Buildout: Precision Assembly, Repeatability, Islands

Author: Jeff Rush

Date: March 13, 2008
Series: Python Eggs and Buildout Deployment - PyCon 2008 Chicago

A follow-on to the setuptools talk introducing the buildout tool that uses parts specifications to repeatably bring together specific combinations and versions of Python eggs, along with non-Python elements, into controlled islands of development and deployment.

Roadmap to Talk

- The Benefits
- What Is It?
- How Does It Work?
- The Concepts
- Getting Started: Dirs, Specs, Args
- Recipe Functionality
- Examples of Usage
- For More Information

Index of Slides

The Benefits of Buildout

- intended for
    - creating applications, not libraries
    - final deployment AND
    - daily development
- islands of development
    - does NOT install into system Python
- blueprints lead to repeatability
- not just for Python software
- not necessary to eggify software base
- offline usage

In the prior talks on distutils and then setuptools, we focused on creating distributions of reusable modules, more in the sense of libraries. Buildout takes us in a different direction, using those packaging capabilities to bring together sets of distributions into whole applications in a controlled manner.

These applications can be deployed as self-contained source releases and RPMs in ways that facilitate operation by experienced Unix system administrators. Prior to deployment however, buildout is a useful tool in the development phase as well.

The buildout tool is based on the premise that installing distributions into the system instance of Python is, for a developer, a bad thing that leads to conflicts and unknown interactions with packages not under control of buildout. For this reason, buildout relies upon sandboxes or "islands of development", similar to how virtualenv work. In fact it can be used along with virtualenv.

Buildout is based on the idea of engineering blueprints; that an architect can rigorously specify the parts that go into an assembly and construct a product in a repeatable fashion. The word "buildout" comes from the manufacturing industry and refers to a specification of a set of parts and instructions on how to assemble them.

Note that parts could still behave differently due to changes in parts of the environment, such as system libraries, not controlled by the buildout.

Unlike the packaging tools covered previously, buildout encompasses not just Python software but non-Python elements such as configuration sets, multiple programs, Apache instances, database servers and so forth.

As a result of this, it is NOT necessary to eggify your software base to use buildout.

And buildout, while relying upon a package repository such as the Cheeseshop, is also able to function offline from the net from collections of parts within a cache directory.

What is Buildout?

- Jim Fulton of Zope Corporation
- leverages setuptools and eggs
- for developers, not end-users
- coarse-grained build system (config-based)
- NOT fine-grained (rules-based)
    - Make, scons, distutils
- explicit, declarative, not tweakable
- anything built by buildout ...
    - is controlled by it.

buildout was conceived by Jim Fulton of Zope Corporation in 2006 and, while often used with the Zope web framework, is completely independent of it.

It draws from a extensible collection of recipes in driving the assembly process and leverages setuptools and eggs in managing Python packages.

Because of its architectural focus, the audience for buildout is more toward developers than the end-user.

buildout is a course-grained build system, differing from fine-grained approaches such as Make, scons and distutils. Those systems focus on individual files and use rules to determine how to compute one from another. Buildout works with larger elements such as applications, configuration files and databases, and uses configuration instead of rules to fit them together.

Rule systems are better used where the sheer number of many low-level elements require taking advantage of regularities to reduce complexity. Configuration systems are better at specifying the one-off relationships when you have relatively few high-level elements.

In one sense it is a better Make but works at a higher level than Make, dealing with large components rather than individual files.

buildout is not ideal for informal experimentation, in that it requires explicit specification of the parts used in an application. This is done in a declarative manner, in that the architect says "what to use", not "how to do it". This makes tweaking to control the low-level process difficult.

Part of using buildout is understanding that anything built by buildout is controlled by it. Temporary hacks to created files will be thrown away on the next build. To make a permanent change, it is necessary to update the buildout configuration.

There can be exceptions to this such as the recipes that manage checkouts. They don't remove checkouts to avoid losing user data. Similarly, the zc.recipes.filestorage recipe doesn't remove data directories it creates on uninstall.

How does Buildout Work?

- reads an assembly specification
- to install (uninstall) into a deployment tree
- specific parts and scripts, based on
- a state file indicating what is in the tree
- to make the tree match the specification.

- uses setuptools to find/install eggs
- and plugins (eggs) called recipes
- for flexible parts handling, both egg and non-egg
- allowing parts to reference other parts
- and scripts to have custom import module-sets.

buildout is a tool that, **each** time it is run, ...

