Archetypes Basic Reference

Author: Sidnei da Silva
Contact: sidnei@plone.org

Date: $Date$
Version: $Revision$
Web site: http://sourceforge.net/projects/archetypes

Contents

Introduction

Archetypes is a framework for developing new content types in Plone. The power of Archetypes is, first, in automatically generating forms; second, in providing a library of stock field types, form widgets, and field validators; third, in easily integrating custom fields, widgets, and validators; and fourth, in automating transformations of rich content.

The project is hosted on the Archetypes Project at SourceForge. The latest version of this document can be always found under the under the docs directory of Archetypes.

Installation

Requirements

Archetypes is currently being tested and run in various environments using the following combination:

- Zope 2.6.2+
- CMFPlone 1.0.4
- CMF 1.3.1

It is also known to work smoothly with Zope 2.5.

You should install the *validation* and *generator* packages available on the archetypes'sourceforge page before installing Archetypes itself. WARNING: those packages was used to be installed as Zope products, this not the case anymore. They should be installed as regular python package (look at the packages'README file for more info).

Using the tarball

1. Download the latest stable version from the Archetypes Project on Sourceforge.
2. Decompress it into the `Products` dir of your Zope installation. It should contain the following directories:

   ```
   Archetypes
   ArchExample
   ```

3. You should install the *validation* and *generator* packages available on the Archetypes Project page before installing Archetypes itself.

   WARNING: those packages used to be installed as Zope products, this not the case anymore. They should be installed as regular python package (look at the packages README file for more info).
4. Restart your Zope.
5. Check in the `Control Panel` of your Zope if everything imported just fine.
6. Good luck!

Checking out from CVS

Using Windows

If you want to get the latest version of Archetypes from CVS, here is how to do it.

1. Get TortoiseCVS from http://prdownloads.sourceforge.net/tortoisecvs/TortoiseCVS-1-2-2.exe
2. Download and install the program.
3. Reboot if necessary.

XXX Need more info here.

Using `*nix`

Quick and dirty:

```
cvs -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/archetypes login
cvs -z3 -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/archetypes co Archetypes
cvs -z3 -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/archetypes co ArchExample
cvs -z3 -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/archetypes co validation
cvs -z3 -d:pserver:anonymous@cvs.sourceforge.net:/cvsroot/archetypes co generator
```

Schema

The heart of an archetype is its `Schema`, which is a sequence of fields. Archetypes includes three stock schemas: BaseSchema, BaseFolderSchema, and BaseBTreeFolderSchema. All three include two fields, 'id' and 'title', as well as the standard metadata fields.

The `Schema` works like a definition of what your object will contain and how to present the information contained. When Zope starts up, during product initialization, Archetypes reads the schema of the registered classes and automagically generates methods to access and mutate each of the fields defined on a Schema.

Fields

You add additional fields to a schema by using one of available field types. These fields share a set of properties (below, with their default values), which you may modify on instantiation. Your fields override those that are defined in the base schema.

More commonly used field properties:

required Makes the field required upon validation. Defaults to 0 (not required).

widget One of the Widgets to be used for displaying and editing the content of the given field.

Less commonly used field properties:

default Sets the default value of the field upon initialization.

vocabulary This parameter specifies a vocabulary. It can be given either as a static instance of DisplayList or as a method name (it has to be the name as a string). The method is called and the result is taken as the vocabulary. The method should return a `DisplayList`.

The contents of the vocabulary are used as the values which can be choosen from to fill this field.

An example for a `DisplayList` usage can be found in the `ArchExample` directory in `config.py`.

enforceVocabulary If set, checks if the value is within the range of `vocabulary` upon validation

multiValued If set, allows the field to have multiple values (eg. a list) instead of a single value

isMetadata If set, the field is considered metadata

accessor [1]

Name of the method that will be used for getting data out of the field. If the method already exists, nothing is done. If the method doesn't exist, Archetypes will generate a basic method for you.

edit_accessor Name of the method that will be used for getting data out of the field just before edition. Unlike the standard accessor method which could apply some transformation to the accessed data, this method should return the raw data without any transformation. If the method already exists, nothing is done. If the method doesn't exist, Archetypes will generate a basic method for you.

mutator Name of the method that will be used for changing the value of the field. If the method already exists, nothing is done. If the method doesn't exist, Archetypes will generate a basic method for you.

mode One of `r`, `w` or `rw`. If `r`, only the accessor is generated. If `w` only the mutator and the edit accessor are generated. If `rw`, accessor and mutator and edit accessor are generated.

read_permission Permission needed to view the field. Defaults to CMFCorePermissions.View. Is checked when the view is being auto-generated.

write_permission Permission needed to view the field. Defaults to CMFCorePermissions.ModifyPortalContent. Is checked when the submitted form is being processed..

storage One of the Storage options. Defaults to `AttributeStorage`, which just sets a simple attribute on the instance.

generateMode Deprecated?

force Deprecated?

validators One of the Validators. You can also create your own validator.

index A string specifying the kind of index to create on `portal_catalog` for this field. To include in catalog metadata, append `:schema`, as in `FieldIndex:schema`. You can specify another field type to try if the first isn't available by using the `|` character. Both combinations can be used together, as in:

```
...
index="TextIndex|FieldIndex:schema",
...
```

schemata Schemata is used for grouping fields into `fieldsets`. Defaults to `default` on normal fields and `metadata` on metadata fields.

