# Buildout

*Release 1.2.1*

Jim Fulton

Nov 20, 2017

# Contents

# CHAPTER 1

# Screencasts

## 1.1 A Brief Introduction to Buildout

by Brandon Craig Rhodes

## 1.2 Using buildout to install Zope and Plone

by The WebLion Team

# CHAPTER 2

# Talks

## 2.1 How AlterWay releases web applications using zc.buildout

by Tarek Ziadé from PyCON 2009

## 2.2 Buildout for pure Python projects

by Carsten Rebbien from EuroPython 2008

## 2.3 Buildout lightning talk

by Philipp von Weitershausen from EuroPython 2007

## 2.4 Delivering egg-based applications with zc.buildout, using a distributed model

by Tarek Ziadé

# CHAPTER 3

# Slides

## 3.1 Django & Buildout

by Horst Gutmann

## 3.2 Delivering applications with zc.buildout and a distributed model

by Tarek Ziade

## 3.3 Buildout and Plone

by Tim Knapp

## 3.4 Python eggs, zc.buildout, zopeproject and zope3

by Clayton Parker

## 3.5 Reaching Deployment Nirvana Using Buildout

CHAPTER 4

# What people says about Buildout ?

- Buildout is an exceedingly civilized way to develop an app. *–Jacob Kaplan-Moss, creator of Django*.
- "While not directly aiming to solve world peace, it perhaps will play a role in the future, as people will be less angry about application deployment and will have more time for making love and music." *–Noah Gift, co-author of 'Python For Unix and Linux' from O'Reilly*.
- "Even if you are not planning on writing any custom code, the buildout approach is an easy way to install Plone in a robust, well-tested manner." *–Martin Aspeli, Plone core developer*.
- "Buildout was a natural choice for a build and deployment management tool." *–Rob Miller*
- "Buildout is no longer just for developers. Get your products ready!" *–Steve McMahon*
- "I really think Buildout is the only sane way to develop anything serious using python these days" *–Ignas Mikalajunas*

CHAPTER 5

# Overview of the Installation Process

The zc.buildout software is very easy to install with only one dependency, on the *setuptools* package that provides manipulation facilities for Python eggs.

1. Many **existing projects** are already based on *zc.buildout* and include within their project files the necessary `bootstrap.py` file. To activate use of *zc.buildout* within such a project, simply run that `bootstrap.py` using the specific Python interpreter for the project. This may be the system Python or in the case of a virtualenv sandbox, the Python within the `bin/` subdirectory of the project.

   ```
   $ cd projectdir
   $ bin/python bootstrap.py
   ```

**Don't have the pip command?**

You can install the **pip** command by following the instructions on the pip installation page.

2. While *zc.buildout* is most often installed within each project directory, it can also be **installed system-wide**, to make it easy to create new projects.

   ```
   $ pip install zc.buildout
   ```

   This gives you a new command named **buildout** to use in initializing or updating a project.

**For an even more isolated build environment...**

To use an isolated instance of Python within the project, the following commands will create a new sandbox and establish use of *zc.buildout* within it.

```
$ virtualenv --no-site-packages newproject
$ cd newproject
$ bin/pip install zc.buildout
$ bin/buildout init
```

3. To **add zc.buildout to a new project**, the primary step is to execute the **`buildout init`** command while your current directory is set to the root of the project directory. This command will create all necessary files/directories, including a minimal `buildout.cfg` file to control buildout.

   ```
   $ cd newproject
   $ buildout init
   ```

   This command sequence will use the system Python for the project. If you have some other project set up that uses *zc.buildout* you can borrow its **`buildout`** command to initialize your new project.

   ```
   $ cd newproject
   $ /oldproject/bin/buildout init
   ```

   Unfortunately this sequence of commands will not provide a `bootstrap.py` command for others to use to initialize *zc.buildout* when they receive a copy of your project. Therefore it is recommended that you download and **incorporate a copy of** `bootstrap.py` **within your project fileset**.

   ```
   $ cd newproject
   $ wget -O bootstrap.py https://bootstrap.pypa.io/bootstrap-buildout.py
   ```

---

A good next step in understanding *zc.buildout* is *Directory Structure of a Buildout* which also covers which file/directories should be under version control and which should not.

CHAPTER 6

# Directory Structure of a Buildout

**Layout of a Buildout**

```
project/
   bootstrap.py  †
   buildout.cfg  †
   .installed.cfg
   parts/
   develop-eggs/
   bin/
       buildout
       mypython
   eggs/
   downloads/

† put red items under version control
```

## bootstrap.py

A script to be included with each project to help a new developer set up the tree for use with *zc.buildout* after checking out the project. It installs from the network the *zc.buildout* and *setuptools* packages into the project directory.

The actual URL for fetching the `bootstrap.py` file is:

> https://bootstrap.pypa.io/bootstrap-buildout.py

## buildout.cfg

Contains the default build specification for the entire project but others can be defined such as `deployment.cfg` and `production.cfg`. Specification files can include other specification files.

### .installed.cfg

A hidden file that represents the current build state of the project. The *zc.buildout* software updates it as parts are installed or removed. The file is used by *zc.buildout* to compute the minimum set of changes to bring the project into sync with the `buildout.cfg` or other specification file.

### parts/

Each part may create a holding directory underneath `parts/` for the specific use of the part's recipe. The part directory belongs to the recipe responsible for installing/uninstalling the part and is not intended for modification by the developer.

### develop-eggs/

The directory holds a kind of symlink or shortcut link to the development directories elsewhere on the system of distributions being worked on. The content of the directory is manipulated by *zc.buildout* in response to "develop = DIRS" entries in the build specification file.

### bin/

The directory receives executable scripts that *zc.buildout* creates from entrypoints defined within eggs and called out in the build specification.

### bin/buildout

The command to invoked *zc.buildout* for bringing the state of the project directory into sync with a particular build specification.

### bin/mypython

Just an example of a specific instance of a Python interpreter that encompasses a specific set of parts. The name is arbitrary and there can be any number of such custom interpreters.

### eggs/

A cache directory of eggs pulled down from the net, ready for mapping onto the *sys.path* of specific parts, as given in the build specification. This directory may be shared across projects if configured to be in a common nnlocation, increasing the speed with which buildouts can be constructed.

### downloads/

A cache directory of raw files pulled down from the net, such as compressed eggs, zipfiles, etc. After being downloaded into this directory, the contents are usually unpacked under a part name under the `parts/`. This directory may also be shared across projects, for greater efficiency.

### lib/

Not strictly part of *zc.buildout* this directory appears when running within a sandbox created by virtualenv. It represents a standard Python lib/ hierarchy underwhich anything installed is outside the control of *zc.buildout*. It generally should be left alone.

### build/

Not strictly part of *zc.buildout* either, it is a scratch directory used by distutils/setuptools in the process of constructing eggs.

### dist/

Not strictly part of *zc.buildout*, this directory receives the final packed representations of distributions such as eggs ready for uploading or sharing.

Here's a list of files and directories that you can ignore in your version control system software:

```
.installed.cfg
parts
develop-eggs
bin
eggs
downloads
lib
build
dist
```

---

Now you are familiar with installation and directory structure of a typical buildout project. Next step is to learn using buildout to develop a package .

CHAPTER 7

# Use Case - A Single Module

This is a simple use of *zc.buildout* to manage a project that is distributed as a single module. Here are the commands one might use to obtain a copy of the source from the Cheeseshop for examination:

```
$ easy_install --editable --build-directory . xanalogica.tumbler
$ cd xanalogica.tumbler
```

If we were going to develop with the project we might instead check out a copy from version control:

```
$ svn co http://svn.taupro.com/xanalogica.tumbler/trunk/ xanalogica.tumbler
```

After obtaining a copy in either manner, the following commands would bootstrap the buildout environment and build it up-to-date, satisfying any additional dependencies given in the module's `setup.py`.

```
$ python bootstrap.py
$ bin/buildout
```

**Single-Module buildout.cfg**

```
[buildout]
develop = .
parts =
  xprompt
  test

[xprompt]
recipe = zc.recipe.egg:scripts
eggs = xanalogica.tumbler
interpreter = xprompt

[test]
recipe = zc.recipe.testrunner
eggs = xanalogica.tumbler
```

Examining the buildout specification, we see that it consists of two parts, one named “test” and one named “xprompt”. The list of parts, in the order they are built, is given in the “parts = ” line. The “[buildout]” section is the top-most section and the only one that is required. Entries that control the global operation of *zc.buildout* go here.

