EnTransit Deployment Server

Version 0.7.0

22nd July 2005

Organization: Enfold Systems, LLC
Author: Sidnei da Silva <sidnei@enfoldsystems.com>
Author: Alan Runyan <alan@enfoldsystems.com>
Date: 2005-07-15 22:58:55 -0300 (Fri, 15 Jul 2005)
Revision: 1137

Abstract

An in-depth look at the nuts and bolts of EnTransit, Enfold Systems' solution for cross-framework content deployment.

Table of Contents

1 Introduction

This system is beta software that enables a content deployment strategy. This means content from inside of Plone can be deployed to a remote machine regardless of platform. The remote machine is responsible to deploy the content. During the Plone Symposium several speakers will demonstrate Entransit Delivery Frontends.

Currently PHP, Python, Java and .NET examples exist. The developers are at the New Orleans Symposium. You can contact the developers or info@enfoldsystems.com for more information.

1.1 Status Quo

- Plone is very simple to use. configure. add functionality.
- Plone is built on Zope. Built using Python. Large stack.
- Missing story for the "Rest of the World", JSP/PHP/ASPX/Ruby/etc.
- Plone has grown the Zope world. Next step. Expose data to other systems.

1.2 Integration not Isolation

- Consuming data sources in Zope is trivial (python is great glue language)
- What about Zope being a producer of content and others subscribing? Limited.
- Many people think of Plone as a black box (look how many default sites exist)
- Lets take it a step further and declare it a black box.
- Add the ability to push the content from Plone to a remote machine in a structured manner

1.3 Zope as Producer

- Relational Database Adapters, data resides in RDBMS
- RSS, clients can intermitently get RSS feeds which may drive other rules
- Filesystem. Writing files to filesystem.
- Messaging systems. Maybe XMLBlaster. MSMQ on Windows. JMS integration?
- Email. Using smtplib. (Sending email from Zope)
- HTTP/DAV/FTP. Running servers where consumers come to Zope.

2 Problem Domain

It is important to understand the Problem Domain of the typical consulting firm. Clients have been burned by large integration companies or software companies. Looking for incremental win and a measurable/visible Return On Investment. As low as possible entry and be sold longer term visions through evolutionary prototyping. The days of clients spending 250k to rebuild large website and paying 125k for services are coming to an end. Vignette, Intewoven and Documentum - all see the handwriting on the wall.

2.1 Java/JSP Integration Problem

Scenario: Client wants to incrementally roll out CMS functionality. They want to use Plone. They have an existing Java web system. There is a site section called *Company* that contains press releases, management team information and office locations/hours. **How can we offer a solution?**

2.2 Proposed Solution Strategies

- Use Apache of Enfold Proxy to rewrite /Company section to be served from Plone. Problems: Maintain 2 "template sets". To change the company logo in the templates; may need to update both Plone and Java System.
- Have Plone walk a container and render the contents as HTML using a template identical to Java web system. Same problem as above. Also Java API has no way to manipulate the data found on the Filesystem.
- Put the content from Plone into RDBMS. Or render it as XML to the filesystem. These are better strategies but what is the "process" that wraps this strategy? Or better yet what if Plone is not available on the delivery server?
- Have Java use Zope in a Web Services capacity. Not easy. Zope provides no native Web Services engine/integration. Objectrealms made some headway. What if the Zope/Plone system is behind a firewall which can not be accessed from the web site.

2.3 Entransit, A Solution

- Run a daemon on the 'target deployment' server.
- Minimal product bundles data as primitives and sends to 'target' server
- 'target' server (Entransit Delivery Server) process the content through a pre-configured pipeline of operations.
- The pipeline of operations is transaction-aware; they either complete or abort.
- Default stores XML on filesystem and DC/RDF in RDBMS

2.4 XML, RDBMS, RDF, Oh My!

- NOTE: XML/RDF does not need to be discussed. Think of Entransit as the middleware to a remote machine's datastore.
- Close your ears for the rest! DO NOT LISTEN.
- The system must be extensible. Take a *Choose Your Poison* approach.
- Allow developers to use familiar toolchain.
- Possibility to just use XSLT/XML; in Apache or IIS.
- Cross platform, language neutral. Low bar of entry.
- RDF adds tremendous flexibility but adds sophisticated, optional, relationships such as Translations/Containment.
- Optional Flexibility/Sophistication is Great!

