Umbraco Courier 2.0

Sample Item Provider

Per Ploug Hansen

5/24/2011

Table of Contents

Introduction

This document outlines how you can build your own Item provider, along with all the moving parts that goes with it.

So this document will cover how to:

- Write your own data resolver to spot your data in umbraco documents
- Write a custom Item provider to expose your data to courier
- Handle database persistence as part of a courier database transaction

Prerequisites

Courier 2 is built on top of Fluent Nhibernate, so it therefore relies heavily on that. If you have no idea what Fluent Nhibernate is or how to use a ORM, then it is highly recommended to give these pages a read first:

http://wiki.fluentnhibernate.org/Getting_started

http://wiki.fluentnhibernate.org/Auto_mapping

Also, this document is intended for developers. All samples are in C#, and a certain knowledge of C# asp.net, umbraco 4 is expected

Files

This sample will reference a Visual Studio solution, which should be in the same location as you found this document, if not, look for it on umbraco.com under “help and support” for Courier.

Installation

Please run the install.sql against your database to create the companies table, so you have data to work with. Also to get the provider running, Compile and put the CompanyResolver.dll in the /bin

Revision History

- Version 1.2 23/5/2011 – Initial version

Architecture of an Item provider

This sample goes into depth on all the aspects of a full ItemProvider for Courier 2, that means that it handles everything from database calls, spotting dependencies, generating xml for deployment and so on. It's a simple exemple, but at the same time also pretty complex as it covers a lot of different parts.

Hopefully this first chapter will give you an idea of what those parts does so you can tell them apart and use only those things you need.

The problem we want to solve

On my site I have a custom Company picker control, implemented through a Usercontrol wrapper datatype. The data for this data type comes from a custom database table called "companies" containing a list of companies. This is listed in a dropdown, and it includes varies ways to list the company data (xslt/rest extensions etc)

When I deploy the company picked for a page is not included, as it is not standard umbraco data. The reference is transferred, but the actual company object is not, so if I add new ones to my test environment, how do I get those deployed with courier as well?

That is the problem we want to solve, moving custom company data from one instance to another, using courier

The moving parts

So with this introduction, let's start with going through all the different parts and explain what they will do. All parts are named based on the abstract `Umbraco.Courier.Core` class they implement, and the name of the file containing code in the Visual studio files are there as well.

Item name and type	Description
Company:Item Company.cs	The actual piece of data that we want to move around between installations. In this case, the Company class represents a row in the database table "companies" containing the values: id, name, category and symbol. This is the simple piece of data that we wish to move around. To handle the packaging of this data to a deployable format, and converting it back again, we will need an `ItemProvider` at both ends of the transfer
CompanyItemProvider:ItemProvider CompanyItemProvider.cs	The item provider is what can translate our custom data into a neutral data format and back again. It also handles connecting Courier to the database for fetching the raw data, as well as telling Courier what items of this type is available to transfer.
CompanyPersister:ItemCrud CompanyPersister.cs	The connection between Courier and Nhibernate. The item provider asks the persistence manager to either

	retrieve or persist an object of a certain type. The persistence manager finds the provider for that type. This is where the CompanyPersister comes in. It handles querying the “Companies” table for company data.
CompanyProxy:ClassMap CompanyMap.cs & companyproxy.cs	The class that tells Nhibernate how to query the Companies table and how to return the data. CompanyProxy class uses virtual properties and can therefore not be returned directly to courier which cannot serialize virtual properties
CompanyPickerDataResolver:ItemDataResolver CompanyPickerDataResolver.cs	The part of the entire provider that spots when company data Is refenced in actual umbraco documents. It does so by looking at property alias’s specified in the courier.config file

Company

The first thing we’ll do is specify how our data looks. This is done with the company class, already used by the Company picker. So all we have to do is make this class inherit from the core class `Item`

```
public class Company : Item
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Category { get; set; }
        public string Symbol { get; set; }
}
```

This makes it serializable, and enables us to set a ItemIdentifier, the unique key that tells courier what the item is and what provider to use. The rest of Company.cs is really not important for this sample, but it contains some xslt and json helpers as well as the original database logic.

CompanyItemProvider

Next up is the provider which connects the company data to the packaging and extraction managers. So when courier package or extract anything, it is handled through a single manager, which calls all the available providers, who then return the requested data.