Every part uses a recipe to oversee its installation/uninstallation and the name of a part is arbitrary. In some cases a recipe will create files using or prefixed with the part name. All other “name = value” lines underneath a part section are simple arguments that are passed to the recipe at build time.

For the “xprompt” part, we simply specify that it requires a single egg, named “xanalogica.tumbler”. We also say using the “interpreter =” line that we want a Python interpreter named “xprompt” that maps only that egg onto *sys.path*. In this manner we can exercise the logic of the egg interactively.

For the “test” part, the xanalogica.tumbler egg has provided unit tests and we want access to a script that will invoke them. The “zc.recipe.testrunner” recipe provides this and generates such a script under the bin/ directory named after the name or in the case “test”. To run the unit tests:

```
$ bin/test
```

Both parts reference the xanalogica.tumbler egg and ordinarily *zc.buildout* would fetch an egg from the Cheeseshop by that name. However, the “develop =” line tells *zc.buildout* to add the egg defined by the `setup.py` in the current directory to the list of candidates. Since *zc.buildout* prefers eggs under development over finished eggs in the Cheeseshop, this means it will use our local module to satisfy the search for “xanalogica.tumbler”.

---

Now you are familiar with installation, directory structure of a typical buildout project. Also you understood the basic usage of Buildout from here. Now you can go to the more detailed documentation.

# CHAPTER 8

# Buildout Documentation

## 8.1 Getting Started

- Overview of the Installation Process
- Directory Structure of a Buildout
- Use Case - A Single Module

## 8.2 Tutorials

The tutorial goes through all the features of Buildout and its usage. This is a slightly modified version of Jim Fulton's original presentation hand-out.

Developing Django apps with zc.buildout by Jacob Kaplan-Moss is a good introductory tutorial to learn about Buildout.

## 8.3 Creating Recipes

Creating Recipes describes in depth about creating recipes.

## 8.4 List of Recipes

A list of the more commonly used recipes used with Buildout.

## 8.5 Links

A collection of links related to Buildout.

# CHAPTER 9

# Buildout Tutorial

Jim Fulton, Zope Corporation

DZUG 2007

## 9.1 What is zc.buildout?

- Coarse-grained python-based configuration-driven build tool
- Tool for working with eggs
- Repeatable

  It should be possible to check-in a buildout specification and reproduce the same software later by checking out the specification and rebuilding.

- Developer oriented

## 9.2 Coarse-grained building

- make and scons (and distutils) are fine grained
  - Focus on individual files
  - Good when one file is computed from another

    .c -> .o -> .so

  - rule-driven
  - dependency and change driven
- zc.buildout is coarse-grained
  - Build large components of a system

        - applications
        - configurations files
        - databases
    - configuration driven

## 9.3 Python-based

- make is an awful scripting language
    - uses shell
    - non-portable
- Python is a good scripting language

    Fortunately, distutils addresses most of my building needs. If I had to write my own fine-grained build definition, I'd use scons.

## 9.4 Working with eggs

- Eggs rock!
- easy_install
    - Easy!
    - Installs into system Python
    - Not much control
- workingenv makes easy_install much more usable
    - Avoids installing into system Python
    - Avoids conflicts with packages installed in site_packages
    - Really nice for experimentation
    - Easy!
    - Not much control

## 9.5 `zc.buildout` and eggs

- Control
    - Configuration driven
        - easier to control versions used
        - always look for most recent versions by default

            When upgrading a distribution, `easy_install` doesn't upgrade dependencies,

        - support for custom build options
- Greater emphasis on develop eggs

- Automates install/uninstall
- preference to develop eggs

  I often switch between develop and non-develop eggs. I may be using a regular egg and realize I need to fix it. I checkout the egg's project into my buildout and tell buildout to treat it as a develop egg. It creates the egg link in develop eggs and will load the develop egg in preference to the non-develop egg.

  (`easy_install` gives preference to released eggs of the same version.)

  When I'm done making my change, I make a new egg release and tell buildout to stop using a develop egg.

`zc.buildout` is built on setuptools and `easy_install`.

## 9.6 zc.buildout current status

- Actively used for development and deployment
- Third-generation of ZC buildout tools

  Our earliest buildouts used make. These were difficult to maintain and reuse.

  Two years ago, we created a prototype Python-based buildout system.

  `zc.buildout` is a non-prototype system that reflects experience using the prototype.

- A number of "recipes" available

## 9.7 A Python Egg Primer

Eggs are simple!

- directories to be added to path
  - may be zipped
  - "zero" installation
- Meta data
  - dependencies
  - entry points
- May be distributed as source distributions

  `easy_install` and `zc.buildout` can install source distributions as easily as installing eggs. I've found that source distributions are more convenient to distribute in a lot of ways.

- Automatic discovery through PyPI

## 9.8 Egg jargon

- Distribution

"distribution" is the name distutils uses for something that can be distributed. There are several kinds of distributions that can be created by distutils, including source distributions, binary distributions, eggs, etc.

- source and binary distributions

  A source distribution contains the source for a project.

  A binary distributions contains a compiled version of a project, including .pyc files and built extension modules.

  Eggs are a type of binary distribution.

- Platform independent and platform dependent eggs

  Platform dependent eggs contain built extension modules and are thus tied to a specific operating system. In addition, they may depend on build options that aren't reflected in the egg name.

- develop egg links

  Develop egg links (aka develop eggs) are special files that allow a source directory to be treated as an egg. An egg links is a file containing the path of a source directory.

- requirements

  Requirements are strings that name distributions. They consist of a project name, optional version specifiers, and optional extras specifiers. Extras are names of features of a package that may have special dependencies.

- index and link servers

  `easy_install` and `zc.buildout` will automatically download distributions from the Internet. When looking for distributions, they will look on zero or more links servers for links to distributions.

  They will also look on a single index server, typically (always) http://www.python.org/pypi. Index servers are required to provide a specific web interface.

## 9.9 Entry points

- Very similar to utilities
  - Named entry point groups define entry point types
  - Named entry points within groups provide named components of a given type.
- Allow automated script generation

  Wrapper script:

  - Sets up path

    `easy_install` and `zc.buildout` take very different approaches to this.

    `easy_install` generates scripts that call an API that loads eggs dynamically at run time.

    `zc.buildout` determines the needed eggs at build time and generates code in scripts to explicitly add the eggs to `sys.path`.

    The approach taken by `zc,buildout` is intended to make script execution deterministic and less susceptible to accidental upgrades.

  - Imports entry point
  - Calls entry point without arguments

Buildout allows more control over script generation. Initialization code and entry point arguments can be specified.

## 9.10 Buildout overview

- Configuration driven
  - ConfigParser +

    Buildout uses the raw ConfigParser format extended with a variable-substitution syntax that allows reference to variables by section and option:

    ```
    ${sectionname:optionname}
    ```
  - Allows full system to be defined with a single file

    Although it is possible and common to factor into multiple files.
- Specify a set of "parts"
  - recipe
  - configuration data

    Each part is defined by a recipe, which is Python software for installing or uninstalling the part, and data used by the recipe.
- Install and uninstall

  If a part is removed from a specification, it is uninstalled.

  If a part's recipe or configuration changes, it is uninstalled and reinstalled.

## 9.11 Buildout overview (continued)

- Recipes
  - Written in python
  - Distributed as eggs
- Egg support
  - Develop eggs
  - Egg-support recipes

## 9.12 Quick intro

- Most common case
  - Working on a package
  - Want to run tests
  - Want to generate distributions
- buildout is source project

- Example: `zope.event`

## 9.13 `zope.event` project files

- source in `src` directory

  Placing source in a separate `src` directory is a common convention. It violates "shallow is better than nested". Smaller projects may benefit from putting sources in the root directory,

- `setup.py` for defining egg

  Assuming that the project will eventually produce an egg, we have a setup file for the project. As we'll see later, this can be very minimal to start.

- `README.txt`

  It is conventional to put a README.txt in the root of the project. distutils used to complain if this wasn't available.

- `bootstrap.py` for bootstrapping buildout

  The bootstrap script makes it easy to install the buildout software. We'll see another way to do this later.