2.5 Proposed Java Integration

- Plone is a black box. Services around black box. Explicit deployment contract (pipeline).
- Maybe create a servlet called 'Company' or use WEB-INF/Manifest to bind to Company.
- Reuse the existing Java/JSP/Template. Pick up the data externally from Filesystem and/or RDBMS
- Ability to create touch points to extend before/during/after pipeline processing to notify Java Web System.
- Developers use familiar toolchain (language, platform, IDE, processes).
- Search engine is done by external system; better documented, faster, and more flexible.

3 Plone, Now and Future

- Plone covers 70% of the feature sets of most CMS systems.
- With the absolute low cost of entry - its hard to pass up.
- Plone is one of the easiest and most intuitive interfaces on the market
- I would estimate over 10 million (if not much more) in revenue is being generated soley by Plone marketing and service oriented companies.
- I propose to make Plone even easier to use. And leverage Entransit as the “deployment” middleware for medium to large sized organizations.
- Time to show CMS market that Plone is no longer an island but an integration application eager to play nicely.

4 Entransit, the Implementation

4.1 Entransit, the Philosophy

- Lets try to follow the Python Zen. Keep it simple.
- Explicit is better than implicit. We will explicitly extract data from the content on the Zope/Plone system. The processing target server knows what the expect.
- Complex is better than Complicated. There is almost 0 work done on the Zope/Plone end.
- Flat is better than nested. The current implementation has is very flat. We are not providing for nested namespaces until we actually need it.

4.2 Entransit, the Architecture

- Zope bundles content into a flat datastructure.
- Datastructure is sent over a socket to a 'target' server.
- 'Target' server iterates over order of operations and applies operations on each content.
- The operations are then committed or aborted.
- Zope holds a connection to the remote server through the process. (connection-oriented protocol/ZEO)

5 Installation

Will be made more simple in a commercial product. Open Source will most likely stay “framework” oriented. We focus on Windows being the “Plone Server” and Windows/Linux being the Target Server.

5.1 Plone Installation

- Download Enfold Server 1.2 with Zope 2.8.
- Download Entransit tarball
  - Copy src/entransit into $ENFOLD/Zope/lib/python
  - Copy products/* into $ENFOLD/Products
- In ’Site Setup’ (Plone UI) install products.

5.2 Entransit Target Installation

- Download Entransit tarball
- Extract Entransit tarball into /home/eed
- Edit configuration file
- Start server (note the listening IP and port)

5.3 Delivery Configuration

- Install SQLITE 2
- Install pysqlite 1.x series (stable)
- Install apache/mod_python
- Install vampire

5.4 Test Deployment

- Ensure the Workspace is configured with appropriate target
- Add some content in the Public Node
- Go back to workspace
- Deploy the workspace.
- Look at the delivery system to see if content appears.

6 Explaining the configuration file

EnTransit configuration uses the `ZConfig` library for providing a simple and extensible Apache-like configuration. The configuration file is the heart of the system and it's the first place (and in most cases the last) a EnTransit user will touch.

The configuration file has a few global options, and then a fixed set of `pipelines` where you define components that will be executed in the order they are found in the pipeline.