First we describe the provider with name, description, icon and a unique GUID:

```
public class CompanyItemProvider : ItemProvider
{
    public CompanyItemProvider()
    {
        this.Id = Constants.companyProviderID;
        this.Name = "Company provider";
        this.Description = "Synchronizes items in the companies table";
        this.ExtractionDirectory = "Companies";
        this.ProviderIcon = Constants.providerIcon;
    }
}
```

After this, we will implement 4 methods:

```
public override List<SystemItem> AvailableSystemItems()

public override Item HandleDeserialize(ItemIdentifier id, byte[] byteArray)

public override Item HandlePack(ItemIdentifier id)

public override Item HandleExtract(Item item)
```

These are what makes the provider work:

- **AvailableSystemItems**: returns all available items the provider can find. So in our case it returns all rows in the companies db table
- **HandleDeserialize**: Converts the Company object into xml
- **HandlePack**: Returns a company object when the manager asks for it.
- **HandleExtract**: Installs / deploys a company object when a manager sends it to it.

Deserializer

To deserialize, simply call the built-in deserializer, which just needs to know what class it is handling:

```
public override Item HandleDeserialize(ItemIdentifier id, byte[] byteArray)
{
    return Umbraco.Courier.Core.Serialization.Serializer.Deserialize<Company>(byteArray);
}
```

Provider calls to data

What is important to see is that in our case the provider will just act as a hub, because all the database work happens in another place, so the provider simply just do calls like so:

```
public override Item HandlePack(ItemIdentifier id)
{
    Company item = this.DatabasePersistence.RetrieveItem<Company>(id);
    return item;
}
```

This leads us to the next element which is how to ask the database for this data

CompanyPersister

The companyPersister class implements the abstract class `ItemCrud` and has a attribute called `ItemCrud` as well.

```
[ItemCrud(typeof(Company), typeof(NHibernateProvider))]
public class CompanyPersister : ItemCrud
```

The abstract class implements that this class can retrieve and persist `company` objects, as well as return a collection of available company objects.

The attribute `ItemCrud(typeOf(company)` indentifies this `ItemCrud` implementation to return `Company` objects, and it does it for the default provider which is the `NhibernateProvider`

So when the ItemProvider calls `DatabasePersistence.RetrieveItem<Company>` this call is routed whatever ItemCrud class that has `Company` in its attribute.

So lets look at persist and retrieve:

In the code it shows code that can handle different kinds of Ids, I've cut that out here for simplicity, but code comments explains the concept

Retrieve Company

Extremely important to notice here is that we get a session from the NhibernateProvider, we then ask the provider for an item of the type CompanyProxy. Which then is used to populate a Company object, which is then returned. A proxy class has to used, as Nhibernate expects all properties to be virtual, and Courier expects them not to be, as it has to Serialize the object.

So the solution is to write a proxy class that returns the data we want and then populate the actual Courier object in the ItemCrud class

Notice how at the end we convert it to the Company object and sets the ItemID, the unique key each Courier item must have.

```
public override T RetrieveItem<T>(ItemIdentifier itemId)
        {
            var session = NHibernateProvider.GetCurrentSession();
            string companySymbol = itemId.Id;

            //the item proxy returned from nhibernate
            CompanyProxy retval = GetCompanyBySymbol(companySymbol, session);
            if (retval == null)
                return null;

            //cast as T
           Company c = ParseFromProxy(retval);
           c.ItemId = itemId;

           return c as T;

        }

private CompanyProxy GetCompanyBySymbol(string symbol, ISession session)
        {
            var c = session.Linq<CompanyProxy>().Where(x => x.Symbol == symbol).FirstOrDefault();
            return c;
        }
```

Getting available Companies

Getting available items from the database, is also where we set the ItemID for the first time, this is where we define the unique identifier for this kind of object.

```
public override List<SystemItem> AvailableItems<T>(ItemIdentifier itemId)
{
    List<SystemItem> allItems = new List<SystemItem>();
    var session = NHibernateProvider.GetCurrentSession();
    foreach (var c in session.Linq<CompanyProxy>())
    {
        SystemItem si = new SystemItem();
        si.Description = "Company with the name: " + c.Name;
        si.Name = c.Name;
        si.ItemId = new ItemIdentifier(c.Symbol, Constants.companyProviderID);
        si.Icon = "package2.png";
        si.HasChildren = false;
        allItems.Add(si);
    }

    return allItems;
}
```

It's a pretty simple operation goin on here, we get all available items with session.Linq<CompanyProxy> and then for each of these create a SystemItem object which describes the available item with as little data as possible. Notice that the ItemId is set to a a ItemIdentifier which consists of the company Symbol and the Company Item Provider ID

Persisting Company

To persist, we do the same thing, get a session, and then tell the session to save a CompanyProxy object.

```
public override T PersistItem<T>(T item)
{
    var session = NHibernateProvider.GetCurrentSession();

    Company update = item as Company;
    CompanyProxy current = GetCompanyBySymbol(update.Symbol, session);

    //if it doesn exist
    if (current == null)
        session.Save( ParseToProxy(update) );
    else
    {
        //set the ID to the current one and persist in case company had its other values updated
        update.Id = current.Id;
        session.Update(ParseToProxy(update));
    }
    return update as T;
}
```

So this leads on the question, where does the `CompanyProxy` object go when it's sent to `session.Save()` or retrieved with `session.Linq<CompanyProxy>()` ?