- `buildout.cfg` defines the buildout

## 9.14 zope.event buildout.cfg

```
[buildout]
parts = test
develop = .

[test]
recipe = zc.recipe.testrunner
eggs = zope.event
```

Let's go through this line by line.

```
[buildout]
```

defines the buildout section. It is the only required section in the configuration file. It is options in this section that may cause other sections to be used.

```
parts = test
```

Every buildout is required to specify a list of parts, although the parts list is allowed to be empty. The parts list specifies what to build. If any of the parts listed depend on other parts, then the other parts will be built too.

```
develop = .
```

The develop option is used to specify one or more directories from which to create develop eggs. Here we specify the current directory. Each of these directories must have a setup file.

```
[test]
```

The `test` section is used to define our test part.

```
recipe = zc.recipe.testrunner
```

Every part definition is required to specify a recipe. The recipe contains the Python code with the logic to install the part. A recipe specification is a distribution requirement. The requirement may be followed by an colon and a recipe name. Recipe eggs can contain multiple recipes and can also define an default recipe.

The `zc.recipe.testrunner` egg defines a default recipe that creates a test runner using the `zope.testing.testrunner` framework.

```
eggs = zope.event
```

The zc.recipe.testrunnner recipe has an eggs option for specifying which eggs should be tested. The generated test script will load these eggs along with their dependencies.

For more information on the `zc.recipe.testrunner` recipe, see http://www.python.org/pypi/zc.recipe.testrunner.

## 9.15 Buildout steps

- Bootstrap the buildout:

  ```
  python bootstrap.py


     This installs setuptools and zc.buildout locally in your
     buildout. This avoids changing your system Python.
  ```

- Run the buildout:

  ```
  bin/buildout


     This generates the test script, ``bin/test``.
  ```

- Run the tests:

  ```
  bin/test
  ```

- Generate a distribution:

  ```
  bin/buildout setup . sdist register upload
  bin/buildout setup . bdist_egg register upload
  ```

  ```
  bin/buildout setup . egg_info -rbdev sdist register upload
  ```

Buildout accepts a number of commands, one of which is `setup`. The `setup` command takes a directory name and runs the setup script found there. It arranges for setuptools to be imported before the script runs. This causes setuptools defined commands to work even for distributions that don't use setuptools.

The sdist, register, upload, bdist_egg, and egg_info commands are setuptools and distutils defined commands.

The sdist command causes a source distribution to be created.

The register command causes a release to be registered with PyPI and the upload command uploads the generated distribution. You'll need to have an account on PyPI for this to work, but these commands will actually help you set an account up.

The bdist_egg command generates an egg.

The egg_info command allows control of egg meta-data. The -r option to the egg_info command causes the distribution to have a version number that includes the subversion revision number of the project. The -b option specified a revision tag. Here we specified a revision tag of "dev", which marks the release as a development release. These are useful when making development releases.

## 9.16 Exercise 1

> We won't have time to stop the lecture while you do the exercises. If you can play and listen at the same time, then feel free to work on them while I speak. Otherwise, I recommend doing them later in the week. Feel free to ask me questions if you run into problems.

Try building out `zope.event`.

- Check out: https://github.com/zopefoundation/zope.event
- Bootstrap
- Run the buildout
- Run the tests
- Look around the buildout to see how things are laid out.
- Look at the scripts in the bin directory.

## 9.17 buildout layout

- `bin` directory for generated scripts
- `parts` directory for generated part data

  Many parts don't use this.
- `eggs` directory for (most) installed eggs
  - May be shared across buildouts.
- `develop-eggs` directory
  - develop egg links
  - custom eggs
- `.installed.cfg` records what has been installed

  > Some people find the buildout layout surprising, as it isn't similar to a Unix directory layout. The buildout layout was guided by "shallow is better than nested".
  >
  > If you prefer a different layout, you can specify a different layout using buildout options. You can set these options globally so that all of your buildouts have the same layout.

## 9.18 Common buildout use cases

- Working on a single package

  > zope.event is an example of this use case.

- System assembly
- Try out new packages
  - workingenv usually better
  - buildout better when custom build options needed
- Installing egg-based scripts for personal use

  `~/bin` directory is a buildout

## 9.19 Creating eggs

Three levels of egg development

- Develop eggs, a minimal starting point
- Adding data needed for distribution
- Polished distributions

## 9.20 A Minimal/Develop `setup.py`

```
from setuptools import setup
setup(
    name='foo',
    package_dir = {'':'src'},
    )
```

If we're only going to use a package as a develop egg, we just need to specify the project name, and, if there is a separate source directory, then we need to specify that location.

We'd also need to specify entry points if we had any. We'll see an example of that later.

See the setuptools and distutils documentation for more information.

## 9.21 Distributable `setup.py`

```
from setuptools import setup, find_packages
name='zope.event'
setup(
    name=name,
    version='3.3.0',
    url='http://www.python.org/pypi/'+name,
    author='Zope Corporation and Contributors',
    author_email='zope3-dev@zope.org',
    package_dir = {'': 'src'},
    packages=find_packages('src'),
    namespace_packages=['zope',],
    include_package_data = True,
    install_requires=['setuptools'],
    zip_safe = False,
    )
```

If we want to be able to create a distribution, then we need to specify a lot more information.

The options used are documented in either the distutils or setuptools documentation. Most of the options are fairly obvious.

We have to specify the Python packages used. The `find_packages` function can figure this out for us, although it would often be easy to specify it ourselves. For example, we could have specified:

```
packages=['zope', 'zope.event'],
```

The zope package is a namespace package. This means that it exists solely as a container for other packages. It doesn't have any files or modules of it's own. It only contains an __*init*__ module with:

```
pkg_resources.declare_namespace(__name__)
```

or, perhaps:

```
# this is a namespace package
try:
    import pkg_resources
    pkg_resources.declare_namespace(__name__)
except ImportError:
    import pkgutil
    __path__ = pkgutil.extend_path(__path__, __name__)
```

Namespace packages have to be declared, as we've done here.

We always want to include package data.

Because the __*init*__ module uses setuptools, we declare it as a dependency, using `install_requires`.

We always want to specify whether a package is zip safe. A zip safe package doesn't try to access the package as a directory. If in doubt, specify False. If you don't specify anything, setuptools will guess.

## 9.22 Polished `setup.py` (1/3)

```
import os
from setuptools import setup, find_packages

def read(*rnames):
    return open(os.path.join(os.path.dirname(__file__), *rnames)).read()

long_description=(
        read('README.txt')
        + '\n' +
        'Detailed Documentation\n'
        '**********************\n'
        + '\n' +
        read('src', 'zope', 'event', 'README.txt')
        + '\n' +
        'Download\n'
        '**********************\n'
        )

open('documentation.txt', 'w').write(long_description)
```

In the polished version we flesh out the meta data a bit more.

When I create distributions that I consider ready for broader use and upload to PyPI, I like to include the full documentation in the long description so PyPI serves it for me.

## 9.23 Polished `setup.py` (2/3)

```
name='zope.event'
setup(
    name=name,
    version='3.3.0',
    url='http://www.python.org/pypi/'+name,
    license='ZPL 2.1',
    description='Zope Event Publication',
    author='Zope Corporation and Contributors',
    author_email='zope3-dev@zope.org',
    long_description=long_description,

    packages=find_packages('src'),
    package_dir = {'': 'src'},
    namespace_packages=['zope',],
    include_package_data = True,
    install_requires=['setuptools'],
    zip_safe = False,
    )
```

## 9.24 Extras

```
name = 'zope.component'
setup(name=name,
      ...
      namespace_packages=['zope',],
      install_requires=['zope.deprecation', 'zope.interface',
                        'zope.deferredimport', 'zope.event',
                        'setuptools', ],
      extras_require = dict(
          service = ['zope.exceptions'],
          zcml = ['zope.configuration', 'zope.security', 'zope.proxy',
                  'zope.i18nmessageid',
                  ],
          test = ['zope.testing', 'ZODB3',
                  'zope.configuration', 'zope.security', 'zope.proxy',
                  'zope.i18nmessageid',
                  'zope.location', # should be dependency of zope.security
                  ],
          hook = ['zope.hookable'],
          persistentregistry = ['ZODB3'],
          ),
      )
```

Extras provide a way to help manage dependencies.

A common use of extras is to separate test dependencies from normal dependencies. A package may provide other optional features that cause other dependencies. For example, the zcml module in zope.component adds lots of dependencies that we don’t want to impose on people that don’t use it.

## 9.25 `zc.recipe.egg`

Set of recipes for:

- installing eggs
- generating scripts
- custom egg compilation
- custom interpreters

See: http://www.python.org/pypi/zc.recipe.egg.

## 9.26 Installing eggs

```
[buildout]
parts = some-eggs

[some-eggs]
recipe = zc.recipe.egg:eggs
eggs = docutils
       ZODB3 <=3.8
       zope.event
```

The eggs option accepts one or more distribution requirements. Because requirements may contain spaces, each requirement must be on a separate line. We used the eggs option to specify the eggs we want.

