CMF Types

CMFTypes

ZOPE and CMF promise to ease the creation and managment of content objects. These content objects usually inherit behavior and framework instances from base classes such as `SimpleItem.SimpleItem` in ZOPE or `CMFCore.PortalContent` in CMF. CMFTypes provides a very high level class, `CMFTypes.BaseContent` , that allows your content objects to participate in the CMFTypes framework.

In the most simple cases you can inherit from `CMFTypes.BaseContent` and define the meta_type and portal_type for the content object as well as the Type Descriptor. The Type Descriptor (`CMFTypes.Field.FieldList`) is a container of Field (`CMFTypes.Field.*`) elements. CMFTypes comes with a default set of Field elements, but you can easily add your own. These Field elements usually contain FormInfo (`CMFTypes.Form.*`) objects that provide extra information and occassionally behavior to a Field element.

Field objects contain a FormInfo object that provides more information about the actual processing of the content in a field. Fields (sort of) enforce the schema of the object as well as guard the object attributes. Where as the FormInfo object is more concerned with what specifically is the form element, PasswordInfo, TextAreaInfo or DateTimeInfo. What really show cases off the power of the FormInfo is the RichFormInfo object which can be configured through the Field object to allow it to accept various binary data formats and process them.

A FormInfo object will usually contain the element label and tooltip (description) that will appear on a rendered form. Whereas the Field object will be used to define the attribute on the object, its restrictions (mode can be used for read/write, read only or more), and whether or not it should participate in the SearchableText behavior, which allows automatic full text indexing/searching of a Field's content. The BaseContent object is simply your content object's definition. It is no not much different from the PortalContent or SimpleItem in CMF or ZOPE -- except it provides even higher level services.

- attributes that you define in FieldList become BaseUnit instances on your object

Content objects often have properties associated with it that are most likely shared across many other content objects. The Dublin Core is an example of Metadata that comes stock with CMF objects. CMFTypes allows you to create your own set(s) of Metadata properties that you can use on your content objects. When creating generic Metadata classes you can ensure re-use across PROJECTS.

- ExtensibileMetadata is the mixin class. All metadata appears to be contained on the `md` attribute, which is a ExtensibleMetadata.MDDict
- How do you add your own metadata? it appears you can define metadatatype = BaseContent.metadatatype + FieldList()
- How do you get your vocabulary from a method?

CMFTypes also provides a much needed piece of infrastructure to the CMF system.

The Content Tool

The ContentTool handles the registration/unregistration of content objects from the CMFTypes and potentially your application. It also provide unique id generation, simple mechanisms to iterate over your registered content objects and even a mechanism to register a callback to be applied to registered objects. The CMFTypes framework auto- registers your objects with the content tool on content instantiation.

Before we dive into an explanation of the internals of the system lets go straight to coding and create a very simple content object.

One of the huge benefits to CMFTypes brings to the table is the static generation of forms. In the Plone CMS CMFTypes is capable of generating the validation scripts, mutator scripts and content views (including both view and edit_form). So we will start off recapping how the framework decides what is going to be generated and what classes you may use in defining the "Type Descriptor".

By default there are 2 "Type Descriptors", type and metadatatype. Both of which are static attributes of the classes. You must subclass BaseContent and if you want to customize the *Properties* tab (metadata) you will want to create a XXXMetadata class that specializes ExtensibleMetadata and then customize the *metadatatype* descriptor.

CMFTypes.Field.FieldList is a sequence of Field objects.

The main building block is CMFTypes.Field.Field which all other higher level XXXField objects specialize. You pass into the constructor the following variables:

CMFTypes.Field.Field

- **id** - [`required string`] this will be the id of the field and attribute on the generated class object. This is the only required field.
- required - [1 or 0] which will require the user to enter a value for this field on a form. It is used for the validation machinery.
- default - ['string'] this will be the default value for the Field value form.
- vocabulary - [(sequence,)] passing in a list of possible values can enforce certain values being entered into a form. The easiest way to use this field is by using the helper class CMFTypes.utils.DisplayList() which takes a tuple of tuples. ((`key` , `value`), (`key1` , `value2`))
- enforceVocabulary [1 or 0] determines whether or not the validation machinery will enforce the value for the Field to be in the set of vocabulary values.
- multiValued [1 or 0] I dont quite understand this.
- searchable [1 or 0] defining this will include this fields value in the SearchableText method. SearchableText() is a method defined on classes that is used in search queries.
- default_content_type [`string`] some Fields can have rich content types. You can specify the default_content_type
- allowable_content_types [(`sequence` , `strings`)] given a list of allowable content types the machinery will make sure that if a content object is uploaded into this Field that it has to be one of these types.
- accessor ["string method name" or None] given a string that is a method name of the class it will use this attribute to look up the value for this Field.
- mutator ["string method name" or None] given a string for a attribute name on teh class it will use this attribute when setting the value for this Field.
- editable [`string method name` or None] ?
- mode ['"string rwe"'] can restrict what methods are callable. r = accessor method, w = mutator method, e = editable method.

