called for the different events on the pakaging and extraction managers,
//if this returns true the event in question will trigger
public override bool ShouldExecute(Core.Item item, Core.Enums.ItemEvent itemEvent) {
var propertyData = (Umbraco.Courier.ItemProviders.ContentPropertyData)item;

//test if any of the properties in the property data uses my custom datatype editor
if (propertyData.Data
            .Where(x => x.DataTypeEditor == myDataTypeEditorGuid)
            .FirstOrDefault() != null)
        return true;

//if not found, return false, no need to execute
    return false;
}
```

Intercept packaging

When the resolver know when and on what to execute, you can start implementing those events you want to intercept, common areas whould be before item is serialized (Packaging event) and before it's installed on a system (Extracting event) so these 2 can be implemented:

```
//this happens before the package is serialized so we can still do some changes to the data
public override void Packaging(Core.Item item) {

 //we cast the item to actual type
 var propertyData = (Umbraco.Courier.ItemProviders.ContentPropertyData)item;

 //we then edit all the properties that has my specific data editor
 foreach (var property in propertyData.Data.Where(x => x.DataTypeEditor == myDataTypeEditorGuid)) {

   var value = property.Value.ToString();

   //we can then do alot of stuff here's some exemples:

   //if it contains a file, we could add that as a resource that gets moved over
   item.Resources.Add("filepathToFile");

   //if it contains a value that references something else (like a document type for exemple), we can
   //add that as a dependency
   //the providerIDCollection contains the keys to all the built-in providers, it's just a guid...
   item.Dependencies.Add("AliasOfDocumentType", ProviderIDCollection.documentTypeItemProviderGuid);

   //or you could simply do a value replace
   property.Value = value.Replace("Per", "Sir Per");
    }
}
```

The thing to notice is that the resolver can directly modify the item, so it can add resources, dependencies, or even modify item data, before it's turned into an xml file.

So if you have any custom data format that courier does not understand, the resolver can pick the important pieces and connect the dots

Intercept Extraction

And then on extracting we can do the same things again to revert any replacements, so if we replaced a key on the packaging side, we can now transform it back into something the other installation can understand

```
//in the other end, we can do similiar stuff when the item is extracted again:
public override void Extracting(Core.Item item) {
//we cast the item to actual type
var propertyData = (Umbraco.Courier.ItemProviders.ContentPropertyData)item;

    //we then edit all the properties that has my specific data editor
foreach (var property in propertyData.Data.Where(x => x.DataTypeEditor == myDataTypeEditorGuid)) {

    //value is an object so you can cast to int / datetime / string depending on property.DBType
    var value = property.Value.ToString();

    //here we convert the string “sir per” back to “per” again
    property.Value = value.Replace("Sir Per", "Per");
}

}
```

How to get to existing data?

As I carefully avoided in the sample, there is not performed any lookup of existing data in the system, which you would normally use one of Umbraco’s apis to do, like the node factory. But as Courier 2 is based on NHibernate and encapsulates all data access inside of a transaction, it is advised to follow that pattern. Luckily it is really easy to Retrieve or persist current data with Couriers `PersistenceManager`.

Translate Node Ids into Unique IDs and back again (good for document handling dependencies):

```
//get a node guid
Guid nodeGuid = PersistenceManager.Default.GetUniqueId(docID);

//get a node ID from a guid
int nodeId = PersistenceManager.Default.GetNodeId(docGUID);
```

Get access to Crud for a specific type,

```
var persister = PersistenceManager.Default.GetCrudForType(typeof(Tag));

persister.PersistItem<Tag>(item);
persister.RetrieveItem<Tag>(itemId);
```

Resolving Property Data

Courier 2 also comes with a provider specific for resolving Document or media properties. It is less complex, but can only be used with the data entered on Documents, nothing else

To use, Simply inherite from `Umbraco.Courier.DataResolvers.PropertyDataResolverProvider` and implement the class. Below is a full example, resolving data from a specific data type, which is XML based, converting page IDs to Guids, and adding pages as dependencies to the item, using the `PropertyDataResolver`.

```
public class SomePropertyPicker : PropertyDataResolverProvider
{
    //this is the unique GUID of the datatype we wish to resolve data from
    //this means that this provider will catch and intercept data from all properties
    //that uses this datatype
    private static readonly Guid TemplatePickerDataTypeId = new Guid("554324D8-5B0D-4BE1-B584-
B169A1F91C56");
    public override Guid DataTypeId
    {
        get { return TemplatePickerDataTypeId; }
    }
    public override void PackagingProperty(Core.Item item, ItemProviders.ContentProperty propertyData)
    {
        if(!string.IsNullOrEmpty(propertyData.Value.ToString()){
            XmlDocument xd = new XmlDocument();
            xd.LoadXml(propertyData.Value.ToString());
            foreach(XmlNode node in xd.SelectNodes("//node")){
                int pageId = int.Parse(node.FirstChild.Value);
                Guid n = PersistenceManager.Default.GetUniqueId(pageId);
                node.Value = n.ToString();

