Templer System Manual Documentation

Release 1.0

Cris Ewing and contributors

January 23, 2014

Using ZopeSkel

Templer is derived from an earlier system – ZopeSkel – which was designed specifically for the needs of the Zope and Plone communities.
If you have worked with ZopeSkel in the past, you may continue to do so in the same way you always have. You will find that the selection of templates is a bit different, but the system works exactly as before.
For more information on using ZopeSkel, see *ZopeSkel* in this manual.

This manual covers the templer code generation system. Templer is a general-purpose system for generating code skeletons for any Python project from pre-defined templates. Through an interactive interface, the user provides information which is used to generate a skeleton of files and folders that fits their individual needs.

To get started quickly, see *Using Templer*.

Templer consists of a number of packages, each of which provides a set of templates. Install the package that you need for the templates you want.

- Install *templer.core* to build basic Python namespace and nested namespace packages
- Install *templer.buildout* to build basic buildouts and recipes to extend the zc.buildout system
- Install *templer.zope* to build basic namespace and nested namespace packages for Zope
- Install *templer.plone* to build software projects for the Plone Content Management System
- Install *templer.plone[localcommands]* to get access to local commands for adding features to your Plone software projects

If you are interested in extending the templer system with your own templates, read the developers manual.

CHAPTER 1

Contents

1.1 Using Templer

The fastest way to get started with templer is to install it using virtualenv.

```
$ easy_install virtualenv # if you have not already installed this
$ virtualenv --distribute templerenv
$ source templerenv/bin/activate
(templerenv)$ easy_install templer.core
```

This installs the core of the templer system. See the list of *available packages* to determine which templer package best suits your needs.

You will now find `templer` and `paster` commands in your virtualenv. Use the templer command to create a basic Python namespace package.

```
(templerenv)$ templer basic_namespace my.package
```

After answering a few questions, you will have your new package.

1.2 Templer Packages

The Templer system is made up of a number of small, distinct packages. Read the list of packages below to discover which templates are provided in each. Install the package which provides the functionality you need.

1.2.1 Package Listing

templer.core This package installs the core of the Templer system. It provides the base template and command classes, an extended system of vars which provide inline help and validation of template questions, and the core of the **structures** functionality.

This package provides:

- the `basic_namespace` and `nested_namespace` package templates
- the `egg_docs` structure and structures for each of the supported license options
- the `templer` script, the command-line tool through which users may generate skeleton packages

templer.localcommands This packages installs the Templer local command and local template. Any package which provides local commands **must** depend on this package.

This package provides the `add` paster local command.

templer.buildout The package installs functionality related to the zc.buildout system.

The package provides:

- the `basic_buildout` and `recipe` templates
- the `bootstrap` structure, used by any template which provides a buildout as part of its output

templer.zope This package installs functionality related to Zope.

This package provides:

- the `zope2_basic` and `zope2_nested` templates

Skeletons generated by these templates will include a buildout. The buildout will create a Zope 2 instance and include the generated package in that instance.

templer.plone This package installs functionality related to Plone.

This package provides:

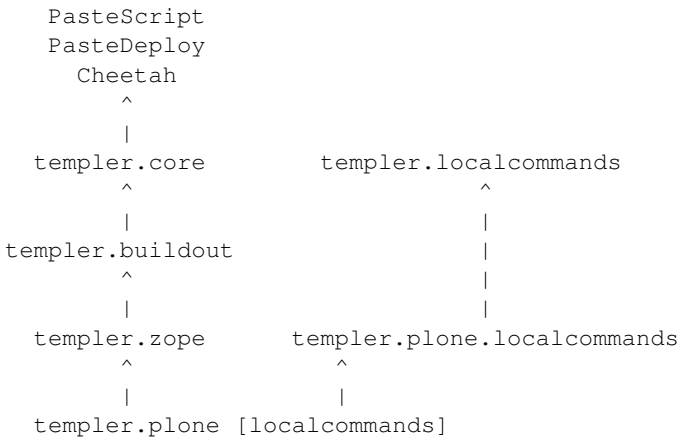
- the `plone_basic`, `plone_nested` and `archetype` templates
- the `namespace_profile` and `nested_namespace_profile` structures, which supply default GenericSetup profiles within a generated skeleton

Skeletons generated by these templates will include a buildout. The buildout will create a Zope 2 instance with Plone installed. The generated package will be included in the instance, and may be available for activation in a Plone site. Skeletons will also include an operational test harness with one or more pre-written tests. The package may be installed with the `[localcommands]` extra, in which case it will depend on templer.plone.localcommands and will have *local commands* available for generated skeletons.

templer.plone.localcommands This package provides *local commands* for templates from the templer.plone package. Provided commands include `browserview`, `browserlayer`, `at_contenttype` and `at_schema_field`.