Any dependencies of the named eggs will also be installed.

## 9.27 Installing scripts

```
[buildout]
parts = rst2

[rst2]
recipe = zc.recipe.egg:scripts
eggs = zc.rst2
```

If any of the named eggs have `console_script` entry points, then scripts will be generated for the entry points.

If a distribution doesn't use setuptools, it may not declare it's entry points. In that case, you can specify entry points in the recipe data.

## 9.28 Script initialization

```
[buildout]
develop = codeblock
parts = rst2
find-links = http://sourceforge.net/project/showfiles.php?group_id=45693

[rst2]
recipe = zc.recipe.egg
```

```
eggs = zc.rst2
       codeblock
initialization =
    sys.argv[1:1] = (
      's5 '
      '--stylesheet ${buildout:directory}/zope/docutils.css '
      '--theme-url file://${buildout:directory}/zope'
      ).split()
scripts = rst2=s5
```

In this example, we omitted the recipe entry point entry name because the scripts recipe is the default recipe for the zc.recipe.egg egg.

The initialization option lets us specify some Python code to be included.

We can control which scripts get installed and what their names are with the scripts option. In this example, we've used the scripts option to request a script named `s5` from the `rst2` entry point.

## 9.29 Custom interpreters

The script recipe allows an interpreter script to be created.

```
[buildout]
parts = mypy

[mypy]
recipe = zc.recipe.egg:script
eggs = zope.component
interpreter = py
```

This will cause a `bin/py` script to created.

Custom interpreters can be used to get an interactive Python prompt with the specified eggs and and their dependencies on `sys.path`.

You can also use custom interpreters to run scripts, just like you would with the usual Python interpreter. Just call the interpreter with the script path and arguments, if any.

## 9.30 Exercise 2

- Add a part to the `zope.event` project to create a custom interpreter.
- Run the interpreter and verify that you can import zope.event.

## 9.31 Custom egg building

```
[buildout]
parts = spreadmodule

[spreadtoolkit]
recipe = zc.recipe.cmmi
url = http://yum.zope.com/buildout/spread-src-3.17.1.tar.gz
```

```
[spreadmodule]
recipe = zc.recipe.egg:custom
egg = SpreadModule ==1.4
find-links = http://www.python.org/other/spread/
include-dirs = ${spreadtoolkit:location}/include
library-dirs = ${spreadtoolkit:location}/lib
rpath = ${spreadtoolkit:location}/lib
```

Sometimes a distribution has extension modules that need to be compiled with special options, such as the location of include files and libraries, The custom recipe supports this. The resulting eggs are placed in the develop-eggs directory because the eggs are buildout specific.

This example illustrates use of the zc.recipe.cmmi recipe with supports installation of software that uses configure, make, make install. Here, we used the recipe to install the spread toolkit, which is installed in the parts directory.

## 9.32 Part dependencies

- Parts can read configuration from other parts
- The parts read become dependencies of the reading parts
  - Dependencies are added to parts list, if necessary
  - Dependencies are installed first

In the previous example, we used the spread toolkit location in the spreadmodule part definition. This reference was sufficient to make the spreadtoolkit part a dependency of the spreadmodule part and cause it to be installed first.

## 9.33 Custom develop eggs

```
[buildout]
parts = zodb

[zodb]
recipe = zc.recipe.egg:develop
setup = zodb
define = ZODB_64BIT_INTS
```

We can also specify custom build options for develop eggs. Here we used a develop egg just to make sure our custom build of ZODB took precedence over normal ZODB eggs in our shared eggs directory.

## 9.34 Writing recipes

- The recipe API
  - install
    - __init__

      The initializer is responsible for computing a part's options. After the initializer call, the options directory must reflect the full configuration of the part. In particular, if a recipe reads any data from other sections, it must be reflected in the options. The options data after the initializer is called is used to determine if a configuration has changed when deciding if a part has to be reinstalled. When a part is reinstalled, it is uninstalled and then installed.

  * install

    The install method installs the part. It is used when a part is added to a buildout, or when a part is reinstalled.

    The install recipe must return a sequence of paths that that should be removed when the part is uninstalled. Most recipes just create files or directories and removing these is sufficient for uninstalling the part.

  * update

    The update method is used when a part is already installed and it's configuration hasn't changed from previous buildouts. It can return None or a sequence of paths. If paths are returned, they are added to the set of installed paths.

- uninstall

  Most recipes simply create files or directories and the built-in buildout uninstall support is sufficient. If a recipe does more than simply create files, then an uninstall recipe will likely be needed.

## 9.35 Install Recipes

`mkdirrecipe`.py:

```
import logging, os, zc.buildout

class Mkdir:

    def __init__(self, buildout, name, options):
        self.name, self.options = name, options
        options['path'] = os.path.join(
                              buildout['buildout']['directory'],
                              options['path'],
                              )
        if not os.path.isdir(os.path.dirname(options['path'])):
            logging.getLogger(self.name).error(
                'Cannot create %s. %s is not a directory.',
                options['path'], os.path.dirname(options['path']))
            raise zc.buildout.UserError('Invalid Path')
```

- The path option in our recipe is interpreted relative to the buildout. We reflect this by saving the adjusted path in the options.
- If there is a user error, we:
  - Log error details using the Python logger module.
  - Raise a zc.buildout.UserError exception.

## 9.36 `mkdirrecipe`.py continued

```
def install(self):
    path = self.options['path']
    logging.getLogger(self.name).info(
        'Creating directory %s', os.path.basename(path))
    os.mkdir(path)
    return path

def update(self):
    pass
```

A well-written recipe will log what it's doing.

Often the update method is empty, as in this case.

## 9.37 Uninstall recipes

`servicerecipe.py`:

```
import os

class Service:

    def __init__(self, buildout, name, options):
        self.options = options

    def install(self):
        os.system("chkconfig --add %s" % self.options['script'])
        return ()

    def update(self):
        pass

def uninstall_service(name, options):
    os.system("chkconfig --del %s" % options['script'])
```

Uninstall recipes are callables that are passed the part name and the **original options**.

## 9.38 Buildout entry points

`setup.py`:

```
from setuptools import setup

entry_points = """
[zc.buildout]
mkdir = mkdirrecipe:Mkdir
service = servicerecipe:Service
default = mkdirrecipe:Mkdir

[zc.buildout.uninstall]
service = servicerecipe:uninstall_service
"""

setup(name='recipes', entry_points=entry_points)
```

## 9.39 Exercise 3

- Write recipe that creates a file from source given in a configuration option.
- Try this out in a buildout, either by creating a new buildout, or by extending the `zope.event` buildout.

## 9.40 Command-line options

Buildout command-line:

- command-line options and option setting
- command and arguments

```
bin/buildout -U -c rpm.cfg install zrs
```

Option settings are of the form:

```
section:option=value
```

Any option you can set in the configuration file, you can set on the command-line. Option settings specified on the command line override settings read from configuration files.

There are a few command-line options, like -c to specify a configuration file, or -U to disable reading user defaults.

See the buildout documentation, or use the -h option to get a list of available options.

## 9.41 Buildout modes

- newest
  - default mode always tries to get newest versions
  - Turn off with -N or buildout newest option set to false.
- offline
  - If enabled, then don't try to do network access
  - Disabled by default
  - If enabled, turn off with -o or buildout offline option set to false.

By default, buildout always tries to find the newest distributions that match requirements. Looking for new distributions can be very time consuming. Many people will want to specify the -N option to disable this. We'll see later how we can change this default behavior.

If you aren't connected to a network, you'll want to use the offline mode, -o.

## 9.42 `~/.buildout/default.cfg`

Provides default buildout settings (unless -U option is used):

```
[buildout]
# Shared eggs directory:
eggs-directory = /home/jim/.eggs
# Newest mode off, reenable with -n
newst = false

[python24]
executabe = /usr/local/python/2.4/bin/python

[python25]
executabe = /usr/local/python/2.5/bin/python
```

Unless the -U command-line option is used, user default settings are read before reading regular configuration files. The user defaults are read from the default.cfg file in the .buildout subdirectory of the directory specified in the HOME environment variable, if any.

In this example:

- I set up a shared eggs directory.
- I changed the default mode to non-newest so that buildout doesn't look for new distributions if the distributions it has meet it's requirements. To get the newest distributions, I'll have to use the -n option.
- I've specified Python 2.4 and 2.5 sections that specify locations of Python interpreters. Sometimes, a buildout uses multiple versions of Python. Many recipes accept a python option that specifies the name of a section with an executable option specifying the location of a Python interpreter.