- force [`string rwe`] forces the generation of a method. r = accessor method, w = mutator, e = editable methods will be regenerated on the class instance.

- read_premission [`permission string` i.e. CMFCorePermissions.View] The user must be able to satisfy this permission to access the `accesor`

- write_permission [`permission string` i.e. CMFCorePermissions.ModifyPortalContent] The user must be able to satisfy this permission to apply the `mutator`

- form_info [CMFTypes.Form.FormInfo instance] this instance is a decorator of the Field and contains the implementation of the Form widget. For instance if you have a simple Field (which is very generic) and you want to decorate it with a PasswordFormInfo so that it becomes a input type="password".

  All of the below XXXField classes derive from the base Field class mentioned above. So they have usually have the same construction parameters.

  CMFTypes.Field.DateTimeField the DateTimeField by default uses the Plone calendar widget. An example:

```
DateTimeField('fact_date',
              form_info=DateTimeInfo(description="When does this fact originate from",
                                     label="Date"))
```

  CMFTypes.Field.LinesField in TEXTAREA's we can create :lines elements which basically allows you to seperate list elements by a \n. so you can write a element and when you hit enter it will end the list element and start a new one. An example:

```
LinesField('sources', form_info=LinesInfo())
```

  CMFTypes.Field.IntegerField is a input box which validates the input to make sure the data is numeric. An example:

```
IntegerField('age',
             form_info=IntegerInfo(description='How old are you?',
                                   label='Years of age'))
```

  CMFTypes.Field.SlotField need a field to be a PageTemplate element? You can use this in conjunction with a SlotInfo instance and pass in a path to a PageTemplate element. An example:

```
SlotField("about",
          form_info=SlotInfo(slot_metal="here/about_slot/macros/aboutBox"))
```

  The Field instances that are contained in the FieldList of the Type Descriptor for your BaseContent or ExtensibleMetadata usually serve teh purpose of defining the attribute and specifying the implementation. Some default definitions can be found in Products/CMFTypes/Form.py module. The base Form element implementaiton is FormInfo.py.

  NOTE We should realy specify the kwargs that are available for each class. i.e. MultiSelectionInfo can take `size` (which is the number of elements to show before scrolling) ? are the Field definitions confusing? should I remove them?

  CMFTypes.Form.FormInfo

- description [`string`] this is used in Plone as a form tooltip.

- label [`string`] this is the label on the form element that will be displayed.

- visible [1, 0, -1] setting these values will result in:

- 1 the form element being displayed
- 0 the form element not being displayed
- -1 the form element being hidden

The other form widget implementations that come stock with the Form module are:

StringInfo defines a input element. An example:

```
Field('quote',
      searchable=1,
      required=1,
      form_info=StringInfo(description="What are you quoting, what is the fact",
                           label="Quote"))
```

PasswordInfo an input type whos type="hidden". An example:

```
Field('weight',
      required=1,
      form_info=PasswordInfo(description="Your current weight, no one will see what you type.",
                             label="Weight"))
```

TextAreaInfo creates a TextArea form element. An example:

```
Field('teaser',
      searchable=1,
      form_info=TextAreaInfo(description="A short lead-in to the article so that we might get people to read the body",
                             label="Teaser",
                             rows=3))
```

LinesInfo will display a `textarea` that will be of type :lines. Which means you will be able to deliminate line items by a carriage return, \n. An example:

```
LinesField('sources', form_info=LinesInfo()),
```

KeywordsInfo this is a special FormInfo. It allows you to be able to pick from a prepopulated list of items or add your own. Its based on the Keywords form element in the `Properties` tab in Plone. **NOTE** seems like it requires some understanding of the implementation. An example:

```
LinesField('subject',
           multiValued=1,
           form_info=KeywordsInfo(label="Keywords"))
```

LanguageInfo will display a list of Languages. An example:

```
Field('language',
      default="en-US",
      form_info=LanguageInfo())
```

BooleanInfo can this be a checkbox? need example.