1.2.2 Package Dependencies

This diagram shows the dependency tree for existing Templer packages. Installing any Templer package will also install all of its dependencies. The base system provided by `templer.core` depends on PasteScript, PasteDeploy and Cheetah.



1.3 Templer Developer's Manual

The templer developer's manual will provide the information a developer needs to be able to create new templates to address their own particular needs.

The following are topics this manual will address:

- Structure of a templer plugin
- Making a new template
  - Am I subclassing something to do so?
  - What classes are available?
  - Why would I choose one over another?
  - How do I make a template?
- How do I ask questions of the end-user?
  - What types of questions are available?
- How do I insert responses into the template?
- What is a template?
- What is a structure?
- Why/when would I use a template vs. a structure?
- pre and post functions, how are they used? for what?
- if else in template vs pre command
- entry points
- category in your new templer template class
- how add egg to buildout/virtual env

1.4 Templer Information

1.4.1 Definition of Terms

local command A Paste class which provides additional functionality *within* a generated skeleton.

package A Python package.

skeleton The generated files and folders resulting from a user running a template.

structure 1) A Python class which controls the generation of a tree of files and folders.

2) A unit of folders and files provided by the templer system for use in a template or templates. Structures provide shared, static resources that may be used by any package in the templer system.

Structures differ from templates in that they do not provide vars.

template 1) A Python class which controls the generation of a skeleton. Templates contain a list of vars to gather input from a user. Templates are run by executing the `templer` command and providing the template name as an argument: `templer basic_namespace my.package`

2) The files and folders provided by a templer package as content to be generated. The answers provided by a user in response to vars are used to fill placeholders in this content.

var A question to be answered by the user when generating a skeleton from a template. A var provides a description of the information required, help text and validation rules to ensure accurate user input.

1.4.2 The History of Templer

Templer is a system that grew out of the refactoring and improvement of the ZopeSkel code generation tool, used by the Plone and Zope communities for years. However, unlike its predecessor templer is not limited specifically to generating packages for Plone or Zope. With templer you can get a quick start building python-based projects for a number of systems and frameworks.

Templer and ZopeSkel

In the beginning, there was ZopeSkel. First created in 2006, it was used to install and run commands to create new Zope and Plone packages. And it was good. But it was also monolithic and inflexible.

And so, at the No Fun BBQ Sprint in 2009, the decision was made to *split the package up* in order to make it easier to work with only the parts you wanted. At the same time, the sprinters decided that keeping consistency in usage was a very important goal, and so the ZopeSkel package has been preserved. Information about the rationale behind this decision may be read in *Splitting ZopeSkel into Egg Packages*.

Starting with version 3.0, ZopeSkel is simply a thin wrapper around the functionality and templates provided by the templer system. Installing ZopeSkel installs the templer packages needed to reproduce the ZopeSkel experience. All the templates and supporting code are in templer packages and none remains in the ZopeSkel package.

1.4.3 Splitting ZopeSkel into Egg Packages

Author joel@joelburton.com

Date 5 Oct 2009

Background

ZopeSkel is currently a single egg, "ZopeSkel". It contains templates for:

- scripts/utilities that are not template specific
- basic nested Python packages, without any Zope/Plone bits
- Basic Zope product/buildout templates
- Plone product/buildout templates
- Silva buildout template
- Code for the "local commands" system
- Local commands for Plone products

Proposal

We propose to divide ZopeSkel into separate packages & eggs:

`zopeskel.base` Local commands system, scripts/utilities

`zopeskel.zope` ***(will depend on zopeskel.base)*** basic_zope template and Zope-only buildouts

`zopeskel.plone` ***(will depend on zopeskel.zope)*** all plone templates/buildouts and local commands for Plone

zopeskel.silva ***(will depend on zopeskel.zope)*** Silva buildout

Backwards Compatibility

Since there is a great deal of documentation that tells users to "easy_install ZopeSkel", we need to make sure there is still a package called this that provides the assumed components.