## 9.43 Extending configurations

The `extends` option allows one configuration file to extend another.

For example:

- `base.cfg` has common definitions and settings
- `dev.cfg` adds development-time options:

  ```
  [buildout]
  extends = base.cfg

  ...
  ```

- `rpm.cfg` has options for generating an RPM packages from a buildout.

## 9.44 Bootstrapping from existing buildout

- The buildout script has a `bootstrap` command
- Can use it to bootstrap any directory.
- Much faster than running `bootstrap.py` because it can use an already installed `setuptools` egg.

## 9.45 Example: ~/bin directory

```
[buildout]
parts = rst2 buildout24 buildout25
bin-directory = .

[rst2]
recipe = zc.recipe.egg
eggs = zc.rst2

[buildout24]
recipe = zc.recipe.egg
eggs = zc.buildout
scripts = buildout=buildout24
python = python24

[buildout25]
recipe = zc.recipe.egg
eggs = zc.buildout
scripts = buildout=buildout25
python = python25
```

Many people have a personal scripts directory.

I've converted mine to a buildout using a buildout configuration like the one above.

I've overridden the bin-directory location so that scripts are installed directly into the buildout directory.

I've specified that I want the zc.rst2 distribution installed. The rst2 distribution has a generalized version of the restructured text processing scripts in a form that can be installed by buildout (or easy_install).

I've specified that I want buildout scripts for Python 2.4 and 2.5. (In my buildout, I also create one for Python 2.3.) These buildout scripts allow me to quickly bootstrap buildouts or to run setup files for a given version of python. For example, to bootstrap a buildout with Python 2.4, I'll run:

```
buildout24 bootstrap
```

in the directory containing the buildout. This can also be used to convert a directory to a buildout, creating a buildout.cfg file is it doesn't exist.

## 9.46 Example: zc.sharing (1/2)

```
[buildout]
develop = . zc.security
parts = instance test
find-links = http://download.zope.org/distribution/

[instance]
recipe = zc.recipe.zope3instance
database = data
user = jim:123
eggs = zc.sharing
zcml =
  zc.resourcelibrary zc.resourcelibrary-meta
  zc.sharing-overrides:configure.zcml zc.sharing-meta
  zc.sharing:privs.zcml zc.sharing:zope.manager-admin.zcml
  zc.security zc.table zope.app.securitypolicy-meta zope.app.twisted
  zope.app.authentication
```

This is a small example of the “system assembly” use case. In this case, we define a Zope 3 instance, and a test script.

You can largely ignore the details of the Zope 3 instance recipe. If you aren’t a Zope user, you don’t care. If you are a Zope user, you should be aware that much better recipes have been developped.

This project uses multiple source directories, the current directory and the zc.security directory, which is a subversion external to a project without its own distribution. We’ve listed both in the develop option.

We’ve requested the instance and test parts. We’ll get other parts installed due to dependencies of the instance part. In particular, we’ll get a Zope 3 checkout because the instance recipe refers to the zope3 part. We’ll get a database part because of the reference in the database option of the instance recipe.

The buildout will look for distributions at http://download.zope.org/distribution/.

## 9.47 Example: zc.sharing (2/2)

```
[zope3]
recipe = zc.recipe.zope3checkout
url = svn://svn.zope.org/repos/main/Zope3/branches/3.3

[data]
recipe = zc.recipe.filestorage

[test]
recipe = zc.recipe.testrunner
defaults = ['--tests-pattern', 'f?tests$']
eggs = zc.sharing
       zc.security
extra-paths = ${zope3:location}/src
```

Here we see the definition of the remaining parts.

The test part has some options we haven’t seen before.

- We’ve customized the way the testrunner finds tests by providing some testrunner default arguments.
- We’ve used the extra-paths option to tell the test runner to include the Zope 3 checkout source directory in sys.path. This is not necessary as Zope 3 is now available entirely as eggs.

## 9.48 Source vs Binary

- Binary distributions are Python version and often platform specific
- Platform-dependent distribution can reflect build-time setting not reflected in egg specification.
  - Unicode size
  - Library names and locations
- Source distributions are more flexible
- Binary eggs can go rotten when system libraries are upgraded

Recently, I had to manually remove eggs from my shared eggs directory. I had installed an operating system upgrade that caused the names of open-ssl library files to change. Eggs build against the old libraries no-longer functioned.

## 9.49 RPM experiments

Initial work creating RPMs for deployment in our hosting environment:

- Separation of software and configuration
- Buildout used to create rpm containing software
- Later, the installed buildout is used to set up specific processes
  - Run as root in offline mode
  - Uses network configuration server

Our philosophy is to separate software and configuration. We install software using RPMs. Later, we configure the use of the software using a centralized configuration database.

I'll briefly present the RPM building process below. This is interesting, in part, because it illustrates some interesting issues.

## 9.50 ZRS spec file (1/3)

```
%define python zpython
%define svn_url svn+ssh://svn.zope.com/repos/main/ZRS-buildout/trunk
requires: zpython
Name: zrs15
Version: 1.5.1
Release: 1
Summary: Zope Replication Service
URL: http://www.zope.com/products/zope_replication_services.html

Copyright: ZVSL
Vendor: Zope Corporation
Packager: Zope Corporation <sales@zope.com>
Buildroot: /tmp/buildroot
Prefix: /opt
Group: Applications/Database
AutoReqProv: no
```

Most of the options above are pretty run of the mill.

We specify the Python that we're going to use as a dependency. We build our Python RPMs so we can control what's in them. System packagers tend to be too creative for us.

Normally, RPM installs files in their run-time locations at build time. This is undesirable in a number of ways. I used the rpm build-root mechanism to allow files to be build in a temporary tree.

Because the build location is different than the final install location, paths written by the buildout, such as egg paths in scripts are wrong. There are a couple of ways to deal with this:

- I could try to adjust the paths at build time,
- I could try to adjust the paths at install time.

Adjusting the paths at build time means that the install locations can;'t be controlled at install time. It would also add complexity to all recipes that deal with paths. Adjusting the paths at install time simply requires rerunning some of the recipes to generate the paths.

To reinforce the decision to allow paths to be specified at install time, we've made the RPM relocatable using the prefix option.

## 9.51 ZRS spec file (2/3)

```
%description
%{summary}

%build
rm -rf $RPM_BUILD_ROOT
mkdir $RPM_BUILD_ROOT
mkdir $RPM_BUILD_ROOT/opt
mkdir $RPM_BUILD_ROOT/etc
mkdir $RPM_BUILD_ROOT/etc/init.d
touch $RPM_BUILD_ROOT/etc/init.d/%{name}
svn export %{svn_url} $RPM_BUILD_ROOT/opt/%{name}
cd $RPM_BUILD_ROOT/opt/%{name}
%{python} bootstrap.py -Uc rpm.cfg
bin/buildout -Uc rpm.cfg buildout:installed= \
   bootstrap:recipe=zc.rebootstrap
```

I'm not an RPM expert and RPM experts would probably cringe to see my spec file. RPM specifies a number of build steps that I've collapsed into one.

- The first few lines set up build root.
- We export the buildout into the build root.
- We run the buildout
  - The -U option is used mainly to avoid using a shared eggs directory
  - The -c option is used to specify an RPM-specific buildout file that installs just software, including recipe eggs that will be needed after installation for configuration.
  - We suppress creation of an .installed.cfg file
  - We specify a recipe for a special bootstrap part. The bootstrap part is a script that will adjust the paths in the buildout script after installation of the rpm.

## 9.52 ZRS spec file (3/3)

```
%post
cd $RPM_INSTALL_PREFIX/%{name}
%{python} bin/bootstrap -Uc rpmpost.cfg
bin/buildout -Uc rpmpost.cfg \
   buildout:offline=true buildout:find-links= buildout:installed= \
   mercury:name=%{name} mercury:recipe=buildoutmercury
chmod -R -w .

%preun
cd $RPM_INSTALL_PREFIX/%{name}
chmod -R +w .
find . -name \*.pyc | xargs rm -f

%files
%attr(-, root, root) /opt/%{name}
%attr(744, root, root) /etc/init.d/%{name}
```

We specify a post-installation script that:

- Re-bootstraps the buildout using the special bootstrap script installed in the RPM.
- Reruns the buildout:
  - Using a post-installation configuration that specified the parts whose paths need to be adjusted.
  - In offline mode because we don't want any network access or new software installed that isn't in the RPM.
  - Removing any find links. This is largely due to a specific detail of our configurations.
  - Suppressing the creation of .installed.cfg
  - Specifying information for installing a special script that reads our centralized configuration database to configure the application after the RPM is installed.