To accomplish this, buildout ...

The Concepts of Buildout

- specification
- recipes
- parts
- custom interpreters
- executable scripts
- deployment tree

Concepts: What is a Specification?

- one or more (included) text files
- format of ConfigParser
- different specs for different modes
- constrain versions and repositories

A specification is a text file that itemizes the parts that go into an assembly, names the recipes used by the various parts, and provides to those recipes an open-ended form of configuration.

When stored in a software control repository, it can reproduce an exact deployment or development scenario, upon being checked out and having a build operation invoked upon it.

The format of a specification corresponds to that accepted by ConfigParser, a standard Python module. A specification can be given in a single such file or factored into multiple ones.

Most commonly, there will be multiple specifications for a project, say one for development, one for testing and another for field deployment.

Specifications do more than just list the parts involved. They can place constraints on acceptable versions and provide details on where to automatically download them from, whether the Cheeseshop or project-specific websites.

If a part is removed from a specification, it is uninstalled from the deployment tree. If a part's receipe or configuration changes, the part is uninstalled and reinstalled.

Concepts: What is a Part?

- "something to be managed by a buildout"

- may be an egg, tarball, checkout, config file, etc.
- has a name and its own data directory
- is an object with attributes, inside buildout
- gets installed, updated and uninstalled
- can reference other parts
- is defined by one recipe that handles it
  - along with recipe-specific parameters

A part is simply something to be managed by a buildout.

It can be almost anything, such as a Python package, a program, a directory, or even a configuration file.

It has a name unique within the specification and its own directory within which it can scribble anything. These scribbles can be referenced by other parts.

Within buildout, a part is an object with an open-ended set of attributes.

It may be installed, updated and uninstalled, over a series of builds. If a part reference is removed from a specification, upon the next invocation of Buildout, it is uninstalled from the deployment tree. If a part's recipe or configuration changes, the part is uninstalled and reinstalled.

A part can reference other parts within the same specification, accessing their attributes, configuration and private directory.

Each part is defined by a recipe, which contains the logic to manage them, along with some data used by that recipe specific to that part.

Concepts: What is a Recipe?

- is what buildout is made of
- are themselves eggs
- can contain multiple sub-recipes
- get installed automatically
- a set of starter recipes for
  - installing eggs
  - generating scripts
  - custom Python interpreters
  - custom egg compilation
- many 3rd party recipes in the Cheeseshop

buildout itself is constructed out of recipes, which are objects that know how to install, update and uninstall a type of part.

Receipes are themselves eggs and when one is referenced in a specification, buildout will automatically locate and install the recipe in the buildout environment.

A recipe can contain multiple sub-recipes, accessible as distinct egg entrypoints.

A set of starter recipes ships with the buildout, in the egg named zc.recipe.egg.

The Cheeseshop contains many add-on recipes, if you search for "recipe" in the name or keyword field.

Getting Started with Buildout

- Installing Buildout
- Project Directory Structure
- Specification File
- Controlling Versions
- About Caches
- How Distributions Are Found
- The Command-Line
- Configuration of Defaults

Getting Started: Installing Buildout

- globally as an egg
  - `easy_install zc.buildout`
- locally under a project
  - `python bootstrap.py`

```
$ svn co svn://svn.zope.org/.../z3hello/trunk z3hello
$ cd z3hello
$ python2.4 bootstrap.py
$ bin/buildout buildout:newest=false
$ bin/instance fg
```

The buildout software can be installed system-wide, using easy_install or locally under a project, by running the "bootstrap.py" that is bundled with most existing projects.

The "bootstrap.py" command will:

- create support directories, like bin, eggs, and work, as needed,
- download and install the zc.buildout and setuptools eggs,

Here is an example of setting up an existing project that uses buildout. Note that it takes a while to download and build everything it needs.

The full URL for the example is:

svn://svn.zope.org/repos/main/Sandbox/baijum/z3hello/trunk z3hello

Getting Started: Project Directory Structure

```
project/
    bootstrap.py
    buildout.cfg
    bin/
        buildout
    develop-eggs/
    parts/

.installed.cfg
eggs/
downloads/
```

A project directory usually contains a "bootstrap.py" script to help a new developer set up the tree after checking out a project. The file is optional.