Here's an example configuration file:

```
%define TARGET /tmp/deployment
%define BACKUP /tmp/backup
%define FILE_PERM 0644
%define DIR_PERM 0755
%define URL http://awkly.org/files
%define CONNECTION_STRING sqlite:///tmp/test.sqlite
%define ADDRESS 127.0.0.1:9001

<config>
target-path $TARGET
directory-creation-mode $DIR_PERM
address $ADDRESS

<pipeline before-hooks>
  <command>
    path src/entransit/scripts/report/start.sh
  </command>
</pipeline>

<pipeline after-hooks>
  <command>
    path src/entransit/scripts/report/finish.sh
  </command>
</pipeline>

<pipeline storage>
  <urlrewrite>
    # Must come first. It rewrites URLs in marshalled files in-place,
    # before they are moved to the destination stage by the datastorage
```

```
    # component.
    engine entransit.storage.url.UIDToURL
    base-url $URL
    prefix $TARGET
  </urlrewrite>
  <datastorage>
    backup-path $BACKUP
  </datastorage>
  <fileperms>
    file-mode $FILE_PERM
    directory-mode $DIR_PERM
    storage-name data-storage
  </fileperms>
  <blobstorage>
    backup-path $BACKUP
  </blobstorage>
  <fileperms>
    file-mode $FILE_PERM
    directory-mode $DIR_PERM
    storage-name blob-storage
  </fileperms>
  <command>
    path src/entransit/scripts/report/step.sh
  </command>
  <archetypes-triple-storage>
    connection-string $CONNECTION_STRING
    backup-path $BACKUP
    target-path $TARGET/relationships.rdf
  </archetypes-triple-storage>
</pipeline>

<pipeline metadata>
  <sqlobjectmeta>
    implementation entransit.metadata.CMFMeta
    connection-string $CONNECTION_STRING
  </sqlobjectmeta>
</pipeline>

</config>
```

The components in the pipeline are always called with the same order and components in the same pipeline will always be called with the same payload.

All operations executed by components in the pipeline should happen transactionally.

Components that plug in the pipeline should work in isolation and should not affect any global state except the minimal necessary to coordinate with the main transaction. However if some dependency between two components in the pipeline they can use a dictionary that is shared

by all components in the pipeline for communicating some state.

6.1 Pipelines

We have defined five pipelines. Those should suit most of the use-cases. The existing pipelines are:

marshall Convert incoming data to a flat file representation

storage Takes care of storing the file representation and any other infomration concerning the file, transactionally

metadata Handles the storage of metadata *about* the file, like last modification, etc

before-hooks Executed at the start of the deployment operation

after-hooks Executed at the end of the deployment operation

The flow across the pipelines goes like this:

1. The `before-hooks` pipeline is executed. This only happens once, at the start of the deployment operation. No information about the content being deployed is passed to this pipeline.
2. For each entry deployed, the content goes first through the `marshall` pipeline, then through the `storage` pipeline and then through the `metadata` pipeline.
3. After all entries have been processed, the `after-hooks` pipeline gets executed.

6.2 Components

There are a couple default components provided with the system. They were designed for a specific use-case which was the target goal since we started planning the system so might not fit all the needs, but we hope they can be useful for most of the cases and also serve as starting point for creating new components.

6.2.1 XMLRPCLib Marshaller

Marshalls incoming data to the format used for XML-RPC communication using the `xmlrpclib` library. This was used because:

1. There are libraries available for parsing this simple format in most existing languages/frameworks.
2. It's a very simple and still readable XML format, so if there's no parser provided by default you can easily write one.

To change the format which incoming data is marshalled to, you just need to replace this component in the configuration file.

6.2.2 URL Rewriter

Content being deployed might have URLs pointing to the server where they are deployed from, specially with Zope.

One of the goals of our use-case was to use `kupu` to create content in Plone and then deploy content. `kupu` when used in conjunction with `Archetypes` generates urls that look like:

```
<a href="resolveuid?uid=a89073b893240c899008d0988903f89980" />
```

When inside Zope, `resolveuid` is actually a script that will resolve the UID to a URL and direct the user there. While we *could* do the same, instead we went a step further and created a component to resolve those UIDs to URLs at deployment time, on the deployment server.

This is more of a `proof of concept` as it does depend on objects being pointed by the UID to have been already deployed by the time the UID gets resolved to a URL, but will work just fine if that constraint is fulfilled.

The `core` of the rewriting engine can be changed by providing a different dotted path to a class on the `engine` parameter of the `<urlrewrite>` directive.