We have a pre-uninstall script that cleans up .pyc files.

We specify the files to be installed. This is just the buildout directory and a configuration script.

## 9.53 Repeatability

We want to be able to check certain configuration into svn that can be checked out and reproduced.

- We let buildout tell what versions it picked for distributions
  - Run with -v
  - Look for outout lines of form:

    ```
    Picked: foo = 1.2
    ```

- Include a versions section:

  ```
  [buildout]
  ...
  versions = myversions

  [myversions]
  foo = 1.2
  ...
  ```

## 9.54 Deployment issues

- Need a way to record the versions of eggs used.
- Need a way to generate distributable buildouts that contain all of the source distributions needed to build on a target machine (e.g. source RPMs).
- Need to be able to generate source distributions. We need a way of gathering the sources used by a buildout so they can be distributed with it.

## 9.55 PyPI availability

A fairly significant issue is the availability of PyPI. PyPI is sometimes not available for minutes or hours at a time. This can cause buildout to become unusable.

## 9.56 For more information

See http://www.python.org/pypi/zc.buildout

# CHAPTER 10

# Developing Buildout Recipes

## 10.1 Introduction

Recipes are the plugin mechanism provided by Buildout to add new functionalities to your software building. A Buildout **part** is created by a recipe. Recipes are always installed as Python eggs. They can be downloaded from a package server, such as the Python Package Index (PyPI) , or they can be developed as part of a project using a **develop** egg.

A **develop** egg is a special kind of egg that gets installed as an *egg link* that contains the name of a source directory. Develop eggs don't have to be packaged for distribution to be used and can be modified in place, which is especially useful while they are being developed.

## 10.2 Initial steps

Let's create a recipe as part of the sample project. We'll create a recipe for creating directories. First, we'll create a recipes source directory for our local recipes:

```
$ mkdir mkdir_recipe
```

and then we'll create a source file for our *mkdir* recipe, mkdir.py:

```
import logging, os, zc.buildout

class Mkdir:

    def __init__(self, buildout, name, options):
        self.name, self.options = name, options
        options['path'] = os.path.join(
                              buildout['buildout']['directory'],
                              options['path'],
                              )
        if not os.path.isdir(os.path.dirname(options['path'])):
            logging.getLogger(self.name).error(
                'Cannot create %s. %s is not a directory.',
                options['path'], os.path.dirname(options['path']))
            raise zc.buildout.UserError('Invalid Path')


    def install(self):
        path = self.options['path']
        logging.getLogger(self.name).info(
            'Creating directory %s', os.path.basename(path))
        os.mkdir(path)
        return path

    def update(self):
        pass
```

Currently, recipes must define three methods:

- a constructor,
- an install method, and
- an update method.

The constructor is responsible for updating a parts options to reflect data read from other sections. The buildout system keeps track of whether a part specification has changed. A part specification has changed if its options, after adjusting for data read from other sections, has changed, or if the recipe has changed. Only the options for the part are considered. If data are read from other sections, then that information has to be reflected in the parts options. In the *Mkdir* example, the given path is interpreted relative to the buildout directory, and data from the buildout directory is read. The path option is updated to reflect this. If the directory option was changed in the buildout sections, we would know to update parts created using the mkdir recipe using relative path names.

When buildout is run, it saves configuration data for installed parts in a file named *.installed.cfg* . In subsequent runs, it compares part-configuration data stored in the *.installed.cfg* file and the part-configuration data loaded from the configuration files as modified by recipe constructors to decide if the configuration of a part has changed. If the configuration has changed, or if the recipe has changed, then the part is uninstalled and reinstalled. The buildout only looks at the part's options, so any data used to configure the part needs to be reflected in the part's options. It is the job of a recipe constructor to make sure that the options include all relevant data.

Of course, parts are also uninstalled if they are no-longer used.

The recipe defines a constructor that takes a buildout object, a part name, and an options dictionary. It saves them in instance attributes. If the path is relative, we'll interpret it as relative to the buildout directory. The buildout object passed in is a mapping from section name to a mapping of options for that section. The buildout directory is available as the directory option of the buildout section. We normalize the path and save it back into the options directory.

The install method is responsible for creating the part. In this case, we need the path of the directory to create. We'll use a path option from our options dictionary. The install method logs what it's doing using the Python logging call. We return the path that we installed. If the part is uninstalled or reinstalled, then the path returned will be removed by the buildout machinery. A recipe install method is expected to return a string, or an iterable of strings containing paths to be removed if a part is uninstalled. For most recipes, this is all of the uninstall support needed. For more complex uninstallation scenarios use Uninstall recipes.

The update method is responsible for updating an already installed part. An empty method is often provided, as in this example, if parts can't be updated. An update method can return None, a string, or an iterable of strings. If a string or iterable of strings is returned, then the saved list of paths to be uninstalled is updated with the new information by adding any new files returned by the update method.

## 10.3 Packaging recipe

We need to provide packaging information so that our recipe can be installed as a develop egg. The minimum information we need to specify is a name. For recipes, we also need to define the names of the recipe classes as entry points. Packaging information is provided via a *setup.py* script:

```
from setuptools import setup

setup(
    name = "mkdir_recipe",
    entry_points = {'zc.buildout': ['mkdir = mkdir:Mkdir']},
    )
```

**Default Recipe**

If your package is providing only one recipe, the name of entry point can be given as *default*. So, the user of recipe need not to explicitly mention it in the parts.

Eg:- `entry_points = {'zc.buildout':  ['default = mkdir:Mkdir']}`

Usage:

```
[partname]
recipe = mkdir_recipe
path = mystuff
```

Our setup script defines an entry point. Entry points provide a way for an egg to define the services it provides. Here we've said that we define a zc.buildout entry point named *mkdir*. Recipe classes must be exposed as entry points in the zc.buildout group. we give entry points names within the group.

We also need a README.txt for our recipes to avoid an annoying warning from distutils, on which setuptools and zc.buildout are based:

```
$ touch README.txt
```

The above command will create an empty *README.txt* file.

## 10.4 Using recipes

Now let's update our *buildout.cfg*:

```
[buildout]
develop = mkdir_recipe
parts = data-dir

[data-dir]
recipe = mkdir_recipe:mkdir
path = mystuff
```

Let's go through the changes one by one:

```
develop = mkdir_recipe
```

This tells the buildout to install a development egg for our recipes. Any number of paths can be listed. The paths can be relative or absolute. If relative, they are treated as relative to the buildout directory. They can be directory or file paths. If a file path is given, it should point to a Python setup script. If a directory path is given, it should point to a directory containing a setup.py file. Development eggs are installed before building any parts, as they may provide locally-defined recipes needed by the parts.

```
parts = data-dir
```

Here we've named a part to be "built". We can use any name we want except that different part names must be unique and recipes will often use the part name to decide what to do.

```
[data-dir]
recipe = recipes:mkdir
path = mystuff
```

When we name a part, we also create a section of the same name that contains part data. In this section, we'll define the recipe to be used to install the part. In this case, we also specify the path to be created.

Let's run the buildout. We do so by running the build script in the buildout:

```
$ cd sample_buildout
$ ./bin/buildout
Develop: '/sample-buildout/mkdir_recipe'
Installing data-dir.
data-dir: Creating directory mystuff
```

We see that the recipe created the directory, as expected:

```
$ ls
.installed.cfg
bin
buildout.cfg
develop-eggs
eggs
mystuff
parts
mkdir_recipe
```

In addition, *.installed.cfg* has been created containing information about the part we installed:

```
$ cat .installed.cfg
[buildout]
installed_develop_eggs = /sample-buildout/develop-eggs/mkdir_recipe.egg-link
parts = data-dir
<BLANKLINE>
[data-dir]
__buildout_installed__ = /sample-buildout/mystuff
__buildout_signature__ = recipes-c7vHV6ekIDUPy/7fjAaYjg==
path = /sample-buildout/mystuff
recipe = mkdir_recipe:mkdir
```

Note that the directory we installed is included in *.installed.cfg*. In addition, the path option includes the actual destination directory.