The specification for the entire project defaults to "buildout.cfg" but there are often others, such as "deployment.cfg" and "production.cfg".

In the "bin/" directory are the executable scripts that buildout generates from entrypoints within distributions.

The "develop-eggs/" directory holds **egg links** for software being developed in the buildout. We separate "develop-eggs/" and "eggs/" to allow egg cache directories to be shared across multiple buildouts. For example, a common developer technique is to define a common eggs directory in their home that all non-develop eggs are stored in. This allows larger buildouts to be set up much more quickly and saves disk space.

And the "parts/" directory is contains code and data managed by buildout, or more precisely the recipes that make it up.

If you look hard, you will also find a hidden file named ".installed.cfg", which is where buildout keeps its state of what is currently installed. Do not tamper with it.

And if you did not change the default locations of the cache directories for eggs and tarballs, there will be an "eggs/" and "downloads/" directory. A difference between the two is that those in "eggs/" will be referenced "in-place" while those in "downloads/" will be unpacked into a subdirectory of "parts/".

Getting Started: Project Directory Structure (cont'd)

README.txt
setup.py
setup.cfg
src/

var/ (Zope instance data area)
products/ (Zope2 products)

- svn-ignore:
  - eggs/, downloads/
  - bin/, develop-eggs/, parts/
  - var/
  - .installed.cfg

And of course there are the other files and directories about which buildout is not concerned.

There is usually a "README.txt" file because several tools complain if it is not there. If the build is itself an egg (and not all are), there will also be "setup.py" and "setup.cfg" files.

And there is often a "src/" directory under which the source of your own eggs or checkouts reside.

If the build represents a Zope instance, there may also be a "var/" directory to hold the instance data such as a ZODB, and a "products/" directory to contain Zope Products, which are used in Zope 2.

A question that usually arises with a project is which parts to check into a version control system and which are automatically generated and managed by buildout.

Obviously the two distribution cache directories should not be checked in.

Nor should the "bin/" directory into which buildout places generated scripts, the "develop-eggs/" directory which is really just a collection of egg-links that point into your "src/" directory for work under development, or the "parts" directory under which recipes store somewhat transient data belong to the part they manage.

And if you're running Zope, it is not common to check the "var/" directory in, unless your policy is to store frozen ZODB databases.

And last, the ".installed.cfg" file that buildout uses to keep track of the state of parts should not be checked in. buildout will generate it as needed upon the next build operation.

Getting Started: Specification File (syntax)

```
[buildout]
parts = ODBC_installation ODBC_configuration

[ODBC_installation]
recipe = zc.recipe.cmmi
url = http://www.demo.org/unixODBC-2.2.12.tar.gz
extra_options = --disable-gui

[ODBC_configuration]
recipe = tau.recipe.odbc:iniwriter
odbc_ini =
    [SQL2000DEV]
    Driver = FreeTDS
```

Specification files are in the format accepted by the ConfigParser Python module, with variable-definition and substitution extensions. Such a file is broken into `[sections]`, where each part has their own section and name.

Within sections are "option = value" lines. A value can be spread across multiple lines by indenting it.

The "[buildout]" defines the buildout section and is the only required section in the specification file. It is options in this section that may cause other sections to be used.

The "parts = <space-delimited names>" option lists the parts that go into an assembly. Parts that depend on other other parts not specified here will automatically be identified and pulled in as well.

Each part is then further described under its section. The first option described for every part is "recipe=", which identifies the plugin used to manage it. All other options under a part description are dependent upon what that recipe accepts. For the curious, options are passed as keyword arguments to recipe objects.

The recipe "zc.recipe.cmmi" is one that understands how to download a tarball and perform the common sequence of commands: ./configure; make; make install". That installation occurs into the "parts/" directory, into a subdirectory named after the part. The recipe takes a "url=" option that tells it from where to download the archive.

Notice that installation and configuration are treated as separate operations. This is a good policy to folow for buildouts, to among other things, enhance specification reusability in different environments (development, testing, deployment).

The recipe "tau.recipe.odbc" accepts a multiline value and writes it into a file of the name as the option. The value can contain any text, as long as it is indented in the specification.