Therefore, we will keep a ZopeSkel egg, but have this provide no code/packages– it will only exist so that it has setuptools requires to pull in zopeskel.base, zopeskel.zope, zopeskel.plone, zopeskel.silva. Therefore, people following this documentation will get the "full" ZopeSkel.

Rationale

Curently, ZopeSkel can be a bit of magnet for recipes that may not be widely needed by all members–there are non-Plone users of it that don't want to get all of the Plone recipes, for example. In the future, they would be able to

```
easy_install zopeskel.zope
```

to just get the Base/Zope parts.

With additional adoption of ZopeSkel, we anticipate other communities (Repoze, etc) wishing to add templates, and would prefer to avoid an overly- long list of packages. This is especially important as, at least in the Plone world, ZopeSkel is increasingly used by integrators/non-developers, and a long list of packages unrelated to their needs is confusing.

In addition, this will subtly reinforce to people that there *can be* 3rd party packages that add templates. Larger institutional users of Python/Zope/Plone/Silva may find it beneficial to write their own, customized templates (the author of this document already does, for example); however, that this is possible is slightly obscured by the fact that we ship only one monolithic system with all the recipes in it.

Tasks Needed

1. Change the imports and entry points inside of ZopeSkel to match these new package names; for example, changing "plone.py" to import the BasicZope class as "from zopeskel.zope.basic_zope import BasicZope".
2. Adding imports to zopeskel/__init__.py to import everything into this namespace that was previously there. This will ensure that 3rd party templates that made assumptions like "from zopeskel import basic_zope" will still work.
3. Break packages into separate eggs and check into new repository.
4. Empty ZopeSkel package and add setuptools requires so that this egg now installs all the new eggs.

New Repository

Given that ZopeSkel has a wider audience than just Plone, we don't feel it make sense to move it into the plone repository. However, it also doesn't seem right to leave it in the collective–here, it has become a magnet for individual, not-well-organized changes that run counter to the requirement that it be a stable, best-practice product.

We recommend creating a new repository, "zopeskel", which would contain the zopeskel packages. This would allow us to grant svn access to people without sharing core plone access, and would discourage collective-style drive-by improvements.

1.5 Templer-Based Applications

It is possible, using the templer system and its provided packages, to create an application which addresses a specific set of needs. The following are applications which have been built using the templer system. If you build such a system, please add it here.

1.5.1 ZopeSkel

Installing ZopeSkel

ZopeSkel can be installed in one of two ways: with buildout or with virtualenv.

Note: Despite existing documentation to the contrary, it is not recommended to install ZopeSkel in your system python.

Buildout installation

Add to your `buildout.cfg`:

```
parts =
   ...
   zopeskel

[zopeskel]
# installs paster and Zopeskel
recipe = zc.recipe.egg
eggs =
   PasteScript
   ZopeSkel
```

After re-running buildout, you will have `zopeskel` and `paster` commands in the `bin` directory of your buildout.

Virtualenv installation

A Warning About Pip

Pip is a popular packaging tool for Python. It has not always properly supported the installation of setuptools `extras`. If you use pip to install ZopeSkel, you will need at least version 1.1.
If you have questions, see the discussion in this ZopeSkel issue report

First, install virtualenv into your system:

```
$ easy_install virtualenv
```

Next, create a virtual environment with the new `virtualenv` command:

```
$ virtualenv --distribute zopeskelenv
```

Once virtualenv has finished creating your new virtual environment, you can install zopeskel to your new virtual environment by:

```
$ zopeskelenv/bin/easy_install zopeskel
```

Once this is complete, you will have `zopeskel` and `paster` commands in the `bin` directory of your virtualenv.

Basic Usage

ZopeSkel is used to create empty projects for Zope and Plone. A number of templates are included with ZopeSkel:

- `basic_namespace`
- `nested_namespace`
- `basic_buildout`
- `recipe`
- `zope2_basic`
- `zope2_nested`
- `plone_basic`
- `plone_nested`
- `archetype`

The most basic template for Plone is plone_basic, which creates an empty Plone add-on. Optionally you may add a GenericSetup profile to make your add-on appear in the list of available add-ons in Plone's *Site Setup*. In this case a profiles/default directory will be created in your new add-on.