If we change the name of the directory in the configuration file, we'll see that the directory gets removed and recreated:

```
[buildout]
develop = mkdir_recipe
parts = data-dir
```

```
[data-dir]
recipe = mkdir_recipe:mkdir
path = mydata


$ ./bin/buildout
Develop: '/sample-buildout/mkdir_recipe'
Uninstalling data-dir.
Installing data-dir.
data-dir: Creating directory mydata

$ ls
.installed.cfg
bin
buildout.cfg
develop-eggs
eggs
mydata
parts
mkdir_recipe
```

If any of the files or directories created by a recipe are removed, the part will be reinstalled:

```
$ rmdir mydata
$ ./bin/buildout
Develop: '/sample-buildout/recipes'
Uninstalling data-dir.
Installing data-dir.
data-dir: Creating directory mydata
```

## 10.5 Publishing recipe to PyPI

If you are adding a recipe to PyPI, use the `Framework :: Buildout` trove classifier, so that it will be automatically listed in the PyPI list .

More details about uploading eggs to PyPI are given in setuptools documentation.

## 10.6 Conclusion

Recipes are the plugin mechanism provided by Buildout. There are hundreds of recipes available in PyPI and some important ones are listed in the recipe list . If you need any functionality for building your application check the list of recipes available, otherwise you can create one yourself.

# CHAPTER 11

# Links

- Developing Django apps with zc.buildout

  An article by Jacob Kaplan-Moss, the creator of Django.

- More buildout notes

  Another article by Jacob Kaplan-Moss, the creator of Django.

- A Django Development Environment with zc.buildout

  This article will show you how to create a repeatable Django development environment from scratch using zc.buildout.

- Howto install Pylons with buildout

  This document describes installing Pylons using Buildout.

- Managing projects with Buildout

  Learn about eggs, setuptools and dependency management, and how to use Buildout to set up a development environment.

- Buildout Quick Reference Card

  Always going searching for answers to your buildout questions? This reference card puts all that information into one handy four-page laminated leaflet.

- Brandon's page about Buildout

  Brandon says, "This page is where I am collecting all of the hints that I accumulate about using buildout, the Python development and deployment technology invented by the Zope folks."

- Buildout docs from Grok community

  Modified version of Jim Fulton's tutorial for using buildout, originally given at DZUG 2007.

- buildout tutorial. buildout howto. buildout review.

  This post is a review, a tutorial, and a howto - for and about buildout - a software development tool for the python language.

- zc.buildout vs. "plain" setuptools

  A criticism about Buildout. There is one reply to the criticism.

- Buildout development/production strategy

  Reinout van Rees says, "Buildout is great for development and for deployment. How to combine the two? Jean-Paul Ladage and me did some brainstorming on this and we'll show you what we came up with."

- django-buildout

  Django support for zc.buildout.

- Using buildout on Windows

  Detailed explanation on what to do to get buildout running with a Plone / Zope environment on Windows.

- Easily creating repeatable buildouts

  When you create a buildout for a production website, you want to be able to recreate that exact buildout one year from now on a different server. How do you do that? This is a more simple and better working version of an earlier post.

- Buildout tutorial

  Yet another tutorial.

- mr.developer

  mr.developer is a zc.buildout extension which makes it easier to work with buildouts containing lots of packages of which you only want to develop some.

- Installing GeoDjango with PostgreSQL and zc.buildout

# CHAPTER 12

# Buildout Recipes

Recipes are the plugin mechanism provided by Buildout to add new functionalities to your software building. You can read the Developing Buildout Recipes to learn more about creating Buildout recipes.

This is a list of Buildout recipes created by the community. All the recipes listed here are available from PyPI . You can see a complete list of recipes in PyPI .

If you are adding a recipe to PyPI, please use the `Framework :: Buildout` trove classifier, so that it will be automatically listed in the PyPI list .

The Recipes with their short description are as follows :

## 12.1 zc.recipe.egg

zc.recipe.egg is a buildout recipe to install Python package distributions as eggs

## 12.2 zc.zope3recipes

zc.zope3recipes is a buildout recipe to define Zope 3 applications

## 12.3 djangorecipe

djangorecipeRecipe is a buildout recipe for Django .

## 12.4 plone.recipe.apache

plone.recipe.apache is a buildout recipe to build and configure apache.

## 12.5 z3c.recipe.ldap

z3c.recipe.ldap is a buildout recipe to deploy an OpenLDAP server.

## 12.6 gp.recipe.pip

gp.recipe.pip is a buildout recipe to to install python packages using pip. More

## 12.7 fez.djangoskel

fez.djangoskel is a buildout recipes for creating Django applications as eggs.

## 12.8 z3c.recipe.eggbasket

z3c.recipe.eggbasket is a buildout recipe to install eggs from a tarball and create that egg.

## 12.9 tl.buildout_gtk

tl.buildout_gtk is a buildout recipe to install PyGTK, PyObject and PyCairo.

## 12.10 rod.recipe.appengine

rod.recipe.appengine is a buildout to set up a google appengine development environment.

## 12.11 buildout_script

buidlout_scripe is a zc.buidlout recipe which is used for generating scripts from template file. The scripts are generated from templates and the buildout-variables get replaced by actual data at run time.

## 12.12 cc.buildout_reports

The cc.bulidout_reports:xxx can be used to scan a project for comments flagged with a specifed pattern (XXX|TODO by default). The recipe does not scan temporary files (*~) or traverse into egg and .svn directories.

## 12.13 cc.gettext

The cc.gettext:msgfmt can be used to compile gettext catalogs from the .po source format to the binary .mo representation needed by Zope 3.

## 12.14 Collective.recipe.ant

Collective.recipe.ant executes an ant build. It assumes java, and and ant is installed on the system.

## 12.15 collective.recipe.backup

collective.recipe.backup provides sensible defaults for your common backup tasks. bin/repozo is a zope script to make backups of your Data.fs.

## 12.16 collective.recipe.bootstrap

collective.recipe.bootstrap is used to automatically update the bootstrap.py file in buildout directory.

## 12.17 collective.recipe.i18noverrides

collective.recipe.i18noverrides is a buildout recipe which creates an i18n directory within one or more zope 2 instances in your buildout. It copies some .po files to those directories. The translations in those .po files will override any other translations.

## 12.18 collective.recipe.isapiwsgi

collective.recipe.isapiwsgi is a zc.buildout recipe which creates a paste.deploy entry point for isapi-wsgi.

## 12.19 collective.recipe.libsvm

collective.recipe.libsvm is a recipe to compile libsvm with python in a buildout

## 12.20 collective.recipe.minify

collective.recipe.minify is a minify-wrapper for CSS & JavaScript resources for removing all the unnecessary white spaces and comments.

## 12.21 collective.recipe.modwsgi

collective.recipe.modwsgi is a collective.recipe.modwsgi" is a zc.buildout recipe which creates a paste.deploy entry point for mod_wsgi.

## 12.22 collective.recipe.mxbase

collective.recipe.mxbase is a buildout recipe to install eGenix mx.base

## 12.23 collective.recipe.mxodbc

collective.recipe.mxodbc is a buildout recipe to install eGenix mx.ODBC and a license.

## 12.24 collective.recipe.omelette

collective.recipe.omelette is a buidlout recipe which creates a unified directory structure of all namespace packages, symlinking to the actual contents, in order to ease navigation.

## 12.25 collective.recipe.patch

collective.recipe.patch is a buildout recipe for patching eggs.

## 12.26 collective.recipe.platform

collective.recipe.platform is a buildout recipe which provide buildout variables with platform specific values.

## 12.27 collective.recipe.plonesite

collective.recipe.plonesite is a buildout recipe to create and update a plone site. This recipe enables you to create and update a Plone site as part of a buildout run. This recipe only aims to run profiles and Quickinstall products. It is assumed that the install methods, setuphandlers, upgrade steps, and other recipes will handle the rest of the work.

## 12.28 collective.recipe.rsync_datafs

collective.recipe.rsync_datafs is a simple zc.buildout recipe to to synchronize data from one place to another. Typically, it is used to transfer a Zope Data.fs file from production to development.

It assumes you have a UNIX-based operating system and that the rsync binary is in your path when you run buildout.

## 12.29 collective.recipe.scriptgen

collective.recipe.scriptgen is a zc.buildout recipe for generating a script.

## 12.30 collective.recipe.seleniumrc

collective.recipe.seleniumrc is a zc.buildout recipe for installing the Selenium RC distribution. This package downloads and installs Selenium RC using zc.buildout. It is based on hexagonit.recipe.download.

## 12.31 collective.recipe.solrinstance

collective.recipe.solrinstance is a zc.buildout to configure a solr instance.

## 12.32 collective.recipe.sphinxbuilder

collective.recipe.sphinxbuilder is a zc.buildout recipe to generate and build Sphinx-based documentation in the buildout.