Getting Started: Specification File (references)

- ${partname:optionname}
- parts processed in order, beware circular refs

```
[Zope2_installation]
recipe = plone.recipe.zope2install
url = http://www.zope.org/Products/Zope-2.9.7.tgz

[Zope2_instance]
recipe = plone.recipe.zope2instance
zope2-location = ${Zope2_installation:location}
```

Within a specification file, parts can reference attributes of other parts, such as the "location" of their parts directory. Any "option = value" field can be referenced in this way.

Parts declarations are processed in the order they appear in the specification file, so avoid circular references.

Parts referenced in this manner automatically become dependencies of the reading part. It is the same as putting its name in the [buildout] parts= option.

Getting Started: Specification File (includes)

- **base.cfg** common specification

```
[buildout]
...
```

- **dev.cfg** development specification

```
[buildout]
extends = base.cfg
```

- **rpms.cfg** RPM generation specification

```
[buildout]
extends = base.cfg
```

Specification files can include one another, to factor our common options and provide for distinct deployment target environments.

Getting Started: Controlling Versions

```
[buildout]
newest = false # or current acceptable egg?
prefer-final = true # or under-develop releases?
versions = release-1 # section of explicit versions

[release-1]
spam = 1
eggs = 2.2

# error if any version is picked automatically?
allow-picked-versions = false
```

buildout offers several degrees of control over the versions of parts used for assemblies. These options can be specified either in the per-user $HOME/.buildout/default.cfg or in a per-project buildout specification file. Some policies make more sense in one than the other.

The default mode of operation for buildout is to always try to find the latest distributions that match requirements. Often going over the network, this lookup operation can be very time consuming. The **newest** option can disable this, so that buildout will use the currently installed eggs as long as they meet the requirements. It also lends a certain stability to the development environment. The **-N** command-line option also disables it.

When searching for new releases is enabled, the newest available release is used. This isn't usually ideal, as you may get a development release or alpha releases not ready to be widely used. The **prefer-final** option controls whether to only use the latest final or stable releases.

In buildout version 2, final releases will be preferred by default. You will then need to use a false value for **prefer-final** to get the newest releases.

In order to give more control over the precise version of distributions used, a **versions** option can be specified in the [buildout] section that points to a section that itemizes the versions to be used.

To populate this section, running buildout in verbose mode will print the versions selected of the various distributions.

To insure no versions slip past and are picked automatically, the **allow-picked-versions** can be used to disable the automatic process and generate an error, giving absolute control over version selection.

Getting Started: About Caches

- establishing shared caches:

```
[buildout]
eggs-directory = /var/tmp/buildout/eggs
download-cache = /var/tmp/buildout/downloads
```

- only get things from the cache?

install-from-cache = true
offline = true # obsolete

Normally, when distributions are installed, if any processing is needed, they are downloaded from the internet to a temporary directory and then installed from there. A download cache can be used to avoid the download step. This can be useful to reduce network access and to create source distributions of an entire buildout.

buildout supports two cache locations: one for eggs, and one for tarball archives. Without specifying these options, the default is to use directories "eggs" and "downloads" within each project directory tree.

A cache can be used as the basis of application source releases. In an application source release, we want to distribute an application that can be built without making any network accesses. In this case, we distribute a buildout with download cache and tell the buildout to install from the download cache only, without making network accesses. The buildout **install-from-cache** option can be used to signal that packages should be installed _only_ from the download cache.

The **offline** option is related, in that it tells buildout whether it is allowed to search distribution repositories on the network.

Getting Started: How Distributions Are Found

```
[buildout]
find-links =
    http://dist.plone.org
    http://download.zope.org/ppix/
    http://download.zope.org/distribution/
    /opt/packages/zope3/
```

- buildout: other sites, then Cheeseshop
- setuptools: Cheeseshop, then other sites

- find-links also specified per some recipes

To find distributions, buildout uses the search mechanism built into setuptools, and allows specification of places, in addition to the Cheeseshop, in which to look.

To use an index server other than the Cheeseshop, specify its URL with the --index-url (or index-url = URL) configuration option. There is no provision to have multiple index servers.

NOTE: buildout searches those sites given with --find-links after it searches an index server like the Cheeseshop. setuptools searches in the opposite order.

For installing on non-networked machines, a link server can be represented as simply a directory of eggs or source packages, pointed to with the --find-links* command-line option.

Getting Started: The Command-Line

- buildout [options] [assignments] [cmd [cmd args]]
- options
  - -c deploy.cfg
  - -o (offline)
  - -n (newest)
  - -D (debug)
- assignments
  - section:option=value
- commands
  - buildout init
  - buildout install [parts]
  - buildout runsetup setup.py sdist register upload

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.