Here is an example of a schema (from 'examples/SimpleType.py'):

```
schema = BaseSchema + Schema((
  TextField("body",
        required=1,
        searchable=1,
        default_output_type="text/html",
        allowable_content_types=("text/plain",
                                 "text/restructured",
                                 "text/html",
                                 "application/msword"),
        widget  = RichWidget,
        ),
  ))
```

Validators

Archetypes provides some pre-defined validators. You use them by passing a sequence of strings in the `validator` field property, each string being a name of a validator. The validators and the conditions they test are:

inNumericRange The argument must be numeric

isDecimal The argument must be decimal, may be positive or negative, may be in scientific notation

isInt The argument must be an integer, may be positive or negative

isPrintable The argument must only contain one or more alphanumerics or spaces

isSSN The argument must contain only nine digits (no separators) (Social Security Number?)

isUSPhoneNumber The argument must contain only 10 digits (no separators)

isInternationalPhoneNumber The argument must contain only one or more digits (no separators)

isZipCode The argument must contain only five or nine digits (no separators)

isURL The argument must be a valid URL (including protocol, no spaces or newlines)

isEmail The argument must be a valid email address

[1] Depending on the mode of each Field in the Schema the runtime system will look for an accessor or mutator. If, for example, the mode of a field is "rw" (as is the default), then the generator will ensure that accessors and mutators exist for that field. This can happen one of two ways: either you define the methods directly on your class, or you let the generator provide them for you. If you don't require specialized logic, then letting the generator create these methods on your new type is a good idea.
The format for accessors and mutators is as follows:

```
field -> title

accessor -> getTitle()          here/getTitle
mutator  -> setTitle(value)
```

The current usefulness of Archetypes' validators is mitigated by weak error messaging, and the lack of support for separators in SSNs, phone numbers, and ZIP codes.

There are also hooks for pre and post validation that can be used to assert things about the entire object. These hooks are:

```
pre_validate(self, REQUEST, errors)
post_validate(self, REQUEST, errrors)
```

You must extract values from `REQUEST` and write values into `errors` using the field name as the key. If `pre_validate` throws errors, then other custom validators (including post) will not be called.

Writing a custom validator

If you need custom validation, you can write a new validator in your product.:

```
from validation.interfaces import ivalidator
class FooValidator:
    __implements__ = (ivalidator,)
    def __init__(self, name):
        self.name = name
    def __call__(self, value, *args, **kwargs):
        if value == 'Foo':
            return """Validation failed"""
        return 1
```

Then you need to register it in FooProduct/__init__.py method initialize:

```
from validation import validation
from validator import FooValidator
validation.register(FooValidator('isFoo'))
```

The validator is now registered, and can be used in the schema of your type.

Widgets

When Archetypes generates a form from a schema, it uses one of the available Widgets for each field. You can tell Archetypes which widget to use for your field using the `widget` field property. Note, though, that a field cannot use just any widget, only one that yields data appropriate to its type. Below is a list of possible widget properties, with their default values (see 'generator/widget.py'). Individual widgets may have additional properties.

attributes Used for??

description The tooltip for this field. Appears in response to `onFocus`.

description_msgid i18n id for the description

label Is used as the label for the field when rendering the form

label_msgid i18n id for the label

visible Defaults to 1. Use 0 to render a hidden field, and -1 to skip rendering.

Views

Views are auto-generated for you by default, based on the options you specified on your `Schema` (Widgets, Fields, widget labels, etc.) if you use the default FTI actions (eg: don't provide an `actions` attribute in your class. See Additional notes about Factory Type Information).

Customizing Views

If you want only to override a few parts of the generated View, like the header or footer, you can:

1. Create a template named `${your_portal_type_lowercase}_view` [2]
2. On this template, you may provide the following macros:

   ```
   header
   body
   footer
   ```

3. When building the auto-generated view, archetypes looks for these macros and includes them in the view, if available. Note that the body macro overrides the auto-generated list of fields/values.