                //adds the document as a dependency, which will include it in the revision
                item.Dependencies.Add(new Dependency(n.ToString(), ItemProviders.ProviderIDCollection.docu
mentItemProviderGuid));
            }
        }
    }
    public override void ExtractingProperty(Core.Item item, ItemProviders.ContentProperty propertyData)
    {
         if(!string.IsNullOrEmpty(propertyData.Value.ToString()){
            XmlDocument xd = new XmlDocument();
            xd.LoadXml(propertyData.Value.ToString());
            foreach(XmlNode node in xd.SelectNodes("//node")){
                Guid pageUniqueId = Guid.Parse(node.FirstChild.Value);

                //page has already been added as it was a depenency so now we just get the new nodeID
                int n = PersistenceManager.Default.GetNodeId(pageUniqueId);
                node.Value = n.ToString();
                }
        }

    }
}
}
```

Working with item Providers

An item provider is what enables Courier to transfer anything. Courier simply providers a set structure of 2 engines. 1 for converting any object into xml, and 1 for converting this object back again, and peristing it.

But in reality, the engines only makes sure that the right provider is called to perform the right action, in the right context. Courier comes with a collection of ItemProviders, but you do not in any way need to use those, you can implement your own and transfer pretty much anything with Courier.

Implementing your own Item provider

An item provider can transfer anything you can describe in a plain .net class. As long as umbraco has access to fetch and save the data associated then. An item provider provides the following parts for transferring an item:

1. **Package**: Package Item based on a unique Key and return it as a .net class, which inherits the Item class
2. **Deserialize**: Deserializes a Courier file into a .net class
3. **Extract**: Unwraps the data in the .net class and saves it as the original item
4. **PostProcess**: Can perform additional tasks on the item after all items have been extracted (like publishing, cleanup, etc)

The main items to be aware of is Package and Extract. Package converts umbraco data into something that can be converted into Courier XML, and Extract takes the Courier XML and saves it as a Umbraco Object. You are however not limited to Umbraco Objects, you can package and extract anything.

Item Provider Sample: Company Resolver

As a separate document, there is a complete sample library available to show how you can resolve custom data, from a custom data type, pointing at a custom database table.

This goes through everything from spotting the data, getting it from the database, to handling packaging and extraction on both ends of a courier deploy.

There are several parts to this, so it is available as a separate document.

Override parts of an Item Provider

If a provider doesn't do a specific thing right for you. You can change the part of it that you do not like. Either by hooking into the Item Provider Event Model, or by overriding a specific part.

To override, simply inherit from the provider, and override the method, the Courier provider engine will take care of instantiating the right provider later.

```
using Umbraco.Courier.Core;
using Umbraco.Courier.Providers.ItemProviders;

namespace Umbraco.Courier.Test
{

public class OverRideFiles : Umbraco.Courier.Providers.ItemProviders.FileItemProvider
{
  public override Item HandlePack(ItemIdentifier id)
  {

    //this returns the result from the original provider
    File file = (File)base.HandlePack(id);

    //alternatively I could return a File object from anywhere..
      File item = new File();
      item.ItemId = id;
      item.FilePath = "something.jpg";

    //here I test if the file exists... again, could replace with whatever
    if (System.IO.File.Exists(Umbraco.Courier.Core.Context.Current.MapPath(item.FilePath)))
    {

      //add the file path as a resource so courier will transfer it as a resource
      item.Resources.Add(item.FilePath);
      return item;
    }

    return null;
  }
}
}
```

References

This part of the Courier documentation lists such things as available providers and their configuration options. File lists and other relevant reference data.

Events

Courier comes with a complete and easy to hook into event model. Easy accessible through Umbraco ApplicationBase integration. All providers comes with a standard set of events as well as the 2 main engines, the extraction and packaging engines.

Item Provider Events

An item provider is what ensures a specific type of object is packaged, compared and extracted. An item event is accessed by connecting to the specific provider you want to subscribe on:

```
using Umbraco.Courier.Providers.ItemProviders;

//Subscripe to the template provider Packaged provider
TemplateItemProvider.Instance().Packaged+=new EventHandler<ItemEventArgs>(Packaged);

//Subscripe to The property item provider
PropertyItemProvider.Instance().Extracting += new EventHandler< ItemEventArgs>(Extracting);
```

Name	Description
Packaging	Fires before packaging of an item
Packaged	Fires after packaging of an item
Extracting	Fires before extracting of an item
Extracted	Fires after extracting of an item
Comparing	Fires before comparison of an item
Compared	Fires after comparison of an item
Deserializing	Fires before deserialization of an item
Deserialized	Fires after deserialization of an item
ComparingResources	Fires before comparing the resources of an item
ComparedResources	Fires after comparing the resources of an item
ComparingDependencies	Fires before comparing the dependencies of an item
ComparedDependencies	Fires after comparing the dependencies of an item
ExtractingResources	Fires before extracting the resources of an item
ExtractedResources	Fires After extracting the resources of an item
ExtractingDependencies	Fires before extracting the dependencies of an item
ExtractedDependencies	Fires after extracting the dependencies of an item
PostProcessing	Fires before postprocessing of an item
PostProcessed	Fires after postprocessing of an item

Extraction Manager Events

The extraction events trigger every time an item is processed in one way or the other via this engine. The can be during Compare, Extract or Post processing.

To subscribe to extraction events, hook into the engine instance.