`-c` *`config_file`*	Specify path to the buildout configuration file to be used. This defaults to the file named "buildout.cfg" in the current working directory.
`-o`	Run in off-line mode. This is equivalent to the assignment "buildout:offline=true".
`-n`	Run in newest mode. This is equivalent to the assignment "buildout:newest=true". With this setting, which is the default, buildout will try to find the newest versions of distributions available that satisfy its requirements.
`-D`	Debug errors. If an error occurs, then the post-mortem debugger will be started. This is especially useful for debugging recipe problems.

Getting Started: Configuration of Defaults

- two-layer configuration
  - $HOME/.buildout/default.cfg
  - project specification
- no system-wide settings
- per-project settings in specification

buildout always looks for an initial configuration file under the $HOME directory and loads it before the assembly specification file. The syntax of the two files is identical; anything that can go into a specification file can go into a defaults file.

Notice from this that there are no system-wide settings, like there was with setuptools.

Besides parts information, buildout settings can also go into the per-project assembly specification.

About Recipes

- zc.recipe.egg
- iw.recipe.subversion
- hexagonit.recipe.download
- zc.recipe.cmmi
- zc.recipe.testrunner
- zc.sshtunnel
- z3c.recipe.ldap
- tl.buildout_apache
- z3c.recipe.openoffice
- zc.recipe.zope3checkout

zc.recipe.egg

> Installs one or more eggs, along with their dependencies. It installs their console-script entry points with the eggs needed included in their paths.

zc.recipe.testrunner

> Generates scripts to run project-specific unit tests over a collection of eggs. The eggs must already be installed (using the zc.recipe.egg recipe).

zc.recipe.zope3checkout

> Installs a checkout from the Zope 3 repository.

zc.recipe.zope3instance

> Sets up a server instance for running Zope 3.

zc.recipe.filestorage

> Create an empty instance of ZODB filestorage and generates a configuration clause in the style of ZConfig for using it.

Recipes: Egg Installation and Scripts

- zc.recipe.egg
    - :eggs
    - :scripts
    - :custom
    - :develop

```
[doculator]
recipe = zc.recipe.egg:eggs
eggs = docutils
    ZODB3 <=3.8
find-links =
index =
```

The zc.recipe.egg recipe installs one or more eggs, with their dependencies. It has four sub-recipes that can be references by adding a colon and their name to the recipe= line. The default sub-recipe is "scripts".

The eggs option accepts one or more distribution requirements, one per line. Acceptable versions can be specified. Any dependencies of the named eggs will also be installed.

It is also possible to specify a part-custom "find-links=" list of places to look for distributions as well as the location of a specific index server such as the Cheeseshop.

Recipes: Customizing the Building of Eggs

```
[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
```

- zc.recipe.egg:develop

The ":custom" sub-recipe of zc.recipe.egg provides for custom building of an egg from its source distribution. Sometimes a distribution has extension modules that need to be compiled with special options, such as the location of include files and libraries.

In this example, we have a part representing a non-Python library that needs to be built using the "./configure; make; make install" dance.

And then a part that uses that library to build a Python extension module. Notice how the second part references the location into which the first part was installed.

There is a ":develop" sub-recipe that is similar to ":custom", except that it operates upon develop-eggs that you may be working on. The resulting eggs are placed in the develop-eggs directory because the eggs are buildout specific.

Recipes: Generating Scripts for Eggs

```
[docreader]
recipe = zc.recipe.egg:scripts
eggs = zc.rst2
    codeblock
scripts = rst2=s5
extra-paths =
    ${Zope2_installation:location}/lib/python
    ${mxODBC-ZopeDA_installation:location}/lib/python
entry-points = ...
```

The zc.recipe.egg:scripts recipe scans those eggs specified with "eggs=" for entrypoints of the group "console_scripts" and, for each one found that appears in "scripts=", generates a script, usually in the "bin/" directory, that invokes it. If there is no "scripts=" option, all found entrypoints have a script generated for them.

The "eggs=" option also controls the set of distributions that will be "baked into" or activated within those specific scripts.

The "scripts=" option also permits aliasing a script, by providing an alternate name, after the second '=', for the script file itself. In this case the "rst2" entrypoint will be invoked from a script file named "s5".

