Buildout Documentation

Release 2.7

Jim Fulton

May 25, 2017

Contents

Contents:

CHAPTER 1

Buildout, an automation tool written in and extended with Python

Buildout is a tool for automating software assembly.

- Run build tools to build software.
- Apply software and templates to generate configuration files and scripts.
- Applicable to all software phases, from development to production deployment.
- Based on **core principles**:
  - Repeatability
  - Componentization
  - Automation

Repeatability

It's important that given a project configuration, two checkouts of the configuration in the same environment (operating system, Python version) should produce the same result, regardless of their history.

For example, is someone has been working on a project for a long time, and has committed their changes to a version control system, they should be able tell a colleague to check out their project and run buildout and the resulting build should have the same result as the build in the original working area.

Componentization

We believe that software should be self contained, or at least, that it should be possible. The tools for satisfying the software responsibilities should largely reside within the software project itself.

Some examples:

- Software services should include tools for monitoring them. Operations, including monitoring is a software responsibility, because the creators of the software are the ones who know best how to assess whether it is operating correctly.

  It should be possible, when deploying production software, for the software to configure the monitoring system to monitor the software.

- Software should provide facilities to automate its configuration. It shouldn't be necessary for people to create separate configuration whether it be in development or deployment (or stages in between).

Automation

Software deployment should be highly automated. It should be possible to checkout a project and with a single simple command (or two), get a working system. This is necessary to achieve the goals of repeatability and componentization and generally not to waste people's time.

Learning more

Learn more:

- *Getting started*
- *Topics*
- *Reference*

Additional resources

Issue tracker https://github.com/buildout/buildout/issues

Mailing list You can ask questions on the Python Distutils-SIG mailing list, https://mail.python.org/mailman/listinfo/distutils-sig

Github repository https://github.com/buildout/buildout

Contributing Join the buildout-development google group, https://groups.google.com/forum/#!forum/buildout-development to discuss ideas and submit pull requests against the buildout repository.

CHAPTER 2

Getting started with Buildout

Note: In the Buildout documentation, we'll use the word *buildout* to refer to:

- The Buildout software

  We'll capitalize the word when we do this.

- A particular use of Buildout, a directory having a Buildout configuration file.

  We'll use lower case to refer to these.

- A `buildout` section in a Buildout configuration (in a particular buildout).

  We'll use a lowercase fixed-width font for these.

First steps

The easiest way to install Buildout is with pip:

```
pip install zc.buildout
```

To use Buildout, you need to provide a Buildout configuration. Here is a minimal configuration:

```
[buildout]
parts =
```

A minimal (and useless) Buildout configuration has a `buildout` section with a parts option. If we run Buildout:

```
buildout
```

Four directories are created:

bin A directory to hold executables.

develop-eggs A directory to hold develop egg links. More about these later.

eggs A directory that hold installed packages in egg[1] format.

parts A directory that provides a default location for installed parts.

Buildout configuration files use an INI syntax[2]. Configuration is arranged in sections, beginning with section names in square brackets. Section options are names, followed by equal signs, followed by values. Values may be continued over multiple lines as long as the continuation lines start with whitespace.

Buildout is all about building things and the things to be built are specified using *parts*. The parts to be built are listed in the `parts` option. For each part, there must be a section with the same name that specifies the software to build the part and provides parameters to control how the part is built.

Installing software

In this tutorial, we're going to install a simple web server. The details of the server aren't important. It just provides a useful example that illustrates a number of ways that Buildout can make things easier.

We'll start by adding a part to install the server software. We'll update our Buildout configuration to add a `bobo` part:

```
[buildout]
parts = bobo

[bobo]
recipe = zc.recipe.egg
eggs = bobo
```

We added the part name, `bobo` to the `parts` option in the `buildout` section. We also added a `bobo` section with two options:

recipe The standard `recipe` option names the software component that will implement the part. The value is a Python distribution requirement, as would be used with `pip`. In this case, we've specified zc.recipe.egg which is the name of a Python project that provides a number of recipe implementations.

eggs A list of distribution requirements, one per line.[3] (The name of this option is unfortunate, because the values are requirements, not egg names.) Listed requirements are installed, along with their dependencies. In addition, any scripts provided by the listed requirements (but not their dependencies) are installed in the `bin` directory.

If we run this:

```
buildout
```

Then a number of things will happen:

- `zc.recipe.egg` will be downloaded and installed in your `eggs` directory.
- `bobo` and its dependencies will be downloaded and installed. (bobo is a small Python database server.)

  After this, the eggs directory will look something like:

  ```
  $ ls -l eggs
  total 0
  ```

[1] You may have heard bad things about eggs. This stems in part from the way that eggs were applied to regular Python installs. We think eggs, which were inspired by jar files, when used as an installation format, are a good fit for Buildout's goals. Learn more in the topic on *Buildout and packaging*.

[2] Buildout uses a variation (fork) of standard `ConfigParser` module and follows (mostly) the same parsing rules.

[3] Requirements can have whitespace characters as in `bobo <3`, so they're separated by newlines.

```
drwxr-xr-x  4 jim  staff  136 Feb 23 09:01 WebOb-1.7.1-py2.7.egg
drwxr-xr-x  9 jim  staff  306 Feb 23 09:10 bobo-2.3.0-py2.7.egg
```

- A `bobo` script will be installed in the `bin` directory:

```
$ ls -l bin
total 8
-rwxr-xr-x  1 jim  staff  391 Feb 23 09:10 bobo
```

This script is used to run a bobo server.

Generating configuration and custom scripts

The `bobo` program doesn't daemonize itself. Rather, it's meant to be used with a dedicated daemonizer like zdaemon or supervisord. We'll use a recipe to set up zdaemon. Our Buildout configuration becomes:

```
[buildout]
parts = bobo server

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

[server]
recipe = zc.zdaemonrecipe
program =
  ${buildout:bin-directory}/bobo
    --static /=${buildout:directory}
    --port 8200
```

Here we've added a new `server` part that uses `zc.zdaemonrecipe`. We used a `program` option to define what program should be run. There are a couple of interesting things to note about this option:

- We used *variable substitutions*:

  `${buildout:directory}` Expands to the full path of the buildout directory.

  `${buildout:bin-directory}` Expands to the full path of the buildout's `bin` directory.

  Variable substitution provides a way to access Buildout settings and share information between parts and avoid repetition.

  See the *reference* to see what buildout settings are available.

- We spread the program over multiple lines. A configuration value can be spread over multiple lines as long as the continuation lines begin with whitespace.

  The interpretation of a value is up to the recipe that uses it. The `zc.zdaemonrecipe` recipe combines the program value into a single line.

If we run Buildout:

```
buildout
```

- The `zc.zdaemonrecipe` recipe will be downloaded and installed in the eggs directory.
- A `server` script is added to the `bin` directory. This script is generated by the recipe. It can be run like:

```
bin/server start
```

to start a server and:

```
bin/server stop
```

to stop it. The script references a zdaemon configuration file generated by the recipe in `parts/server/zdaemon.conf`.