It goes to a fluent Nhibernate mapping.

Company ClassMap

This is the final step of the road that data travels in Courier, the Fluent Nhibernate mapping which actually dictates how data in the Company object is saved/retrieved from the Companies table.

Luckily, a fluent Nhibernate mapping is really really simply:

In this case it specifies the database table used, and it then maps each property on the CompanyProxy object to the different columns in the table. So CompanyProxy.Name maps to the name Column and so on.

```
public class CompanyMap : ClassMap<CompanyProxy>
{
    public CompanyMap()
    {
        Table("[dbo].[companies]");
        OptimisticLock.None();
        LazyLoad();

        Id(x => x.Id).GeneratedBy.Identity();
        Map(x => x.Name);
        Map(x => x.Category);
        Map(x => x.Symbol);
    }
}
```

If your table is not so nicely named, you can also specify a coloumn name directly like so:

```
Map(x => x.Name).Column("companyName");
Map(x => x.Category).Column("weirdName");
```

More about Fluent Nhibernate mappings can be found here:

http://wiki.fluentnhibernate.org/Auto_mapping

Data abstractions wrap-up

So now that we have all the parts described which handles the flow of data, let's just summarize:

`Company` is the data we move around, inheriting from `Item`, which is what gives each piece of data a unique ID.

Company `ItemProvider` is what exposes and consumes this data when Courier extracts or packages anything

The company `ItemProvider` asks the Company implementation of a `ItemCrud` class for the actual data

…And finally the `ItemCrud` implementation asks `Nhibernate` to return data based on a `Company ClassMap` which finally maps to the Companies database table.

Phew! Loads of layers, but all of them are needed to have this provider based approach. Hopefully we can in the future simplify a lot of these parts

Company Picker Data Resolver

So how we have all this data, and all these providers setup. So we can now open the Courier application in Umbraco and transfer them by selecting them manually.

But, the whole idea of Courier is also that it is able to spot a dependency automaticly and include the needed data.

So for this we need a Custom **ItemDataResolverProvider**. This resolver helps Courier spot and resolve data that is needed by other items. So for instance we can use to spot if a document depends on Company data.

So this implementation has a Company Picker which saves the Symbol of the company on the document. It is implemented with the UserControlWrapper.

Deciding what type of objects it can resolve

When the ItemDataResolverProvider class is implemented, you will need to tell it what type of objects it can resolve, in out case we set it to ContentPropertyData, which is the object used to transport the actual property data of a document

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

After this, we need a fast way to tell if an event should trigger the resolver. This is added to keep things fast, so all providers are not triggered all the time:

So here we test if the property data contains properties with the alias “company” or “pickerfield”

```
public override bool ShouldExecute(Core.Item item,  Core.Enums.ItemEvent itemEvent)
{
    string[] propertyAliass = "company,pickerField".Split(',');

    if(item.GetType() == typeof(ContentPropertyData) && itemEvent ==  Core.Enums.ItemEvent.Packaging)
    {
        ContentPropertyData cpd = (ContentPropertyData)item;
        return (cpd.Data.Where(x => propertyAliass.Contains(x.Alias)).Count() > 0);
    }
    return false;
}
```

If true, the resolver will execute its resolving logic.

Finally, we just need to specify what to do, if there is company data in our document properties. In this case we simply add the Company as a dependency.

```
public override void Packaging(Item item)
{
    ContentPropertyData cpd = (ContentPropertyData)item;

    foreach(var prop in cpd.Data.Where(x => companyProperties.Contains(x.Alias))){
        if (prop.Value != null)
            item.Dependencies.Add(new Dependency(prop.Value.ToString(), Constants.companyProviderID));
    }

}
```

Item.Dependencies contains all the other items which should be installed **before** this item. So by adding a Company item as a dependency to the ContentPropertyData item, we tell Courier that this property data, cannot be installed, untell the Company item has been installed.

Thereby ensuring that Company objects used on the site, is automaticly included in a deploy, and that picked Company data is in the database before the document is created.

And that is it, we have now build an entire model for handling our custom data. Again, this document just shows snippets. Browse and compile the VS solution file to get even more details on this implementation.