```
var engine = Umbraco.Courier.Core.ExtractionManager.Instance;
engine.ExtractedItem += new EventHandler<ItemEventArgs>(engine_ExtractedItem);
engine.Extracted += new EventHandler<ExtractionEventArgs>(engine_Extracted);
```

These events trigger each time the engine is running no matter what provider is doing the actual extraction.

The extraction engine supports 2 types of events: Events triggered by a single item, which triggers on each individual item. And events which relates to the entire queue of items currently being processed.

Name	Description
Extracting	Fires before extraction of the entire item queue
Extracted	Fires after extraction of the entire item queue
ExtractingItem	Fires before extraction of a single item
ExtractedItem	Fires after extraction of a single item
PostProcessing	Fires before postprocessing of the entire item queue
PostProcessed	Fires after postprocessing of the entire item queue
PostProcessingItem	Fires before postprocessing of a single item
PostProcessedItem	Fires after postprocessing of a single item

Packaging Manager Events

The packaging events trigger every time an item is processed in one way or the other via this engine. The can be during Compare, Extract or Post processing.

To subscribe to extraction events, hook into the engine instance.

```
var engine = Umbraco.Courier.Core.PackagingManager.Instance;
engine.ExtractedItem += new EventHandler<ItemEventArgs>(engine_ExtractedItem);
engine.Extracted += new EventHandler<ExtractionEventArgs>(engine_Extracted);
```

These events trigger each time the engine is running no matter what provider is doing the actual extraction.

The extraction engine supports 2 types of events: Events triggered by a single item, which triggers on each individual item. And events which relates to the entire queue of items currently being processed.

Name	Description
Extracting	Fires before extraction of the entire item queue
Extracted	Fires after extraction of the entire item queue
ExtractingItem	Fires before extraction of a single item
ExtractedItem	Fires after extraction of a single item
PostProcessing	Fires before postprocessing of the entire item queue
PostProcessed	Fires after postprocessing of the entire item queue
PostProcessingItem	Fires before postprocessing of a single item
PostProcessedItem	Fires after postprocessing of a single item

Available Repository Providers

A repository provider encapsulates the logic needed to connect to remote location. API and configuration wise, these are known as Repository, but in the UI they are always refered to the more common term “locations”.

Current available repository providers

- **Courier Webservice**
  Gives you access to another umbraco site running Courier 2. Enables you to remote extract and package contents
- **Network share**
  Enables you to save revisions to a folder on your local machine or on a network share.
- **Subversion** (experimental)
  Can connect to a subversion repository and Pull revision data into your Umbraco instance, however you cannot send changes back to it currently

Network Share

- **Type**: NetworkShareProvider
- **Guid**: e0472598-e73b-11df-9492-0800200c9a66
- **Full name**: Umbraco.Courier.Providers.RepositoryProviders. NetworkShareProvider

The network share repository can transfer items back and forth to a local or network directory where the asp.net application has access. To add a network share repository, add the following to the courier.config under “repositories”

Configuration XML

```
<repository name="Revisions" alias="revisions" type="NetworkShareProvider" visible="true">
    <path>C:\path\to\repository</path>
</repository>
```

Settings

- **Path**: Contains the fully valid path to the directory where revisions should be stored

Courier Webservice

- **Type**: CourierWebserviceRepositoryProvider
- **Guid**: e0472596-e73b-11df-9492-0800200c9a66
- **Full name**: Umbraco.Courier.Providers.RepositoryProviders. CourierWebserviceRepositoryProvider

The courier webservice provider can connect any other website running umbraco, with courier installed as a repository. It is possible to transfer items back and forth using the http protocol. To install, add the following to your courier.config under "repositories".

Configuration XML

```
<repository name="Live" alias="1" type="CourierWebserviceRepositoryProvider" visible="true">
    <url>http://cws.local</url>
    <user>0</user>

    <login>login</login>
    <password>pass</password>
    <passwordEncoding>Clear|Hashed</passwordEncoding>
</repository>
```

Settings

- **Url**: url to the website where the other instance is accessible
- **User**: The ID of the umbraco user you want to use to authenticate with
- **Login**: (optional) Instead of user ID you can set a specific login name
- **Password**: (optional) Instead of user ID, you can set a specific password
- **PasswordEncoding**: (optional) specify if Courier should keep password clear or Hashed to match your target repository
    - **Note**: Courier alwas encrypts credentials. Encoding is more to do with how Umbraco stores user passwords.

Subversion

- **Type**: SubversionRepository
- **Guid**: e0474ca8-e73b-11df-9492-0800200c9a66
- **Full name**: Umbraco.Courier.SubversionRepository. SubversionRepository

Provider can connect to a subversion repository

Configuration XML

```
<repository name="SVN Repo" alias="svnRepo" type="SubversionRepository" visible="true">
    <url>http://cws.local</url>
    <login>login</login>
    <password>pass</password>
</repository>
```

Settings

- **Url**: url to subversion repository
- **Login**: Your subversion username
- **Password**: Your subversion password