- A zdaemon configuration script is generated in `parts/server/zdaemon.conf` that looks something like:

```
<runner>
  daemon on
  directory /Users/jim/t/0214/parts/server
  program /Users/jim/t/0214/bin/bobo --static /=/Users/jim/t/0214 --port 8200
  socket-name /Users/jim/t/0214/parts/server/zdaemon.sock
  transcript /Users/jim/t/0214/parts/server/transcript.log
</runner>

<eventlog>
  <logfile>
    path /Users/jim/t/0214/parts/server/transcript.log
  </logfile>
</eventlog>
```

The **details aren't important**, other than the fact that the configuration file reflects part options and the actual buildout location.

Version control

In this example, the only file that needs to be checked into version control is the configuration file, `buildout.cfg`. Everything else is generated. Someone else could check out the project, and get the same result[4].

More than just a package installer

The example shown above illustrates how Buildout is more than just a package installer, like `pip`. Using Buildout recipes, we can install custom scripts and configuration files, and much more. For example, we could use configure and make to install non-Python software from source, we could run JavaScript builders, or do anything else that can be automated with Python.

Buildout is a simple automation framework. There are hundreds of recipes to choose from[5] and *writing new recipes is easy*.

Repeatability

A major goal of Buildout is to provide repeatability. But what does this mean exactly?

[4] This assumes the same environment and that dependencies haven't changed. We'll explain further in the section on repeatability.

[5] You can list Buildout-related software, consisting mostly of Buildout recipes, using the Framework :: Buildout classifier search. These results miss recipes that don't provide classifier meta data. Generally you can find a recipe for a task by searching the name of the task and the "recipe" in the package index.

> If two buildouts with the same configuration are built in the same environments at the same time, they should produce the same result, regardless of their build history.

That definition is rather dense. Let's look at the pieces:

Buildout environment

A Buildout environment includes the operating system and the Python installation it's run with. The more a buildout depends on its environment, the more variation is likely between builds.

If a Python installation is shared, packages installed by one application affect other applications, including buildouts. This can lead to unexpected errors. This is why it's recommended to use a virtual environment or a "clean python" built from source with no third-party packages installed[6].

To limit dependence on the operating system, people sometimes install libraries or even database servers as Buildout parts.

Modern Linux container technology (e.g. Docker) makes it a lot easier to control the environment. If you develop entirely with respect to a particular container image, you can have repeatability with respect to that image, which is usually good enough because the environment, defined by the image, is itself repeatable and unshared with other applications.

Python requirement versions

Another potential source of variation is the versions of Python dependencies used.

Newest versions

If you don't specify versions, Buildout will always try to get the most recent version of everything it installs. This is a major reason that Buildout can be slow. It checks for new versions every time it runs. It does this to satisfy the repeatability requirement above. If it didn't do this, then an older buildout would likely have different versions of Python packages than newer buildouts.

To speed things up, you can use the `-N` Buildout option to tell Buildout to *not* check for newer versions of Python requirements:

```
buildout -N
```

This relaxes repeatability, but with little risk if there was a recent run without this option.

Pinned versions

You can also pin required versions in two ways. You can specify them where you list them, as in:

```
[bobo]
recipe = zc.recipe.egg
eggs = bobo <5.0
```

In this example, we've requested a version of bobo less than 5.0.

The more common way to pin version is using a `versions` section:

[6] It's a little hypocritical to recommend installing Buildout into an otherwise clean environment, which is why Buildout provides a *bootstrapping mechanism* which allows setting up a buildout without having to contaminate a virtual environment or clean Python install.)

```
[buildout]
parts = bobo server

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

[server]
recipe = zc.zdaemonrecipe
program =
  ${buildout:bin-directory}/bobo
    --static /=${buildout:directory}
    --port 8200

[versions]
bobo = 2.3.0
```

Larger projects may need to pin many versions, so it's common to put versions in their own file:

```
[buildout]
extends = versions.cfg
parts = bobo server

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

[server]
recipe = zc.zdaemonrecipe
program =
  ${buildout:bin-directory}/bobo
    --static /=${buildout:directory}
    --port 8200
```

Here, we've used the Buildout `extends` option to say that configurations should be read from the named file (or files) and that configuration in the current file should override configuration in the extended files. To continue the example, our `versions.cfg` file might look like:

```
[versions]
bobo = 2.3.0
```

We can use the `update-versions-file` option to ask Buildout to maintain our `versions.cfg` file for us:

```
[buildout]
extends = versions.cfg
show-picked-versions = true
update-versions-file = versions.cfg

parts = bobo server

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

[server]
recipe = zc.zdaemonrecipe
program =
  ${buildout:bin-directory}/bobo
```

```
    --static /=${buildout:directory}
    --port 8200
```

With `update-versions-file`, whenever Buildout gets the newest version for a requirement (subject to requirement constraints), it appends the version to the named file, along with a comment saying when and why the requirement is installed. If you later want to upgrade a dependency, just edit this file with the new version, or to remove the entry altogether and Buildout will add a new entry the next time it runs.

We also used the `show-picked-versions` to tell Buildout to tell us when it got (picked) the newest version of a requirement.

When versions are pinned, Buildout doesn't look for new versions of the requirements, which can speed buildouts quite a bit. In fact, The `-N` option doesn't provide any speedup for projects whose requirement versions are all pinned.

When should you pin versions?

The rule of thumb is that you should pin versions for a whole system, such as an application or service. You do this because after integration tests, you want to be sure that you can reproduce the tested configuration.

You shouldn't pin versions for a component, such as a library, because doing so inhibits the ability for users of your component to integrate it with their dependencies, which may overlap with yours. If you know that your component only works a range of versions of some dependency, the express the range in your project requirements. Don't require specific versions.

Unpinning versions

You can unpin a version by just removing it (or commenting it out of) your `versions` section.

You can also unpin a version by setting the version to an empty string:

```
[versions]
ZEO =
```

In an extending configuration (`buildout.cfg` in the example above), or *on the buildout command line*.

You might do this if pins are shared between projects and you want to unpin a requirement for one of the projects, or want to remove a pin while using a requirement in *development mode*.

Buildout versions and automatic upgrade

In the interest of repeatability, Buildout can upgrade itself or its dependencies to use the newest versions or downgrade to respect pinned versions. This only happens if you run Buildout from a buildout's own `bin` directory.

We can use Buildout's `bootstrap` command to install a local buildout script:

```
buildout bootstrap
```

Then, if the installed script is used:

```
bin/buildout
```

Then Buildout will upgrade or downgrade to be consistent with version requirements. See the *bootstrapping topic* to learn more about bootstrapping.

Python development projects

A very common Buildout use case is to manage the development of a library or main part of an application written in Python. Buildout facilitates this with the `develop` option:

```
[buildout]
develop = .
...
```

The `develop` option takes one more more paths to project setup.py files or, more commonly, directories containing them. Buildout then creates "develop eggs"[7] for the corresponding projects.

With develop eggs, you can modify the sources and the modified sources are reflected in future Python runs (or after reloads).

For libraries that you plan to distribute using the Python packaging infrastructure, You'll need to write a setup file, because it's needed to generate a distribution.

If you're writing an application that won't be distributed as a separate Python distribution, writing a setup script can feel like overkill, but it's useful for:

- naming your project, so you can refer to it like any Python requirement in your Buildout configuration, and for
- specifying the requirements your application code uses, separate from requirements your buildout might have.