## 12.33 collective.recipe.supervisor

collective.recipe.supervisor is a buildout recipe to install supervisor.

## 12.34 collective.recipe.template

collective.recipe.template is a buildout recipe to generate a text file from a template.

## 12.35 collective.recipe.updateplone

collective.recipe.updateplone is a buildout recipe to update plone sites.

## 12.36 collective.recipe.vimproject

collective.recipe.vimproject is a buildout recipe to set up a VIM development environment.

## 12.37 collective.recipe.z2testrunner

collective.recipe.z2testrunner is a buildout recipe for generating zope2-based test runner. A zc.buildout recipe for generating test runners that run under a Zope 2 environment and is "Products"-aware.

## 12.38 collective.recipe.zcml

collective.recipe.zcml is a buildout recipe to create zcml slugs. ZCML slug generation to be used separately e.g for repoze based setups.

## 12.39 collective.recipe.zope2cluster

collective.recipe.zope2cluster is a buildout recipe to create a zope cluster.

NOTE: This recipe is no longer needed as of zc.buildout 1.4.

## 12.40 collective.recipe.zope2wsgi

collective.recipe.zope2wsgi is a buildout recipe to generate zope instances using repoze.zope2.

## 12.41 collective.transcode.recipe

collective.transcode.recipe is a buildout recipe to setup a transcode daemon.

## 12.42 gocept.cxoracle

gocept.cxoracle is a zc.buildout recipe for installing cx_Oracle.

## 12.43 gocept.download

gocept.download is a zc.buildout recipe for downloading and extracting an archive.

## 12.44 iw.recipe.cmd

iw.recipe.cmd is a zc.buildout recipe to execute a command line.

## 12.45 zc.recipe.cmmi

zc.recipe.cmmi is a zc.buildout recipe for configure/make/make install. The configure-make-make-install recipe automates installation of configure-based source distribution into buildouts.

## 12.46 zc.recipe.filestorage

zc.recipe.filestorage is a zc.buildout recipe for defining a file-storage.

## 12.47 z3c.recipe.mkdir

z3c.recipe.mkdir is a buildout recipe to create directories.

## 12.48 z3c.recipe.sphinxdoc

z3c.recipe.sphinxdoc is a buildout recipe which use Sphinx to build documentation for zope.org.

## 12.49 z3c.recipe.template

z3c.recipe.template is a buildout recipe to generate a text file from a template.

## 12.50 zest.recipe.mysql

zest.recipe.mysql is a buildout recipe to setup a MySQL database.

## 12.51 zc.sshtunnel

zc.sshtunnel is a zc.buildout recipe to manage an SSH tunnel.

## 12.52 zeomega.recipe.mxodbcconnect

zeomega.recipe.mxodbcconnect is a buildout recipe to install eGenix mx.ODBCConnect.Client.

## 12.53 zc.recipe.icu

zc.recipe.icu is a zc.buildout recipe for installing the ICU library into a buildout.

## 12.54 z3c.recipe.filetemplate

z3c.recipe.filetemplate is a zc.buildout recipe for creating files from file templates.

## 12.55 tl.buildout_apache

tl.buildout_apache is a zc.buildout recipes for setting up an Apache web server environment.

## 12.56 plone.recipe.zeoserver

plone.recipe.zeoserver is a zc.buildout recipe for installing a ZEO server.

## 12.57 plone.recipe.varnish

plone.recipe.varnish is a buildout recipe to install varnish.

## 12.58 plone.recipe.zope2install

plone.recipe.zope2install is a zc.buildout recipe for installing Zope 2.

## 12.59 plone.recipe.zope2instance

plone.recipe.zope2instance is a zc.buildout recipe for installing a Zope 2 instance.

## 12.60 plone.recipe.zope2zeoserver

plone.recipe.zope2zeoserver is a zc.buildout recipe for installing a Zope 2 ZEO server.

## 12.61 hexagonit.recipe.download

hexagonit.recipe.download is a zc.buildout recipe for downloading and extracting packages.

## 12.62 amplecode.recipe.template

amplecode.recipe.template is a buildout recipe for making files out of Jinja2 templates.

## 12.63 gocept.recipe.env

gocept.recipe.env is a buildout recipe which provides a section for Mirror environment variables.

## 12.64 hexagonit.recipe.cmmi

hexagonit.recipe.cmmi is a zc.buildout recipe for compiling and installing source distributions.

## 12.65 iw.recipe.sendmail

iw.recipe.sendmail is a zc.buildout recipe to setup zope.sendmail in a Zope2 instance.

## 12.66 plone.recipe.runscript

plone.recipe.runscript is a buildout recipe to run a zope script.

## 12.67 tl.buildout_virtual_python

tl.buildout_virtual_python is a zc.buildout recipe for creating a virtual Python installation.

## 12.68 gocept.ctl

gocept.ctl is a zc.buildout recipe to create a convenience-script for controlling services.

## 12.69 djangout

djangout is a buildout recipes for Django.

## 12.70 gocept.zope3instance

gocept.zope3instance is a zc.buildout recipe for defining a Zope 3 instance.

## 12.71 z3c.recipe.staticlxml

z3c.recipe.staticlxml is a buildout recipe to build lxml.

## 12.72 z3c.recipe.tag

z3c.recipe.tag is a buildout recipe to generate ctags from eggs for development.

## 12.73 z3c.recipe.usercrontab

z3c.recipe.usercrontab is a user Crontab install buildout recipe.

# CHAPTER 13

# Community

Buildout is used by many Python based projects and communities. Also it is used by many organization for developing and deploying applications.

## 13.1 Mailing list

Buildout use Python's distutils-sig for all discussions. You can send your queries to the distutils-sig mailing list.

## 13.2 Source code

The source code of Buildout is maintained in a Git repository. To check out the latest release:

```
git clone git://github.com/buildout/buildout
```

You can also browse the source code online .

You can contact Jim Fulton or ask in the *distutils-sig* for contributing to this project.

## 13.3 IRC channel

There is an IRC channel *#buildout* at freenode.net .

## 13.4 Issue tracker

Bugs and other issues are tracked on Github . Feel free to submit issues.

## 13.5 Website development

The www.buildout.org source code is maintained in a Git repository. To check out the trunk:

```
git clone git://github.com/buildout/buildout.github.com
```

You can also browse the source code online.

# CHAPTER 14

# Glossary

**builder** A class (inheriting from `Builder`) that takes parsed documents and performs an action on them. Normally, builders translate the documents to an output format, but it is also possible to use the builder builders that e.g. check for broken links in the documentation, or build coverage information.

**configuration directory** The directory containing `conf.py`. By default, this is the same as the *source directory*, but can be set differently with the **-c** command-line option.

**description unit** The basic building block of Sphinx documentation. Every “description directive” creates such a unit; and most units can be cross-referenced to.

**environment** A structure where information about all documents under the root is saved, and used for cross-referencing. The environment is pickled after the parsing stage, so that successive runs only need to read and parse new and changed documents.

**source directory** The directory which, including its subdirectories, contains all source files for one Sphinx project.

# CHAPTER 15

# Thanks

- Thanks to Jim Fulton for developing this great tool.
- Thanks to all other contributors to Buildout.
- Thanks to creators of all Buildout recipes.
- Thanks to all contributors to this website (please add your name here):
  - Baiju M
  - Jeff Rush
  - Marius Gedminas
  - Martijn Faassen
  - Paul Carduner
  - Roberto Allende
- Thanks to Jens Vagelpohl for setting up domain & site.
- Thanks to Christian Theune, Gocept & DZUG for donating domain to Zope Foundation
- Thanks to the creators of images used in the front page. They are selected from Flickr and all are licensed under CC licenses.
  - watch.jpg: http://www.flickr.com/photos/66164549@N00/2491909299
  - start.jpg: http://www.flickr.com/photos/waytru/511060488
  - learn.jpg: http://www.flickr.com/photos/munaz/2517170549
  - involve.jpg: http://www.flickr.com/photos/e_phots/2642111978
- Thanks to all user of Buildout.

CHAPTER 16

# What is Buildout?

Buildout is a system for managing development buildouts. While often identified as a Zope project, and indeed licensed under the ZPL by Zope creator Jim Fulton, buildout is useful for configurations beyond Zope, and even, in rare cases, a few that have nothing to do with Python.

# CHAPTER 17

# Features

- Coarse-grained python-based configuration-driven build tool
- Tool for working with eggs
- Repeatable
- Configuration driven
- Developer oriented
- Python-based
- Can use with virtualenv

# Index