6.2.3 Data Storage

The Data Storage takes the file (or files) created by the `marshall` pipeline and stores them in the hierarchy pointed by the `target-path` global config option.

There is one configurable argument for the Data Storage which is `backup-path`. This is used to backup existing files that are being updated or removed by the deployment operation. Once the operation is finished sucessfully those backup files are discarded, and if the operation fails to finish those backup files are restored to their original location. This provides a `transactional` update to the target repository.

6.2.4 Blob Storage

This is basically the same as the Data Storage, but instead of consuming files generated by the XMLRPCLib Marshaller, it does consume files generated by the Blob Marshaller which in turn doesn't do much more than picking up data marked as `blob-ish` and dump it unmodified to a file.

6.2.5 File Permissions

The File Permissions component updates the permission of the files in the target repository after they have been created/updated. They are dependent on the storage being used and use the state sharing facility to discover the filename of the file changed by the storage component they are bound to.

6.2.6 Archetypes Triple Storage

We keep information about Archetypes relationships between entries deployed by using a Triple Store. This component makes use of the `rdlib` library, backed by a SQLObject back-end, which in turn stores the triples in a relational database.

Upon any modification to the Triple Store, the system is notified and a component is dinamically appended to the `after-hooks` pipeline, so that if at the end of the deployment operation the entire RDF graph is dumped to a file on the filesystem using the RDF-XML representation.

6.2.7 Command Component

The Command Component is the only one that is not expected to have transactional behaviour (though it could in theory). It does basically execute a system call to run a executable or script.

Out-of-the-box, EnTransit comes configured with three command components: One is executed on the `before-hooks` and creates a `report` file recording the time the deployment operation started. The seconds sits in the `storage` pipeline and writes a simple record for each entry deployed. And the last one is executed on the `after-hooks` pipeline and writes a marker to the report file to log the time the deployment operation finished.

6.3 Client-Server Communication

Communication between the client and server is done using the `zrpc` library, originally designed for `ZEO`. `zrpc` does basically `Remote Procedure Call`, like `xml-rpc` for instance, but instead of using a xml-based protocol it uses the pickle python module.

How it works:

1. The client has a `stub` where you can call methods
2. The method name and arguments are pickled and sent over the wire
3. On the other side, the method name and arguments are unpickled
4. The server then calls the method on the server-side handler
5. The return value of the method call is then pickled and sent back to the client
6. The client-side unpickles the returned values and returns then to the caller in step 1

Another nice feature is that if an exception happens during the method call on the server side, the exception is pickled and sent to the client which then re-raises the exception as if it had happened locally.

Care must be taken so the arguments passed here are picklable and can be unpickled on the server. Remember the server needs to be able to find the classes of objects you send as arguments otherwise the server will complain and disconnect you.

6.4 Deploying content

To make things even more simple, we have simplified the API to a single method called `deploy`. The only way to modify state on the server is by calling this method. Other than that, the server acts like a black box.

This function takes only one argument, `struct` which is a list of triples with:

key It's the identifier for that specific entry. By default it's used as the relative filename from the target deployment path.

op Operation to be performed. Can be one of 'add', 'update' or 'remove'.

data Payload for that specific entry.

The default handler has a very simple flow that can be described like this:

1. Start a transaction
2. For each entry, execute the operation requested with the given payload
3. If the transaction was successful, commit the transaction
4. If the transaction failed, rollback the transaction

6.4.1 Transaction Coordination

We use the `transaction` library. that was factored out from ZODB somewhere after ZODB 3.2 mainly for inclusion within Zope 3.

In order to coordinate with the transaction's begin/abort/commit, components can register a `Data Manager` with the main `Transaction Manager`. When any meaningful operation is executed by the Transaction Manager, a hook notifies the Data Managers that should then react accordingly.

We have done this in a couple places for Entransit.

6.4.2 SQLObject Integration

SQLObject provides a `Transaction` abstraction for standalone use. Any time you want to perform some operation in SQLObject and you want that operation to be transactional, you get hold of a Transaction object and call it's `commit` or `rollback` methods at your leisure.