Fortunately, an application setup script can be minimal. Here's an example:

```
from setuptools import setup
setup(name='main', install_requires = ['bobo', 'six'])
```

We suggest copying and modifying the example above, using it as boilerplate. As is probably clear, the setup arguments used:

name The name of your application. This is the name you'll use in Buildout configuration where you want to refer to application code.

install_requires A list of requirement strings for Python distributions your application depends on directly.

A *minimal*[8] development Buildout configuration for a project with a setup script like the one above might look something like this:

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

[py]
recipe = zc.recipe.egg
eggs = main
interpreter = py
```

There's a new option, `interpreter`, which names an *interpreter* script to be generated. An interpreter script[9] mimics a Python interpreter with its path set to include the requirements specified in the eggs option and their (transitive) dependencies. We can run the interpreter:

[7] pip calls these "editable" installs.

[8] A more typical development buildout will include at least a part to specify a test runner. A development buildout might define other support parts, like JavaScript builders, database servers, development web-servers and so on.

[9] An interpreter script is similar to the `bin/python` program included in a virtual environment, except that it's lighter weight and has exactly the packages listed in the `eggs` option and their dependencies, plus whatever comes from the Python environment.

```
bin/py
```

To get an interactive Python prompt, or you can run a script with it:

```
bin/py somescript.py
```

If you need to work on multiple interdependent projects at the same time, you can name multiple directories in the `develop` option, typically pointing to multiple check outs. A popular Buildout extension, mr.developer, automates this process.

Where to go from here?

This depends on what you want to do. We suggest perusing the *topics* based on your needs and interest.

The *reference* section can give you important details, as well as let you know about features not touched on here.

CHAPTER 3

Buildout Topics

History, motivation, and Python packaging

Isolation from environment

In the early 2000s, Zope Corporation was helping customers build Zope-based applications. A major difficulty was helping people deploy the applications in their own environments, which varied not just between customers, but also between customer machines. The customer environments, including operating system versions, libraries and Python modules installed were not well defined and subject to change over time.

We realized that we needed to insulate ourselves from the customer environments[1] to have any chance of predictable success.

We decided to provide our own Python builds into which we installed the application. These were automated with make. Customers would receive tar files, expand them and run `make`. We referred to these as "build outs".

Python

Later, as the applications we were building became more complex, some of us wanted to be able to use Python, rather than make, to automate deployments. In 2005, we created an internal prototype that used builds defined using ConfigParser-formatted configuration files. File sections described things to be built and there were a few built-in build recipes and eventually facilities for implementing custom recipes.

By this time, we were hosting most of the applications we were building, but we were still building Python and critical libraries ourselves as part of deployment, to isolate ourselves from system Python and library packages.

After several months of successful experience with the prototype, we decided to build what became zc.buildout based on our experience, making a recipe framework a main idea.

[1] Ultimately, we moved to a model where we hosted software ourselves for customers, because we needed control over operation, as well as installation and upgrades, and because with the technology of the time, we still weren't able to sufficiently insulate ourselves from the customers' environments.

Buildout and packaging

Around this time, setuptools and easy_install were released, providing automated download and installation of Python packages *and their dependencies*. Because we built large applications, this was something we'd wanted for some time and had even begun building a package manager ourselves. Part of the rationale for creating a new Buildout version, beyond the initial prototype, was to take advantage of the additional automation that setuptools promised.

Initially, we tried to leverage the `easy_install` command[2], but the goals of `easy_install` and Buildout were at odds. `easy_install` sought to make it easy for humans to install and upgrade packages manually, in an ad hoc manner. While it installed dependencies, it didn't upgrade them. It didn't provide ways of managing an installation as a whole. Buildout, on the other hand, was all about automation and repeatability.

To achieve Buildout's goals, it was necessary to interact with setuptools at a much lower level and to write quite a bit more packaging logic than planned.

Eggs

Setuptools defined a packaging format, eggs, used for package distribution and installation. Their design was based on Java jar files, which bundle software together with supporting resources, including meta-data.

Eggs presented a number of challenges, and have a bad reputation as a result:

- As an installation format:
  - They needed to be added to the Python path. The `easy_install` command did this by generating complex .pth files. This often led to hard to diagnose bugs and frustration.
  - By default, eggs were installed as zip files. Software development tools used by most Python developers[3] made working with zip files difficult. Also, importing from zip files was much slower on Unix-like systems.
- As a distribution format, eggs names carry insufficient meta data to distinguish incompatible builds of extensions on Linux.

Buildout uses eggs very differently

Script generation

When Buildout generates a script, it's usually generating a wrapper script. Python package distributions define scripts in two ways, via entry points, or as scripts in a separate `scripts` area of the distribution.

Entry points are meta data that define a main function to be run when a user invokes a generated script. Entry points make it easier to control how a script is run, including what version of Python is used and the Python path. Initially, Buildout only supported installing entry-point-based scripts.

The older way of packaging scripts is harder to deal with, because Buildout has to edit scripts to use the correct Python installation and to set the Python path.

Buildout doesn't use `.pth` files. Instead, when Buildout generates a script, it generates a Python path that names the eggs needed, and only the eggs needed, for a particular script based on its requirements. When Buildout is run, scripts are regenerated if versions of any of their dependencies change. Scripts defined by different parts can use different versions, because they have different Python paths. Changing a version used often requires only updating the path generated for a script.

[2] The `zc.buildout.easy_install` module started out as a thin wrapper around the `easy_install` command. Although it has (almost) nothing to do with the `easy_install` command today, its name has remained, because it provides some public APIs.

[3] Java tools have no problem working with zip files, because of the prominence of jar files, which like eggs, use zip format.

Buildout's approach to assembling applications should be familiar to anyone who's worked with Java applications, which are assembled the same way, using jars and class paths.

Buildout uses eggs almost exclusively as an **installation** format[4], in a way that leverages eggs' strengths. Eggs provide Buildout with the ability to efficiently control which dependencies a script uses, providing repeatability and predictability.

Staying DRY with value substitutions, extending, and macros

A buildout configuration is a collection of sections, each holding a collection of options. It's common for option values to be repeated across options. For examples, many file-path options might start with common path prefixes. Configurations that include clients and servers might share server-address options. This topic presents various ways you can reuse option values without repeating yourself.

Value substitutions

When supplying values in a configuration, you can include values from other options using the syntax:

```
${SECTION:OPTION}
```

For example: `${buildout:directory}` refers to the value of the `directory` option in the in the `buildout` section of the configuration. The value of the referenced option will be substituted for the referencing text.

You can simplify references to options in the current section by omitting the section name. If we wanted to use the `buildout directory` option from within the `buildout` section itself, we could use `${:directory}`. This convenience is especially useful in *macros*, which we'll discuss later in this topic.

There's a special value that's also useful in macros, named `_buildout_section_name_`, which has the name of the current section. We'll show how this is used when we discuss *macros*.

Default and computed option values

Many sections have option values that can be used in substitutions without being defined in a configuration.

The `buildout` section, where settings for the buildout as a whole are provided has many default option values. For example, the directory where scripts are installed is configurable and the value is available as `${buildout:bin-directory}`. See the *Buildout options reference* for a complete list of Buildout options that can be used in substitutions.

Many recipes also have options that have defaults or that are computed and are available for substitutions.

Sources of configuration options

Configuration option values can come from a number of sources (in increasing precedence):

software default values These are defined by buildout and recipe sources.

user default values These are set in *per-user default configuration files* and override default values.

options from one or more configuration files These override user defaults and each other, as described below.

option assignments in the *buildout command line* These override configuration-file options.

[4] Buildout always unzips eggs into ordinary directories, by default.

Extending configuration files

The *extends* option in a `buildout` section can be used to extend one or more configuration files. There are a number of applications for this. For example, common options for a set of projects might be kept in a common base configuration. A production buildout could extend a development buildout, or they could both extend a common base.

The option values in the extending configuration file override those in the files being extended. If multiple configurations are named in the `extends` option (separated by whitespace), then the configurations are processed in order from left/top to right/bottom, with the later (right/bottom) configurations overriding earlier (left/top) ones. For example, in:

```
extends = base1.cfg base2.cfg
          base3.cfg
```

The options in the configuration using the extends option override the options in `base3.cfg`, which override the options in `base2.cfg`, which override the options in `base1.cfg`.

Base configurations may be extended multiple times. For example, in the example above, `base1.cfg` might, itself, extend `base3.cfg`, or they might both extend a common base configuration. Of course, cycles are not allowed.

Configurations may be named with URLs in the `extends` option, in which case they may be downloaded from remote servers. See *The extends-cache buildout option*.

When a relative path is used in an extends option, it's interpreted relative to the path of the extending configuration.

Conditional configuration sections

Sometimes, you need different configuration in different environments (different operating systems, or different versions of Python). To make this easier, you can define environment-specific options by providing conditional sections:

```
[ctl]
suffix =

[ctl:windows]
suffix = .bat
```

In this tiny example, we've defined a `ctl:suffix` option that's `.bat` on Windows and an empty string elsewhere.

A conditional section has a colon and then a Python expression after the name. If the Python expression result is true, the section options from the section are included. If the value is false, the section is ignored.

Some things to note:

- If there is no exception, then options from the section are included.
- Sections and options can be repeated. If an option is repeated, the last value is used. In the example above, on Windows, the second `suffix` option overrides the first. If the order of the sections was reversed, the conditional section would have no effect.

In addition to the normal built-ins, the expression has access to global variables that make common cases short and descriptive as shown below

Name	Value
sys	`sys` module
os	`os` module
platform	`platform` module
re	`re` module
python2	True if running Python 2
python3	True if running Python 3
python26	True if running Python 2.6
python27	True if running Python 2.7
python32	True if running Python 3.2
python33	True if running Python 3.3
python34	True if running Python 3.4
python35	True if running Python 3.5
python36	True if running Python 3.6
sys_version	`sys.version.lower()`
pypy	True if running PyPy
jython	True if running Jython
iron	True if running Iron Python
cpython	True if not running PyPy, Jython, or Iron Python
sys_platform	`str(sys.platform).lower()`
linux	True if running on Linux
windows	True if running on Windows
cygwin	True if running on Cygwin
solaris	True if running on Solaris
macosx	True if running on Mac OS X
posix	True if running on a POSIX-compatible system
bits32	True if running on a 32-bit system.
bits64	True if running on a 64-bit system.
little_endian	True if running on a little-endian system
big_endian	True if running on a big-endian system

Expressions must not contain either the `#` or the `;` character.

User-default configuration

A per-user default configuration may be defined in the `default.cfg` file in the `.buildout` subdirectory of a user's home directory (`~/.buildout/default.cfg` on Mac OS and Linux). This configuration is typically used to set up a shared egg or cache directory, as in:

```
[buildout]
eggs-directory = ~/.buildout/eggs
download-cache = ~/.buildout/download-cache
abi-tag-eggs = true
```

See the section on *optimizing buildouts with shared eggs and download caches* for an explanation of the options used in the example above.

Merging, rather than overriding values

Normally, values in extending configurations override values in extended configurations by replacing them, but it's also possible to augment or trim overridden values. If += is used rather than =, the overriding option value is appended to the original. So, for example if we have a base configuration, `buildout.cfg`:

```
[buildout]
parts =
  py
  test
  server
...
```

And a production configuration `prod.cfg`, we can add another part, `monitor`, like this:

```
[buildout]
extends = buildout.cfg
parts += monitor
...
```

In this example, we didn't have to repeat (or necessarily know) the base parts to add the `monitor` part.

We can also subtract values using `-=`, so if we wanted to exclude the `test` part in production:

```
[buildout]
extends = buildout.cfg
parts += monitor
parts -= test
...
```

Something to keep in mind is that this works by *lines*. The `+=` form adds the lines in the new data to the lines of the old. Similarly, `-=` removes *lines* in the overriding option from the original *lines*. This is a bit delicate. In the example above, we were careful to put the base values on separate lines, in anticipation of using `-=`.

Merging values also works with option assignments provided via the *buildout command line*. For example, if you want to temporarily use a *development version* of another project, you can augment the buildout *develop option* on the command-line when running buildout:

```
buildout develop+=/path/to/other/project
```

Although, if you've pinned the version of that project, you'll need to *unpin it*, which you can also do on the command-line:

```
buildout develop+=/path/to/other/project versions:projectname=
```

Extending sections using macros

We can extend other sections in a configuration as macros by naming then using the `<` option. For example, perhaps we have to create multiple server processes that listen on different ports. We might have a base `server` section, and some sections that use it as a macro:

```
[server]
recipe = zc.zdaemonrecipe
port = 8080
program =
  ${buildout:bin-directory}/serve
     --port ${:port}
     --name ${:_buildout_section_name_}

[server1]
<= server
port = 8081
```

```
[server2]
<= server
port = 8082
```

In the example above, the `server1` and `server2` sections use the `server` section, getting its `recipe` and `program` options. The resulting configuration is equivalent to:

```
[server]
recipe = zc.zdaemonrecipe
port = 8080
program =
  ${buildout:bin-directory}/serve
     --port ${:port}
     --name ${:_buildout_section_name_}

[server1]
recipe = zc.zdaemonrecipe
port = 8081
program =
  ${buildout:bin-directory}/serve
     --port ${:port}
     --name ${:_buildout_section_name_}

[server2]
recipe = zc.zdaemonrecipe
port = 8082
program =
  ${buildout:bin-directory}/serve
     --port ${:port}
     --name ${:_buildout_section_name_}
```

Value substitutions in the base section are applied after its application as a macro, so the substitutions are applied using data from the sections that used the macro (using the < option).

You can extend multiple sections by listing them in the < option on separate lines, as in:

```
[server2]
<= server
   monitored
port = 8082
```

If multiple sections are extended, they're processed in order, with later ones taking precedence. In the example above, if both `server` and `monitored` provided an option, then the value from `monitored` would be used.

A section that's used as a macro can extend another section.

Automatic installation of part dependencies

Buildout parts are requested by the `parts` option of the `buildout` section, but a buildout may install additional parts that are dependencies of the named parts. For example, in

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

```
[server]
=> app
recipe = zc.zdaemonrecipe
program = ${buildout:bin-directory}/app ${config:location}