For example:

```
$ ./bin/zopeskel plone_basic my.example
```

This template asks you a series of questions and creates a new add-on *package* from your answers. When prompted to choose a mode, unless you know what you are doing, select easy mode (it is the default). You will see output like the following:

```
plone_basic: A package for Plone add-ons

This template creates a package for a basic Plone add-on project with
a single namespace (like Products.PloneFormGen).

To create a Plone project with a name like 'collective.geo.bundle'
(2 dots, a 'nested namespace'), use the 'plone_nested' template.

This template supports local commands.  These commands allow you to
add Plone features to your new package.

If you are trying to create a Plone *site* then the best place to
start is with one of the Plone installers.  If you want to build
your own Plone buildout, use one of the plone'N'_buildout templates


If at any point, you need additional help for a question, you can enter
'?' and press RETURN.

Expert Mode? (What question mode would you like? (easy/expert/all)?) ['easy']: easy

Version (Version number for project) ['1.0']: 1.0
Description (One-line description of the project) ['']: This is an example product built with ZopeSke
```

```
Register Profile (Should this package register a GS Profile) [False]: True
Creating directory ./my.example
Replace 1079 bytes with 1273 bytes (1/43 lines changed; 5 lines added)
Replace 42 bytes with 119 bytes (1/1 lines changed; 4 lines added)
-------------------------------------------------------------------------------
The project you just created has local commands. These can be used from within
the product.

usage: paster COMMAND

Commands:
  add  Allows the addition of further templates to an existing package

For more information: paster help COMMAND
-------------------------------------------------------------------------------

**********************************************************************
**  Your new package supports local commands.  To access them, change
**  directories into the 'src' directory inside your new package.
**  From there, you will be able to run the command `paster add
**  --list` to see the local commands available for this package.
**********************************************************************
```

Once complete you will have a brand new Plone package waiting for customization!

Local Commands

A *local command* uses templates to allow you to add features to your newly created add-on. To run a local command, you must first change directory to inside your add-on:

```
$ cd my.example/src
```

From here, you can use the `paster` command to show you which templates are available to use:

```
$ ../../bin/paster add --list
Available templates:
    browserlayer:  A Plone browserlayer
    browserview:   A browser view skeleton
```

To run a specific local command, you provide the name of the template:

```
$ ../../bin/paster add browserview
Enter view_name (Browser view name) ['Example']: Example
```

When this command completes, you will find a new browser module, with the files required to add a browser view to your add-on:

```
$ ls -1 my/example/browser/
__init__.py
configure.zcml
exampleview.pt
exampleview.py
```

Local Commands and Python Paste

Implementation details of local commands mean that any package which supports them will have a direct dependency on Paste, PasteScript and PasteDeploy. As a result, when you first create a package with available local commands,

you will find that these three packages have automatically been installed *inside* your package structure:

```
$ cd ../
$ ls -1
CHANGES.txt
CONTRIBUTORS.txt
Paste-1.7.5.1-py2.6.egg
PasteDeploy-1.5.0-py2.6.egg
PasteScript-1.7.5-py2.6.egg
README.txt
...
```

This is an unfortunate but unavoidable situation so long as local commands are desired. There are a few things you should keep in mind when working with packages that provide local commands:

- Paste, PasteScript and PasteDeploy should **never** be placed under version control.
- Any time you check out the package and include it in a buildout, they will reappear.
- When you are finished with using local commands, you can get rid of these extra packages for good by disabling local commands.

Disabling Local Commands

Local commands are useful for extending a package skeleton when you are first setting up a new project. Once you've completed setup, however, it is a good idea to disable local commands so that you will no longer be bothered by the presence of extra package eggs in your source code tree.

To disable local commands, and stop Paste, PasteScript and PasteDeploy from appearing when you work with your egg, you can edit the source code generated by ZopeSkel. First, you will want to find and remove the following lines from your package `setup.py` file:

```
setup_requires=["PasteScript"],
paster_plugins=["templer.localcommands"],
```

Additionally, you may remove the following from `setup.cfg` in your package root directory:

```
[templer.local]
template = plone_basic # note that the name found here may differ
```

After removing these lines, your package will no longer have local commands available. Furthermore, when you check it out of source control and include it in a buildout, you will no longer find Paste, PasteScript or PasteDeploy in your package source tree.

CHAPTER 2

Indices and tables

- *genindex*
- *modindex*
- *search*