For EnTransit, we have registered a facade Data Manager that delegates the `commit` and `abort` calls down to SQLObject Transaction's `commit` and `rollback` methods accordingly.

The SQLObject Transaction is created implicitly, that is, only when needed. So if some entry being deployed doesn't trigger a component that uses SQLObject, the SQLObject Transaction and the corresponding Data Manager will not be registered with the main Transaction Manager.

6.4.3 Filesystem Integration

EnTransit uses the `Command` pattern for operations performed with files on the filesystem.

Each operation performed results in the creation of a `Command` object that is then appended to a stack. This stack happens to also be a Data Manager that is registered with the Transaction Manager.

When the Data Manager is notified of a `commit`, each command in the stack is executed in the order they were created.

When the Data Manager is notified of a `abort`, each command in the stack is reversed by calling the `revert` method of the `Command` object. The order of the commands is also reversed, so the last command created is executed first, etc.

7 Collecting data on the Client

The easiest and most effective way of doing deployments from a client is to keep some information about the state of the client system at the time the last deployment has been executed and do a incremental deployment. Note that is not required, but helps a lot to keep the consistency of the deployed site and to save time and bandwidth.

Building a application from scratch that did this and integrated with Plone is a non-trivial task. However, we already had a system that did something similar so extending that system a bit to plug the deployment client into it was a snap.

To make the task simpler, we did define a simple set of common attributes that we wanted to see published from Plone into the deployment target and a small registry of functions to be called for each attribute. After running a content object through this, a dictionary mapping attribute to value would be generated.

Later on, we turned this into an `adapter`, so people can register more specific adapters to their custom content type and more easily extract the needed information without having to touch the source.

7.1 EnSimpleStaging (Enfold Simple Staging)

`EnSimpleStaging` is a custom staging implementation using `CMFStaging` and `ZopeVersionControl`. The main difference from default uses of `CMFStaging` is that instead of working with documents individually, we work with the stage as a whole.

In addition to that, we can do `incremental` and `full` deployments.

Also, instead of exposing stages in the way CMFStaging does, we hide them and handle them internally.

The `source stage` is a specialized folderish type called `Staging Area`. A Staging Area has a nice and very informative default page that lists the last modifications, a history of deployments and controls for full/incremental deployment.

Each `source stage` maps to it's own `destination stage`. A `destination stage` can be any container in a Plone site. The destinations can be nested and extra care is taken not to step on nested stage destinations.

Incremental deployments keep information internally in a diff-like fashion, listing the added/removed/updated content objects.

There is also a minimal support for `events` built into Enfold Simple Staging, and that's where the next product comes in.

7.2 StageDeployment

`StageDeployment` basically subscribes to the `deployment` events fired by the `EnSimpleStaging` product to be notified about when a deployment happens.

Once they receive such a notification, they extract the information about added/removed/updated content objects and recursively build a structure with information about those objects as expected by EnTransit.

The next step is then to connect to the `EnTransit` server and deploy those entries.

The trick here is that as any exception that happens at the `EnTransit` server is propagated back to the client, we just let the exception received pop out and abort the Zope transaction. This way the operations both on the Zope server and EnTransit result either in a `commit` or `abort` in **both** sides.

8 Customizing data collection

Creating a custom adapter for collecting information specific to your own custom content type is very simple stuff. This is really the minimal stuff you need to know about Five/Zope 3 to use the system and gives you a lot of flexibility for very little cost.

Suppose you have your very own content object defined in your product. The class is defined in `MyProduct/content/article.py`.

8.1 Creating an Interface

Now, supposing you don't have any Zope3-style interface defined in your product, let's define one. Create a file at `MyProduct/interface.py` and open it in your editor of preference. Now type in the following:

```
from zope.interface import Interface

class IArticle(Interface):
  """A Interface for Article
  """
```

That's about it. Now you have a very simple Zope 3 interface.

8.2 Creating an Adapter

The next step is to create the adapter that will provide the functionality needed by `StageDeployment`. An adapter is nothing too fancy. It's basically a class with a well-defined constructor that happens to implement a specific interface.