[app]
recipe = zc.recipe.egg
eggs = myapp

[config]
recipe = zc.recipe.deployment:configuration
text = port 8080
```

the `server` part depends on the `app` part to install the server software and on the `config` part to provide the server configuration.

The `config` part will be installed before the `server` part because it's referenced in a value substitution. The value substitution makes the `config` part a dependency of the `server` part.

The `server` part has the line:

```
=> app
```

This line[1], uses a feature that's **new in zc.buildout 2.9**. It declares that the `app` part is a dependency of the `server` part. The server part doesn't use any information from the `app` part, so it has to declare the dependency explicitly. It could have declared both dependencies explicitly:

```
=> app config
```

Dependency part selection serves separation of concerns. The buildout `parts` option reflects the requirements of a buildout as a whole. If a named part depends on another part, that's the concern of the named part, not of the buildout itself.

Optimizing buildouts with shared eggs and download caches

Most users should have this *user-default configuration* containing option settings that make Buildout work better:

```
[buildout]
eggs-directory = ~/.buildout/eggs
download-cache = ~/.buildout/download-cache
abi-tag-eggs = true
```

You might be wondering why these settings aren't the default, if they're recommended for everyone. They probably *should* be the default, and perhaps will be in version 3 of buildout. Making them the default now might break existing buildouts.

Shared eggs directory

You can save a lot of time and disk space by sharing eggs between buildouts. You can do this by setting the `eggs-directory` option, as shown above. This will override the default value for this option which puts eggs

[1] The => syntax is a convenience. It's based on the mathematical symbol for implication. It's a short hand for:

```
<part-dependencies> = app
```

Multiple parts may be listed and spread over multiple lines, as long as continuation lines are indented.

in the `eggs` buildout subdirectory. By sharing eggs, you can avoid reinstalling the same popular packages in each and every buildout that uses them.

ABI tag eggs

If you use a shared eggs directory, it's a good idea to set the `abi-tag-eggs` option to `true`. This causes eggs to be segregated by ABI tag. This has two advantages:

1. If you alternate between Python implementations (PyPy versus C Python) or between build configurations (normal versus debug), ABI tagging eggs will avoid mixing incompatible eggs.
2. ABI tagging eggs makes Buildout run faster. Because ABI tags include Python version information, eggs for different Python versions are kept separate, causing the shared eggs directory for a given Python version to be smaller, making it faster to search for installed eggs.

Download cache

When buildout installs distributions, it has to download them first. Specifying a `download-cache` option in your *user-default configuration* causes the download to be cached. This can be helpful when multiple installations might be performed for a source distribution.

Some recipes download information. For example, a number of recipes download non-Python source archives and user configure, and make to install them. Most of these recipes can leverage a download cache to avoid downloading the same information over and over.

Bootstrapping

Bootstrapping a buildout gives its own `buildout` script, independent of its Python environment. There are 2 reasons you might use this:

Enable automatic Buildout upgrade (or downgrade). If the `buildout` script is local to the buildout, then Buildout will check for newest versions of Buildout and its dependencies that are consistent with any version pins and install any that are different, in which case, it restarts to use the new versions.

Doing automatic upgrades allows buildouts to be more independent of their environments and more repeatable.

Using a local `buildout` script may be necessary for a project that pins the version of Buildout itself and the pinned version is different from the version in the Python environment.

Avoid modifying the python environment. From a philosophical point of view, Buildout has tried to be isolated from its environment, and requiring the Python environment to be modified, by installing Buildout, was inconsistent.

Before virtualenv existed, it might not have been possible to modify the environment without building Python from source.

Unfortunately, doing this requires *using a bootstrap script*.

Local bootstrapping using the `bootstrap` command

You can use the *bootstrap command* of a `buildout` script installed in your Python environment to boostrap the buildout in the current directory:

```
buildout bootstrap
```

If you have any other buildouts that have local `buildout` scripts, you can use their `buildout` scripts:

```
/path/to/some/buildout/bin/buildout bootstrap
```

In this case, the buildout being bootstrapped will have the same Python environment as the buildout that was used to bootstrap it.

Using a bootstrapping script

If you download:

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

And then run it:

```
python bootstrap-buildout.py
```

It will download the software needed to run Buildout and install it in the current directory.

This has been the traditional approach to bootstrapping Buildout. It was the best approach for a long time because the `pip` and `easy_install` commands usually weren't available. In the early days, if `easy_install` was installed, it was likely to have an incompatible version of setuptools, because Buildout and setuptools were evolving rapidly, sometimes in lock step.

This approach fails from time to time, due to changes in setuptools or the package index and has been a source of breakage when automated systems depended on it.

It's also possible that this approach will stop being supported. Buildout's bootstrapping script relies on setuptools' bootstrap script, which was used to bootstrap `easy_install`. Now that pip is ubiquitous, there's no reason to bootstrap `easy_install` and setuptools' bootstrapping script exists solely to support Buildout. At some point, that may become too much of a maintenance burden, and there may not be Buildout volunteers motivated to create a new bootstrapping solution.

Bootstrapping requires a `buildout.cfg`, `init` creates one

Normally, when bootstrapping, the local directory must have a `buildout.cfg` file.

If you don't have one, you can use the *init command* instead:

```
buildout init
```

This can be used with the bootstrapping script as well:

```
python bootstrap-buildout.py init
```

This creates an empty Buildout configuration:

```
[buildout]
parts =
```

If you know you're going to use some packages, you can supply requirements on the command line after `init`:

```
buildout init bobo six
```

In which case it will generate and run a buildout that uses them. The command above would generate a buildout configuration file:

```
[buildout]
parts = py

[py]
recipe = zc.recipe.egg
interpreter = py
eggs =
  bobo
  six
```

This can provide an easy way to experiment with a package without adding it to your Python environment or creating a virtualenv.

Buildout extensions

Buildout has a mechanism that can be used to extend it in low-level and often experimental ways. Use the `extensions` option in the `buildout` section to use an existing extension. For example, the buildout.wheel extension provides support for Python wheels:

```
[buildout]
extensions = buildout.wheel
...
```

Some other examples of extensions can be found in the standard package index.

Writing Buildout recipes

There are two kinds of buildout recipes: *install* and *uninstall*. Install recipes are by far the most common. Uninstall recipes are very rarely needed because most install recipes add files and directories that can be removed by Buildout.

Install recipes

Install recipes are typically implemented with classes and have 3 important parts:

- A constructor (typically, `__init__`) initializes a recipe object.

  The constructor plays a very important role, because it may update the configuration data it's passed, making information available to other parts and controlling whether a part will need to be re-installed.

  The constructor performs the first of two phases of recipe work, the second phase being the responsibility of either the `install` or `update` methods.

- The `install` method installs new parts.

- The `update` method updates previously installed parts. It's often an empty method or an alias for `install`.

Buildout phases

When buildout is run using the default *install command*, parts are installed in several phases:

1. Parts are initialized by calling their recipe constructors. This may cause part configuration options to be updated, as described below.

2. Part options are compared to part options from previous runs[1].
   - Parts from previous runs that are no longer part of the buildout are uninstalled.
   - Parts from previous runs whose options have changed are also uninstalled.
3. Parts are either installed or updated.

   `install()` is called on new parts or old parts that were uninstalled.

   `update()` is called on old parts whose configuration hasn't changed.

Initialization phase: the constructor

The constructor is passed 3 arguments:

`buildout` The buildout configuration

The buildout configuration is a mapping from section names to sections. Sections are mappings from option names to values. The buildout configuration allows the recipe to access configuration data in much the same way as configuration files use *value substitutions*.

`name` The name of the section the recipe was used for

`options` The part options

This is a mapping object and may be written to to save derived configuration, to provide information for use by other part recipes, or for *value substitutions*.

Nothing should be installed in this phase.

If the part being installed isn't new, options after calling the constructor are compared to the options from the previous Buildout run. If they are different, then the part will be uninstalled and then re-installed by calling the `install` method, otherwise, the `update` method will be called.

Install or update phase

In this phase, `install()` or `update()` is called, depending on whether the part is new or has new configuration.

This is the phase in which the part does its work. In addition to affecting changes, these methods have some responsibilities that can be a little delicate:

- If an error is raised, it is the responsibility of the recipe to undo any partial changes.
- If the recipe created any files or directories, the recipe should return their paths. Doing so allows Buildout to take care of removing them if the part is uninstalled, making a separate uninstall recipe unnecessary.

To make these responsibilities easier to cope with, the `option` object passed to the constructor has a helper function, `created`. It should be passed one or more paths just before they are created and returns a list of all of the paths passed as well as any earlier paths created. If an exception is raised, any files or directories created will be removed automatically. When the recipe returns, it can just return the result of calling `created()` with no arguments.

Example: configuration from template recipe

In this example, we'll show a recipe that creates a configuration file based on a configuration string computed using value substitutions[2]. A sample usage:

[1] Configuration data from previous runs are saved in a buildout's installed database, *typically saved in* a generated `.installed.cfg` file.

[2] There are a variety of template recipes that provide different features, like using template files and supporting various template engines. Don't re-use the example here.

```
[buildout]
develop = src
parts = server

[config]
recipe = democonfigrecipe
port = 8080
contents =
   <zeo>
     address ${:port}
   </zeo>
   <mappingstorage>
   </mappingstorage>

[server]
recipe = zc.zdaemonrecipe
program = runzeo -C ${config:path}
```

Some things to note about this example:

- The `config` part uses the recipe whose source code we'll show below. It has a `port` option, which it uses in its `contents` option. It could as easily have used options from other sections.
- The `server` part uses `${config:path}` to get the path to the configuration file generated by the `config` part. The `path` option value will be computed by the recipe for use in other parts, as we've seen here.
- We didn't have to list the `config` part in the buildout `parts` option. It's *added automatically* by virtue of its use in the `server` part.
- We used the `develop` option to specify a `src` directory containing our recipe. This allows us to use the recipe locally without having to build a distribution file.

If we were to run this buildout, a `parts/config` file would be generated:

```
<zeo>
  address 8080
</zeo>
<mappingstorage>
</mappingstorage>
```

as would a zdaemon configuration file, `parts/server/zdaemon.conf`, like:

```
<runner>
  daemon on
  directory /sample/parts/server
  program runzeo -C /sample/parts/config
  socket-name /sample/parts/server/zdaemon.sock
  transcript /sample/parts/server/transcript.log
</runner>

<eventlog>
  <logfile>
    path /sample/parts/server/transcript.log
  </logfile>
</eventlog>
```

Here's the recipe source, `src/democonfigrecipe.py`:

```
import os

class Recipe:

    def __init__(self, buildout, name, options):
        options['path'] = os.path.join(
           buildout['buildout']['parts-directory'],
           name,
           )
        self.options = options

    def install(self):
        self.options.created(self.options['path'])
        with open(self.options['path'], 'w') as f:
            f.write(self.options['contents'])
        return self.options.created()

    update = install
```

The constructor computes the `path` option. This is then available for use by the `server` part above. It's also used later in the `install` method. We use `buildout['buildout']['parts-directory']` to get the buildout parts directory. This is equivalent to using `${buildout:parts-directory}` in the configuration. The parts directory is the standard place for recipes to create files or directories. If a recipe uses the parts directory, it should create only one file or directory whose name is the part name, which is passed in as the `name` argument to the constructor.

The constructor saves the options so that the data and `created` method are available in `install`.

The `install` method calls the option object's `created` method **before** creating a file. The order is important, because if the file-creation fails partially, the file will be removed automatically. The recipe itself doesn't need an exception handler. The configuration file is then written out. Finally, the `created` method is called again[3] to return the list of created files (one, in this case).

The `update` method is just an alias for the `install` method. We could have used an empty method, however running install again makes sure the file contents are as expected, overwriting manual changes, if any.

Like the `install` method, the `update` method returns any paths it created. These are merged with values returned by the `install` or `update` in previous runs.

For this recipe to be usable, we need to make it available as a distribution[4], so we need to create a setup script, `src/setup.py`:

```
from setuptools import setup

setup(
    name='democonfigrecipe',
    version='0.1.0',
    py_modules = ['democonfigrecipe'],
    entry_points = {"zc.buildout": ["default=democonfigrecipe:Recipe"]},
    )
```

The setup script specifies a name and version and lists the module to be included.

The setup script also uses an `entry_points` option. Entry points provide a miniature component systems for setuptools. A project can supply named components of given types. In the example above, the type of the component is `"zc.buildout"`, which is the type used for Buildout recipes. A single components named `default` is provided.

[3] Unfortunately, returning the result of calling `created()` is boilerplate. Future versions of buildout won't require this return.

[4] Even though we aren't distributing the recipe in this example, we still need to create a *develop distribution* so that Buildout can find the recipe and its meta data.

The component is named as the `Recipe` attribute of the `democonfigrecipe` module. When you specify a recipe in the `recipe` option, you name a recipe requirement, which names a project, and optionally provide a recipe name. The default name is `default`. Most recipe projects provide a single recipe component named `default`.

If we removed the `server` part from the configuration, the two configuration files would be removed, because Buildout recorded their paths and would remove them automatically.

Uninstall recipes

Uninstall recipes are very rarely needed, because most recipes just install files and Buildout can handle those automatically.

An uninstall recipe is just a function that takes a name and an options mapping. One of the few packages with an uninstall recipe is zc.recipe.rhrc. The `uninstall` function there provides the uninstall recipe. Here's a **highly simplified** version:

```
def uninstall(name, options):
    os.system('/sbin/chkconfig --del ' + name)
```

This was used with a recipe that installed services on older Red Hat Linux servers. When the part was uninstalled, it needed to run `/sbin/chkconfig` to disable the service. Uninstall recipes don't need to return anything.

Like install recipes, uninstall recipes need to be registered using entry points, using the type `zc.buildout.uninstall` as can be seen in the zc.recipe.rhrc setup script.

User interaction: logging and UserError

Recipes communicate to users through logging and errors. Recipes can log information using the Python logging library and messages will be displayed according to buildout's *verbosity setting*.

Errors that a user can potentially correct should be reported by raising `zc.buildout.UserError` exceptions with error messages as arguments.

Buildout will display these as user errors, rather than printing a trace back.

Testing recipes

The recipe API is fairly simple and standard unit-testing approaches can be used. We'll use a helper class, `zc.buildout.testing.Buildout`[5] to provide a minimal buildout environment.

Let's write a test for our configuration recipe. We need to verify that:

- The recipe generates a `path` option.
- The recipe generates a file in the correct place.
- The recipe returns the path it created from `install`.

We create a `testdemoconfigrecipe.py` file containing our tests:

```
import os
import shutil
import tempfile
import unittest
import zc.buildout.testing
```

[5] We're relying on some refinements made to the helper class in zc.buildout 2.9.