DateTimeInfo will have a plone calendar widget. An example:

```
DateTimeField('birthdate',
              form_info=DateTimeInfo(description="The day/month/year you were born",
                                     label="Birthdate"))
```

LinkInfo will enforce a valid URI. An example:

```
Field('url',
      form_info=LinkInfo(description="What URL did you find this information?",
                         label="URL"))
```

FileInfo displays a upload widget. An example:

```
Field('image', form_info=FileInfo())
```

IntegerInfo enforces that the value is a whole number integer. An example:

```
IntegerField('age',
             form_info=IntegerInfo(description='How old are you?',
                                   label='Years of age'))
```

FloatInfo An example:

```
Field('pi',
      form_info=IntegerInfo(description='Enter pi',
                            label='Pi, an infamous float'))
```

EmailInfo a input type="text" where the data must be a valid email address. An example:

```
Field('email_addr',
     form_info=EmailInfo(description='Enter your email address',
                         label='email'))
```

RichFormInfo Uses content drivers. An example:

```
Field('body',
      required=1,
      searchable=1,
      allowable_content_types=('text/plain', 'text/structured', 'text/html', 'application/msword'),
      form_info=RichFormInfo(description="Enter a valid body for this document. This is what you will see",
                             label="Body Text"))
```

SingleSelectionInfo you can specify two formats to display this widget, `radio` or `pulldown` . The field will require a defined vocabulary. In this example we use a module method called getAuthors() which returns a CMFTypes.utils.DisplayList instance. An example:

```
Field('author',
      vocabulary=getAuthors(),
      enforceVocabulary=1,
      form_info=SingleSelectionInfo(description='Author of title',
                                    label="author",
                                    format="pulldown"))
```

MultiSelectionInfo allow someone to select many options. An example:

```
Field('styles',
      vocabulary=getFightingStyles(),
      form_info=MultiSelectionInfo(description='Styles of combat',
                                   label="combat skills",
                                   size="2"))
```

SlotInfo specify a path expression to include. example:

```
SlotField("about",
          form_info=SlotInfo(slot_metal="here/about_slot/macros/aboutBox"))
```

In our collection (CMFTypes.Field.FieldList) of Fields objects we usually decorate them with a FormInfo (CMFTypes.Form.*) that gives specific behavior to a Field. Although CMFTypes comes with an impressive set of Field/FormInfo objects its very possible (and encouraged!) for you to create your own that is customized for your needs.

BaseObject, BaseUnit and Content Driving

BaseContent derives from BaseObject. This BaseObject is a containerish object which will handle form element data sources and how they are stored/behave in the ZODB. For instance if you ever have a File like form object it will be contained in the the BaseObject as a BaseUnit (CMFTypes.BaseUnit.BaseUnit). The BaseUnit is a wrapper around the File object and is responsible for "driving" the data that is housed in it. For instance on a Field that is decorated as a RichFormInfo will have the ability to upload a File of a certain type. When this File is uploaded it is up to the BaseUnit to store it and handle how it is updated and converted into something the system can use.

BaseUnit is a File object that is contained by the BaseContent. These objects are exposed through the web interface as a form widget. But you may also access the source of them programtically or through FTP/WEBDAV. The BaseUnit class is really responsible for rendering itself, and making sure that the content plays nicely with the CMFTypes framework. For instance if someone makes a change to the content - it will be reindexed. BaseUnit is really the bridge between Form element and the ZODB layer. You can also associate mime_types and/or extensions of files to a Content Converter aka Content driver. Usually the BaseUnit provides access/mutators. So if you create a FormInfo(`note`) it will be a BaseUnit and will create accessor, Note.

Converters aka Content Drivers can be found in CMFTypes/content_drivers/ContentType.py which gives you touchpoints into the registry. The registry will associate Plugins (converters) to a content type. CMFTypes ships with a MSWord converter that when you submit a MS Word document into a form field element (lets say RichFormInfo) it will put the file into a BaseUnit. The BaseUnit will then look up the registry for a Plugin for it to convert it into text and finally convert it and store the results in the BaseUnit. This is a extremely powerful design that will enable you to "extract" content from binary file formats. You could make it so someone may upload a Excel spreadsheet and generates a HTML file that creates a html table of data.

You will need to register your converter by importing `from Products.CMFTypes.content_drivers.ContentType import registerConverter` creating a ContentType class such as `OfficeDocument` or something similar that will house the data and actually know how to do the conversion of said Content Type.

The BaseUnit will take the content object from the REQUEST (form field element) find out what sort of ContentType to use, create a ContentType (say CMFTypes.content_drivers.OfficeDocument) that is populated with the data. Then will lookup the registered Converter and apply the `convertData` method on the loaded ContentType object.