Or, for customizing only a widget:

1. Set the attributes `macro_view` or `macro_edit` to the location of your custom macro upon instantiation of the Widget.
2. Your custom macro template must contain a macro with the same name as the mode where it will be used. Eg: a template that is being used on `macro_view` must have a macro named `view`. The same applies to `macro_edit` and `edit`.

Class Attributes

Besides the schema, you can define all of the content properties you see when you click on a content type in the 'portal_types' tool. Here is a list of class attributes, with their default values (see 'ArchetypeTool.py'):

Default class attributes/methods

modify_fti (**method**)

Is looked up on the module and called before product registration. Works as a hook to allow you to modify the standard `factory type information` provided by Archetypes.

add${classname} (**method**)

Is looked up on the module. If it doesnt exist, a basic one is autogenerated for you.

content_icon A name of an image (that must be available in the context of your object) to be used as the icon for your content type inside CMF.

global_allow Overrides the default `global_allow` setting on the default factory type information.

allowed_content_types Overrides the default `allowed_content_types` setting on the default factory type information. If set, supercedes the `filter_content_types` in case it is not provided on the class.

filter_content_types Overrides the default `filter_content_types` setting on the default factory type information.

Additional notes about Factory Type Information

- If your class declares to implement `IReferenceable`, you will get a `references` tab on your object, allowing you to make references to other objects.
- If your class declares to implement `IExtensibleMetadata`, you will get a `properties` tab on your object, allowing you to modify the metadata.
- Custom actions: Define an actions member on your content type and the external method will apply this to the types tool for you. This means the if you want custom views or something you only need to say something like:

[2] Currently, this is only implemented for the auto-generated `view` template.

```
class Foo(BaseContent):
    actions = ({'id': 'view',
                'name': 'View',
                'action': 'custom_view',
                'permissions': (CMFCorePermissions.View,)
               },)
```

Storage

There are a few basic storages available by default on Archetypes, including storages that store data on SQL. Here's a listing:

AttributeStorage Simply stores the attributes right into the instance.

MetadataStorage Stores the attributes inside a `PersistentDict` named `_md` in the instance.

ReadOnlyStorage Used to mark a field as being `ReadOnly`

ObjectManagedStorage Uses the `ObjectManager` methods to keep the attribute inside the instance. Allows you to make a folderish content object behave like a simple content object.

`*SQLStorage` Experimental storage layer, which puts the data inside SQL. Available variations are: MySQL and PostgreSQL. There's an initial implementation of an Oracle storage, but it isn't tested at the moment.

Marshall

From The Free On-line Dictionary of Computing (09 FEB 02) [foldoc]:

> marshalling
>
> > <communications> (US -ll- or -l-) The process of packing one or more items of data into a message {buffer}, prior to transmitting that message buffer over a communication channel. The packing process not only collects together values which may be stored in non-consecutive memory locations but also converts data of different types into a standard representation agreed with the recipient of the message.

Marshalling is used in Archetypes to convert data into a single file for example, when someone fetches the content object via FTP or WebDAV. The inverse process is called `Demarshalling`.

Archetypes currently has a few sample marshallers, but they are somewhat experimental (there are no tests to confirm that they work, and that they will keep working). One of the sample marshallers is the `RFC822Marshaller`, which does a job very similar to what CMF does when using FTP and WebDAV with content types. Here's what happens, basically:

1. Find the primary field for the content object, if any.
2. Get the content type for the primary field and its content.
3. Build a dict with all the other fields and its values.
4. Use the function `formatRFC822Headers` from `CMFCore.utils` to encode the dict into RFC822-like fields.
5. Append the primary field content as the body.
6. Return the result, content_type and data.

When putting content back, the inverse is done:
1. The body is separated from the headers, using `parseHeadersBody` from `CMFCore.utils`.
2. The body, with the content type, is passed to the mutator of the primary field.
3. For each of the headers, we call the mutator of the given matching field with the header value.
That's it.

An example of using a Marshaller

To use a Marshaller, you just need to pass a Marshaller instance as one of the arguments for the Schema. For example:

```
from Products.Archetypes.Marshall import RFC822Marshaller
class Story(BaseContent):
    schema = BaseSchema + Schema ((

        TextField('story_description',
                  primary = 1,
                  default_output_type = 'text/plain',
                  allowable_content_types = ('text/plain', 'text/restructured',),
            widget = TextAreaWidget(label = 'Description',
                                    description = 'A short story.'
                                    )),

        ),
        marshall = RFC822Marshaller())
```

Examples and more information

Examples can be found on the ArchExample product, that is included in the download. You can also browse the cvs repository.

Special Thanks

To Vladimir Iliev, for contributing with i18n and lots of other nice ideas and Bill Schindler, for lots of nice patches and reviewing documentation.