```
class RecipeTests(unittest.TestCase):

    def setUp(self):
        self.here = os.getcwd()
        self.tmp = tempfile.mkdtemp(prefix='testdemoconfigrecipe-')
        os.chdir(self.tmp)
        self.buildout = buildout = zc.buildout.testing.Buildout()
        self.config = 'some config text\n'
        buildout['config'] = dict(contents=self.config)
        import democonfigrecipe
        self.recipe = democonfigrecipe.Recipe(
            buildout, 'config', buildout['config'])

    def tearDown(self):
        os.chdir(self.here)
        shutil.rmtree(self.tmp)

    def test_path_option(self):
        buildout = self.buildout
        self.assertEqual(os.path.join(buildout['buildout']['parts-directory'],
                                      'config'),
                         buildout['config']['path'])

    def test_install(self):
        buildout = self.buildout
        self.assertEqual(self.recipe.install(), [buildout['config']['path']])
        with open(buildout['config']['path']) as f:
            self.assertEqual(self.config, f.read())

if __name__ == '__main__':
    unittest.main()
```

In the `setUp` method, we created a temporary directory and changed to it. This is useful to make sure we have a clean working directory. We clean it up in the `tearDown` method.

Our test uses `zc.buildout` so that we can use the `zc.buildout.testing.Buildout` helper class. We did this so we'd have a more realistic environment, but of course, we could have stubbed this out ourselves. Because we're using `zc.buildout` in our test, we'll add it as a test dependency in our setup script:

```
from setuptools import setup

setup(
    name='democonfigrecipe',
    version='0.1.0',
    py_modules = ['democonfigrecipe', 'testdemoconfigrecipe'],
    entry_points = {"zc.buildout": ["default=democonfigrecipe:Recipe"]},
    extras_require = dict(test=['zc.buildout >=2.9']),
    )
```

Here, we defined an "extra" requirement. These are additional dependencies needed to support optional features. In this case, we're providing an optional `test` feature. (We specified that we want at least version 2.9, because we're depending on some testing-support refinements that were added in zc.buildout 2.9.0.)

We'll write a development buildout to run our tests with:

```
[buildout]
develop = src
parts = py
```

```
[py]
recipe = zc.recipe.egg
eggs = democonfigrecipe [test]
interpreter = py
```

Running Buildout with this gives is an interpreter script that we can run our tests with. The script will make sure that `zc.buildout` and our recipe can be imported.

To run our tests:

```
bin/py src/testdemoconfigrecipe.py
```

In this example, we've tried to keep things simple and as free from external requirements as possible.

More realistically:

- You'd probably arrange your recipe in a Python package rather than as a top-level module and a top-level testing module.
- You might use a test runner like nose or pytest. There are recipes that can help set this up. We just used the test runner built into `unittest`.

`zc.buildout.testing` reference

The zc.buildout.testing module provides an API that can be used when writing recipe tests. This API is documented below.

Many of the functions documented below take a path argument as multiple arguments. These are joined using `os.path.join`. This is more convenient than having to call os.path.join before calling the functions.

`Buildout()` A class you can use to create buildout and sections objects in your tests

This is a subclass of the main object used to run buildout. Its constructor takes no arguments. You can add data to it by setting section names to dictionaries:

```
buildout['config'] = dict(contents=self.config)
```

To get an options object to pass to your recipe, just ask for it back:

```
buildout['config']
```

See the *recipe example* above.

`cat(*path)` Display the contents of a file. The file path is provided as one or more strings, to be joined with os.path.join.

On Windows, if the file doesn't exist, the function will try adding a '-script.py' suffix. This helps to work around a difference in script generation on windows.

`clear_here()` Remove all files and directories in the current working directory.

New in buildout 2.9

`eqs(got, *expected)` Compare a collection with a collection given as multiple arguments.

Both collections are converted to and compared as sets. If the sets are the same, then no output is returned, otherwise a tuple of extras is returned, so, for example:

```
>>> eqs([1, 2, 3], 3, 1, 2)
>>> eqs([1, 2, 3], 1, 2, 4) == ({3}, {4})
True
```

New in buildout 2.9

ls(*path) List the contents of a directory. The directory path is provided as one or more strings, to be joined with os.path.join.

mkdir(*path) Create a directory. The directory path is provided as one or more strings, to be joined with os.path.join.

system(command, input='') Execute a system command with the given input passed to the command's standard input. The output (error and regular output combined into a single string) from the command is returned.

read(*path) Read text from a file at the given path. The file path is provided as one or more strings, to be joined with os.path.join.

If no path is given, the `'out'` is used.

New in buildout 2.9

remove(*path) Remove a directory or file. The path is provided as one or more strings, to be joined with os.path.join.

rmdir(*path) Remove a directory. The directory path is provided as one or more strings, to be joined with os.path.join.

run_buildout_in_process(command='buildout') Run Buildout in a multiprocessing.Process. The command is must be a buildout command string, starting with 'buildout'. You can provide additional arguments, as in `'buildout -v'`.

Some extra options are added to the command to prevent network access when running the command. Any distribution the buildout needs must already be available for import. So, for example, if you want to use some recipe, include it in your rest dependencies.

All output from the buildout run is captured in the file named `out`.

This is useful for integration tests or tests of recipes that interact intimately with buildout or other recipes.

New in buildout 2.9

write(*path_and_contents) Create a file. The file path is provided as one or more strings, to be joined with os.path.join. The last argument is the file contents.

Documenting your recipe

Please, don't use your doctests to document your recipe. (We did that a lot and it didn't turn out well.) Just write straightforward documentation that explains to users how to use your recipe.

If you have examples, however, considering testing them using manuel. You can see examples of how to do that by looking at the source of this topic. Otherwise, it's very easy to end up with mistakes in your examples.

Meta-recipes

Buildout recipes provide reusable Python modules for common configuration tasks. The most widely used recipes tend to provide low-level functions, like installing eggs or software distributions, creating configuration files, and so on. The normal recipe framework is fairly well suited to building these general components.

Full-blown applications may require many, often tens, of parts. Defining the many parts that make up an application can be tedious and often entails a lot of repetition. Buildout provides a number of mechanisms to avoid repetition, including merging of configuration files and macros, but these, while useful to an extent, don't scale very well. Buildout isn't and shouldn't be a programming language.

Meta-recipes allow us to bring Python to bear to provide higher-level abstractions for buildouts.

A meta-recipe is a regular Python recipe that primarily operates by creating parts. A meta recipe isn't merely a high level recipe. It's a recipe that defers most or all of it's work to lower-level recipes by manipulating the buildout database.

A presentation at PyCon 2011 described early work with meta recipes.

A simple meta-recipe example

Let's look at a fairly simple meta-recipe example. First, consider a buildout configuration that builds a database deployment:

```
[buildout]
parts = ctl pack

[deployment]
recipe = zc.recipe.deployment
name = ample
user = zope

[ctl]
recipe = zc.recipe.rhrc
deployment = deployment
chkconfig = 345 99 10
parts = main