The Example

Lets give CMFTypes a whirl. The object we want to build needs to have the following attributes:

- date: DateTime()
- headline: text
- blurb: text
- author: text
- title: text
- artist: text
- body: text
- pagenumber: integer

We will need to create a folder in the $ZOPE/lib/python/Products directory or create a symlink from another folder into the Products directory. Let the name of the directory in Products be, TypesExample.

Lets create 4 folders. These folders are where the CMFTypes machinery will output our forms, views and validation scripts. :

```
Products/TypesExample/Extensions
Products/TypesExample/skins
Products/TypesExample/skins/example_views
Products/TypesExample/skins/example_scripts
```

Now lets code:

- __init__.py : this file is responsible for registering your filesystem content objects into the ZOPE Application framework. It also defines the meta information that is needed for each content object to participate in the CMF framework.

Products/TypesExample/__init__.py:

```
from Products.CMFTypes import process_types
from Products.CMFTypes.Generator import generateViews
from Products.CMFTypes.utils import pathFor
from Products.CMFCore import utils
from Globals import package_home
import os, os.path

ADD_CONTENT_PERMISSION = 'Add example content'
PROJECTNAME = "TypesExample"

_types = {}

def registerType(type):
    _types[type.meta_type] = type

def listTypes():
    return _types.values()
```

```
def initialize(context):
    ##Import Types here to register them
    import Article

    homedir = package_home(globals())
    edit_dir = view_dir = os.path.join(homedir, 'skins', 'example_views')
    script_dir = os.path.join(homedir, 'skins', 'example_scripts')

    content_types, constructors, ftis = process_types( listTypes(),
                                                       PROJECTNAME,
                                                       edit_dir=edit_dir,
                                                       view_dir=view_dir,
                                                       script_dir=script_dir )
    utils.ContentInit(
        PROJECTNAME + ' Content',
        content_types      = content_types,
        permission         = ADD_CONTENT_PERMISSION,
        extra_constructors = constructors,
        fti                = ftis,
        ).initialize(context)

mfederighi - Dec. 8, 2002 8:39 pm:
 Where *is* CMFTypes? I couldn't find it. I also looked for the individual items to be imported, and they
 don't seem to be there either.
```

- Article.py : The content object we are trying to create is an Article object. It will have the following form elements on its edit view: date, headline, blurb, author, title, artist, body, pagenumber. These all have varying Form elements. Available Form elements can be found in Products/CMFTypes/Form.py

Products/TypesExample/Article.py:

```
from AccessControl import ClassSecurityInfo
from Products.TypesExample import registerType
from Products.CMFTypes.BaseContent import BaseContent
from Products.CMFTypes.ExtensibleMetadata import ExtensibleMetadata
from Products.CMFTypes.Field       import *
from Products.CMFTypes.Form        import *
from Products.CMFTypes.debug import log

def addArticle(self, id, **kwargs):
    o = Article(id, **kwargs)
    self._setObject(id, o)

class Article(BaseContent):
    portal_type = meta_type = "Article"

    type = BaseContent.type + FieldList(
      (
        Field( 'blurb'
             , searchable=1
             , required=0
             , form_info=StringInfo(description="Enter a blurb for this article"
                                    ,label="Blurb"
                                    ))
      , LinesField('body'
            , searchable=1
            , required=1
            , form_info=TextAreaInfo(description="The article content.  &lt; break /&gtr; denotes a page break"
                                     ,label="Content"
                                     ))
      , Field( 'secret'
            , required=1
            , form_info=PasswordInfo(description="ssssh"
                                     ,label="passwd"))
      ))

registerType(Article)
```

- Installation Mechanism : the CMF has a fairly unusual way of registering content object with the framework. We do this using External Methods. The convention is to put External Method modules in the Extensions folder. The Install.py will be used later as a External Method to load the content objects into CMF as well as generate the forms.

Products/TypesExample/Extensions/Install.py:

```
from Products.CMFTypes.Extensions.utils import installTypes
from Products.TypesExample import listTypes, PROJECTNAME
from StringIO import StringIO

def install(self):
    out = StringIO()
    installTypes(self, out, listTypes(), PROJECTNAME)
    print >> out, "Successfully installed Article."
    return out.getvalue()

jean - Nov. 1, 2002 12:32 am:
 This looks really promising, but when I follow these steps, the result is:
 Error Type
 Bad Request
 Error Value
 Example: Article not found.
jean - Nov. 1, 2002 1:59 am:
 OK, that one was simply:
 -PROJECTNAME = "Example"
 +PROJECTNAME = "TypesExample"
 See the thread "here":http://sourceforge.net/mailarchive/forum.php?thread_id=1257870&forum_id=8090
 for why that matters ..
mrtopf - Dec. 7, 2002 6:11 pm:
 Zope must be restarted in order to make the new product active. Maybe this should be noted here.
```