The "extra-paths=" option provides directories to be added onto the sys.path for the particular scripts.

If a distribution referenced doesn't use setuptools, it may not have declared in its metadata any entry points. In that case, entry points can be specified in the recipe data, using the "entry-points=" option.

Recipes: How Eggs Are Activated Within Scripts

```
#!/usr/bin/python2.4

import sys
sys.path[0:0] = [
    '/var/tmp/buildout/eggs/zdaemon-2.0.1-py2.4.egg',
    '/var/tmp/buildout/eggs/setuptools-0.6c8-py2.4.egg',
    '/var/tmp/buildout/eggs/ZConfig-2.5.1-py2.4.egg',
    '/var/tmp/buildout/eggs/zc.zope3recipes-0.7.0-py2.4.egg',
]
import myegg
if __name__ == '__main__':
    myegg.foo_func()
```

This is an example of a script generated by buildout, showing how it bakes specific distributions into each script and then invokes code within the egg.

Notice how it differs from a script generated by setuptools, which is more declarative with __requires__ and version constraints, and defers to the entrypoint lookup mechanism.

Recipes: Baking Paths into Python Interpreter Scripts

- a Python interpreter under a custom filename
- that maps onto sys.path

  specific eggs and specific versions

```
[buildout]
parts = zcomponent

[zcomponent]
recipe = zc.recipe.egg:scripts
eggs = zope.component
interpreter = zprompt
```

> buildout provides for the generation of scripts to provide an interactive Python prompt with the specified eggs and their dependencies already activated, which is very useful for debugging specific programming scenarios.
>
> This is similar to a script, but uses the "interpreter=" option instead of the "scripts=" option.

Recipes: Pulling Subversion Files into Buildout

```
[clipart-svn]
recipe = iw.recipe.subversion
urls =
   http://svn.demo.com/clips/color/trunk  colorclips
   http://svn.demo.com/clips/grey/tags/v1_0_0  greyclips
```

```
[server]
recipe = tau.recipe.clipserver
images =
   ${clipart-svn:location}/colorclips
   ${clipart-svn:location}/greyclips
```

This is an example of how to pull into a buildout non-egg content stored under version control. The "iw.recipe.subversion" recipe accepts a list of URLs from which to checkout files and a destination directory name. Those directories are placed under the "parts/clipart-svn/" directory.

In the second part we see a server of some type that knows how to deliver those files to a client, and how it manages to reference those checked-out files.

Recipes: Using a Non-Egg Archive of Python Source

```
[mxODBC_installation]
recipe = hexagonit.recipe.download
url = http://egenix.com/egenix-mxodbc-1.0.10.linux.zip
```

```
[Zope2_prompt]
recipe      = zc.recipe.egg
interpreter = zprompt
eggs        = ${Zope2_instance:eggs}
extra-paths =
    ${Zope2_installation:location}/lib/python
    ${mxODBC-ZopeDA_installation:location}/lib/python
```

Sometimes a needed distribution comes as a zipfile of just .pyc files, particularly for a proprietary package such as mxODBC. They're not an actual egg, just a directory tree of files to be used as-is.

The "hexagonit.recipe.download" downloads archives in a variety of compression formats and unpacks them underneath the "parts/mxODBC_installation/" directory.

The second part can then reference this directory tree explicitly.

Use Case: Starting a New Project

```
$ mkdir newproj
$ cd newproj
$ buildout init

$ virtualenv --no-site-packages newproj
$ cd newproj
$ bin/easy_install zc.buildout
$ bin/buildout init

$ wget http://svn.zope.org/.../bootstrap/bootstrap.py
```

The first case shows how to start a new buildout, without using virtualenv.

It is suggested that every project that makes use of buildout come bundled with a bootstrap.py file to make it easier for the next developer to get started. bootstrap.py installs the setuptools and buildout distributions into the project directory.

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

http://svn.zope.org/checkout/zc.buildout/trunk/bootstrap/bootstrap.py

The second case show how to start a buildout within a virtualenv sandbox, with complete isolation from the system site-packages.

Use Case: Picking Up a Buildout Project

```
$ svn co svn://svn.zope.org/repos/.../Adder Adder
$ cd Adder
$ python bootstrap.py
$ bin/buildout
$ bin/zopectl fg

[buildout]
develop = .
```

This is an example of picking up a project that is **already** packaged for use with buildout.