For our example, we want to implement the `IInformationExtractor` interface that is defined in `StageDeployment/interface.py`. This interface has only one method named `extract` that returns a dictionary mapping attributes to values for the specific content object being deployed.

Here's a listing for the interface definition:

```
class IInformationExtractor(Interface):

    def extract():
```

```
        """Return a simple data structure containing information
        suitable for deployment about the adapted object.
        """
```

Now, to create your adapter, create a file at `MyProduct/adapter.py` and type the following into that file, assuming your `Article` content has methods named `Title` and `getBody` to return the title and body attributes of your content respectively:

```
class ArticleExtractor:

  def __init__(self, context):
      self.context = context

  def extract(self):
      return {'title': self.context.Title(),
              'body': self.context.getBody()
             }
```

Great, so that's your first Zope 3 adapter! The only thing `special` about it is the constructor (the `__init__` method) which takes a single argument named `context`, which is the target object being `adapted` to the requested interface.

8.3 Glueing it all together

Now, for the last part, glueing this together. This step is reasonably simple and consists of creating a configuration file using the `ZCML` language, which is a XML-based configuration language specifically designed for Zope 3.

Create a file under `MyProduct/configure.zcml` with the following content:

```
<configure xmlns="http://namespaces.zope.org/zope"
           xmlns:five="http://namespaces.zope.org/five">

  <adapter for=".interface.IArticle"
     provides="Products.StageDeployment.interface.IInformationExtractor"
     factory=".adapter.ArticleExtractor"
     />

<five:implements
    class=".content.article.Article"
    interface=".interface.IArticle"
    />

</configure>
```

Again, nothing too fancy here. But let's dissect it line by line.

On the first line, we see the tag `<configure>`. This is required and means to Zope 3 that you are starting a configuration statement. In the same tag you see a `xmlns` attribute. That attribute is declaring that any tag inside this block is in the namespace pointed by the tag value. There's also a `xmlns:five` attribute. This is declaring that tags prefixed by `five:` are on the namespace pointed by the tag value.

Next, the `<adapter>` tag. This tag tells Zope 3 you are registering a adapter. It has 3 tags, `for`, `provides` and `factory`.

for This adapted will be registered for objects declaring to implement the interface listed here

provides The interface that the adapter declares to implement

factory The location of the factory to be used for instantiating the adapter

And finally the `five:implements` tag, which has two attributes, `class` and `interface`.

class The class that you are declaring to implement the interface

interface The interface declared to be implemented

8.4 Reviewing prevous steps

Ok, that's it! What you have accomplished here:

1. You created a `marker interface`, that is a `blank` interface that doesn't have any attributes or methods or fields.
2. You created a adapter.
3. You bound the `marker interface` to your content class using the `five:implements` directive.
4. You registered a adapter from `IArticle` to `IInformationExtractor` using the `<adapter>>` directive.

Notice that your product **does not** depend on Five or Zope 3 so far. As long as you do not import the `adapter.py` or `interface.py` modules from your product anywhere your product should still work on a non-Five enabled environment.

As soon as you drop your product on a Five-enabled environment though, Five will find your `configure.zcml` file and execute it, effectively registering the adapter and declaring your class to implement the `IArticle` interface.

8.5 The whole picture

Now, back to `StageDeployment`, what happens during deployment if it finds a instance of your `Article` content is:

1. `StageDeployment` asks for a adapter to `IInformationExtractor`
2. Zope 3 will notice you implement the `IArticle` interface and will lookup a adapter from `IArticle` to `IInformationExtractor`
3. It will find the factory you registered, which is the `ArticleExtractor` class in the `adapter.py` module.
4. And create an instance of `ArticleExtractor` passing your `Article` content instance to the constructor
5. `StageDeployment` finally calls the `extract` method of your `ArticleExtractor` class and receives a dictionary instance with `title` and `body` as elements.
6. `StageDeployment` pushes that piece of data to the `EnTransit` server along with any other entry that was deployed