- Now in the root of your Plone/CMF Site create an External method and click save. Then click the test tab. The External method should look like:

```
   id: inst
   title: install mechanism (optional)
   module: TypesExample.Install
   function: install

- You should see the ouput you printed in TypesExample/Extensions/Install.py

- Your directory structure should now be populated with the following::

  Products/TypesExample/__init__.py
  Products/TypesExample/Article.py
  Products/TypesExample/Extensions
  Products/TypesExample/Extensions/Install.py
  Products/TypesExample/skins
  Products/TypesExample/skins/example_scripts
  Products/TypesExample/skins/example_scripts/validate_article.py
  Products/TypesExample/skins/example_views
  Products/TypesExample/skins/example_views/article_edit_form.pt
  Products/TypesExample/skins/example_views/article_view.pt
```

Using the form type descriptors for BaseContent.type and ExtensibleMetadata.metadatatype your class can specify what sort of fields/attributes on your Content object will exist. When ZOPE starts up or you refresh your project the python code will call Products.CMFTypes.process_types() which in turn will call CMFTypes.ClassGen.generateClass and CMFTypes.Generator.generateViews.

CONTEXT: We are talking about how ContentObjects that define a .type descriptors (FieldList) have methods and security assertions generated. This is a developer chapter that attempts to explain how the core of CMFTypes works.

CMFTypes.ClassGen (Module)

This is responsible for generating class instances based on the FieldList. This generates class definitions at runtime using the FieldList attribute on your content object. It generates accessors/mutators/editable methods for all of the Fields. If you specify any of these methods on the Field during construction it will use those instead of computing the method at runtime. It will also apply ZOPE Security assertions on your methods.

typical CMFTypes.ClassGen.generate(klass) lifecycle:

```
for field in ContentObject.type (FieldList):
    iterate through all of the Field.mode:
        if you have specified a alternative accessor/mutator/editable get a \
          reference for that method else grab a default implemnetation
        check to see if we need to force a re-computation of the method or it \
          has yet to be computer:
            make the method()
            create security assertions()
        bind the method to the klass instance
InitializeClass(klass) #class instance is re-initialized into ZOPE
```

So the CMFTypes framework appears to (at module loading) generate needed class instances based on the definitions that have been registered with a Product via registerType().

NOTE I'm not quite sure if its really "generating" class instances or if its decorating them with generated methods. It seems like its the latter. Because the klass that is getting passed to generateClass() is a class instance that has been registered. So it appears that its modify hte class instances and then "re-registers" them after computing all of the methods. The makeMethod() uses exec() which he generates the method from the templates directory.

CMFTypes.Generator (Module)

Responsile for generating Page Templates (xxx_view, xxx_edit_form) and Python Scripts (validation scripts, xxx_validate and mutator scripts, xxx_edit) . These will be accessible in the Filesystme Directory view (that is registered by your project) and can be `customized` into the ZODB. The "templates" that are used to generate the various FS templates/scripts can be found in Products/CMFTypes/templates. These are text files that use python string substituion, %s.

CMFTypes.Generator.FormGenerator is created in Generator module and we bind the module method generateViews to the FormGenerator's generateViews method. We then call generateViews(klass, targetfolder) on a class with the output folder as arguements. The method will then generate what `views` you have passed in to generateViews() by default [`edit` , `view` , 'validate'] are generated into the target folder.

NOTE Base Generators get a reference to the Field that is being generated and the class instance that it belongs to (Content Object).

typical CMFTypes.Generator.generateViews lifecycle:

```
for field in ContentObject.type (FieldList):
  if the form field is visible/hidden:
    append fieldgenerator from fieldgeneratorfactory to fieldgenerators sequence
  iterate over the types of views:
    specify the output folder
    create a filesystem file, fp
    generate Header for fp
    then iterate through the form generators:
      ask teh fieldgenerator(fp) to fill out template to fp
    generate Footer for fp
```

The generators are quite elegant and since ech Form Field widget bascially has derives from a Generator (and specifies a unique template) to be able to fulfill its contract i.e. a Keyword(Generator) must write out Javascript as well as Keywords box and Add New Item. (i.e. the Keywords field on the Properties tab).

Bugs/Issues:

10/16/02 -

- When you run the external method it does not install subfolders.
- Need to add registerDirectory() to the __init__
- needs to check the skinpaths for existence of the edit_dir/script_dir
- Password validation doesnt work
- Throws excption when using Integer or RichFormFields in BaseContent
    - DDocument is broke