This particular project already has a "develop=" line in its buildout.cfg that points to the setup.py in the project root. This means that, within the buildout, the package will already be a develop-egg, so that one can begin making changes to the source immediately and have it reflected in the runtime behavior without having to build/install it each time.

The actual URL for the example project is:

svn://svn.zope.org/repos/main/grokapps/Adder Adder

Use Case: Buildout Around Other Projects

```
$ virtualenv --no-site-packages myproj
$ cd myproj
$ bin/easy_install zc.buildout
$ mkdir src
$ svn co http://svn.demo.com/.../modulex/trunk src/modulex

[buildout]
develop = src
parts = myproj_eval

[myproj_eval]
recipe = zc.recipe.eggs:script
eggs = modulex
interpreter = aprompt
```

Often you run across a package or two that are not buildout-aware but you want to experiment with them inside a buildout sandbox.

This example sets up a sandbox and then brings the outside package into it under the "src/" directory. It may be a checkout or if this buildout is itself going to be stored under version control, the outside package can be a Subversion "extern" checkout. In this manner, checking out the buildout will pull down all the developmental pieces.

Within our buildout, we tell buildout to treat the outside package as a "develop-egg", and reference its distribution name as "modulex".

Use Case: Making Your Project Buildout-Aware

[buildout]
develop = .
parts = test

[test]
recipe = zc.recipe.testrunner
eggs = zc.ngi

To package your project so that it is buildout-aware, drop a minimal "buildout.cfg" file in the project root, next to the setup.py file.

A common usage of buildout is to support development of a single package along with running tests.

The "develop = ." says to find the "setup.py* file in the current directory and activate it as a development egg, so that I can make changes to it and re-run the tests as I work.

The value of a "develop=" option can be more than one directory, each of which has its own setup.py file.

The location, name and such of this package are provided in that setup.py file, and could be any number of Python packages arranged in any directory structure I choose.

To experiment with an example of this pattern of usage:

```
$ svn co http://svn.zope.org/zc.ngi/trunk/ zc.ngi
```

Use Case: Distributing Project as Egg

```
$ bin/buildout
$ bin/test
$ bin/buildout runsetup . sdist
$ bin/buildout runsetup . bdist_egg
```

This is a example of how, after developing and testing your project, buildout is used to push it up to a package index like the Cheeseshop.

Use Case: Production Deployment (non-RPMs)

```
[releaser]
recipe = zc.recipe.egg:script
eggs = zc.sourcerelease

$ bin/buildout-source-release file:///tmp/jtest buildout.cfg

$ tar xvzf jtest.tgz
$ cd jtest
$ bin/python2.5 install.py
```

This example uses the "zc.sourcerelease" recipe to cause an entire buildout, including dependencies, to be bundled into a tarball.

Note that the tarball does NOT include an actual Python interpreter, which must already be installed on the destination system to run the "install.py" script.

Use Case: Production Deployment (RPMs)

- tarball using zc.sourcerelease
- hand-create a .spec file that has:
    - Source: %{source}.tgz
    - a %prep that unpacks it
    - a %build that:
        - copies it under /opt/MYPROJ/
        - runs

          ```
          python /opt/MYPROJ/install.py bootstrap
          ```

        - runs

          ```
          python /opt/MYPROJ/install.py buildout:extensions=
          ```

    - a %files that grabs everything under /opt/MYPROJ/

This example shows how to produce a RPM for installation. It uses the "zc.sourcerelease" recipe to first produce a tarball, and then a hand-made RPM .spec file to turn that into an RPM.

A key part of this for an application like Zope or ZODB is separating a build into software parts and configuration parts.

The software parts are assembled when the source release/rpm is built.

The configuration is done post-install, by invoking scripts within the %build section of the RPM .spec file. The ZODB and Zope 3 recipes were specifically designed to support this separation.

For More Information

- Primary Buildout Home Page
- EuroPython 2007: Philipp v. Weitershausen
- Minitutorial: Introduction to zc.buildout
- Good documents in the zc.buildout egg itself.
- Questions?

- Primary Buildout Home Page

  http://www.zope.org/DevHome/Buildout
- EuroPython 2007: Philipp v. Weitershausen

  http://blip.tv/file/453645
- Minitutorial: Introduction to zc.buildout

  http://grok.zope.org/minitutorials/buildout.html