[main]
recipe = zc.zodbrecipes:server
deployment = deployment
address = 8100
path = /var/databases/ample/main.fs
zeo.conf =
   <zeo>
      address ${:address}
   </zeo>
   %import zc.zlibstorage
   <zlibstorage>
     <filestorage>
        path ${:path}
     </filestorage>
   </zlibstorage>

[pack]
recipe = zc.recipe.deployment:crontab
deployment = deployment
times = 1 2 * * 6
command = ${buildout:bin-directory}/zeopack -d3 -t00 ${main:address}
```

This buildout doesn't build software. Rather it builds configuration for deploying a database configuration using already-deployed software. For the purpose of this document, however, the details are totally unimportant.

Rather than crafting the configuration above every time, we can write a meta-recipe that crafts it for us. We'll use our meta-recipe as follows:

```
[buildout]
parts = ample

[ample]
recipe = com.example.ample:db
path = /var/databases/ample/main.fs
```

The idea here is that the meta recipe allows us to specify the minimal information necessary. A meta-recipe often automates policies and assumptions that are application and organization dependent. The example above assumes, for example, that we want to pack to 3 days in the past on Saturdays.

So now, let's see the meta recipe that automates this:

```
class Recipe:

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

        buildout.parse('''
            [deployment]
            recipe = zc.recipe.deployment
            name = %s
            user = zope
            ''' % name)

        buildout['main'] = dict(
            recipe = 'zc.zodbrecipes:server',
            deployment = 'deployment',
            address = 8100,
            path = options['path'],
            **{
              'zeo.conf': '''
                <zeo>
                  address ${:address}
                </zeo>

                %import zc.zlibstorage

                <zlibstorage>
                  <filestorage>
                    path ${:path}
                  </filestorage>
                </zlibstorage>
                '''}
            )

        buildout.parse('''
            [pack]
            recipe = zc.recipe.deployment:crontab
            deployment = deployment
            times = 1 2 * * 6
            command =
              ${buildout:bin-directory}/zeopack -d3 -t00 ${main:address}

            [ctl]
            recipe = zc.recipe.rhrc
            deployment = deployment
```

```
            chkconfig = 345 99 10
            parts = main
            ''')

    def install(self):
        pass

    update = install
```

The meta recipe just adds parts to the buildout. It does this by setting items and and calling the `parse` method. The `parse` method just takes a string in buildout configuration syntax. It's useful when we want to add static, or nearly static part data. The setting items syntax is useful when we have non-trivial computation for part data.

The order that we add parts is important. When adding a part, any string substitutions and other dependencies are evaluated, so the referenced parts must be defined first. This is why, for example, the `pack` part is added after the `main` part.

Note that the meta recipe supplied an integer for one of the options. In addition to strings, it's legal to supply integer values.

There are a few things to note about this example:

- The install and update methods are empty.

  While not required, this is a very common pattern for meta recipes. Most meta recipes, simply invoke other recipes.

- Setting a buildout item or calling parse, adds any sections with recipes as parts.
- An exception will be raised if a section already exists.

Testing

Now, let's test our meta recipe. We'll test it without actually running buildout. Rather, we'll use a specialized buildout provided by the zc.buildout.testing module.

```
>>> import zc.buildout.testing
>>> buildout = zc.buildout.testing.Buildout()
```

The testing buildout is intended to be passed to recipes being tested:

```
>>> _ = Recipe(buildout, 'ample', dict(path='/var/databases/ample/main.fs'))
```

After running the recipe, we should see the buildout database populated by the recipe:

```
>>> buildout.print_options(base_path='/sample-buildout')
[ctl]
chkconfig = 345 99 10
deployment = deployment
parts = main
recipe = zc.recipe.rhrc
[deployment]
name = ample
recipe = zc.recipe.deployment
user = zope
[main]
address = 8100
deployment = deployment
```

```
path = /var/databases/ample/main.fs
recipe = zc.zodbrecipes:server
zeo.conf =

                  <zeo>
                    address 8100
                  </zeo>

                  %import zc.zlibstorage

                  <zlibstorage>
                    <filestorage>
                      path /var/databases/ample/main.fs
                    </filestorage>
                  </zlibstorage>

[pack]
command = /sample-buildout/bin/zeopack -d3 -t00 8100
deployment = deployment
recipe = zc.recipe.deployment:crontab
times = 1 2 * * 6
```

CHAPTER 4

Reference

The Buildout command line

A Buildout execution is of the form:

```
buildout [buildout-options] [assignments] [command [command arguments]]
```

Assignments take the form `section:option=value` and override (or augment) options in configuration files. For example, to pin a version of ZEO you could use `versions:ZEO=4.3.1`. The section defaults to the `buildout` section. So, for example: `parts=test` sets the `buildout` section `parts` option.

Command-line assignments can use += and −= to *merge values with existing values*

Buildout command-line options

`-c config_file` Specify the path (or URL) to the buildout configuration file to be used. This defaults to the file named `buildout.cfg` in the current working directory.

`-D` Debug errors. If an error occurs, then the post-mortem debugger will be started. This is especially useful for debugging recipe problems.

`-h, --help` Print basic usage information and exit.

`-N` Run in *non-newest mode*. This is equivalent to the command-line assignment `newest=false`.

`-q` Decrease the level of verbosity. This option can be used multiple times.

Using a single `-q` suppresses normal output, but still shows warnings and errors.

Doubling the option `-qq` (or equivalently `-q -q`) suppresses normal output and warnings.

Using the option more than twice suppresses errors, which is a bad idea.

`-t socket_timeout` Specify the socket timeout in seconds. See the *socket-timeout option* for details.

`-U` Don't use *user-default configuration*.

-v Increase the level of verbosity. This option can be used multiple times.

At the default verbosity, buildout prints messages about significant activities. It also prints warning and error messages.

At the next, "verbose", level (`-v`), it prints much more information. In particular, buildout will show when and why it's installing specific distribution versions.

At the next, "debugging", level, `-vv` (or equivalently `-v -v`), buildout prints low-level debugging information, including a listing of all configuration options, including: default options, computed options and the results of *value substitutions* and *macros*.

Using this option more than twice has no effect.

--version Print buildout version number and exit.

Buildout commands

annotate

Display the buildout configuration options, including their values and where they came from. Try it!

```
buildout annotate
```

Increase the verbosity of the output to display all steps that compute the final values used by buildout.

```
buildout -v annotate
```

Pass one or more section names as arguments to display annotation only for the given sections.

```
buildout annotate versions
```

bootstrap

Install a local `bootstrap` script. The `bootstrap` command doesn't take any arguments.

See *Bootstrapping* for information on why you might want to do this.

init [requirements]

Generate a Buildout configuration file and bootstrap the resulting buildout.

If requirements are given, the generated configuration will have a `py` part that uses the `zc.recipe.egg` recipe to install the requirements and generate an interpreter script that can import them. It then runs the resulting buildout.

See *Bootstrapping* for examples.

install

Install the parts specified in the buildout configuration. This is the default command if no command is specified.

setup PATH SETUP-COMMANDS

Run a setuptools-based setup script to build a distribution.

The path must be the path of a setup script or of a directory containing one named `setup.py`. For example, to create a source distribution using a setup script in the current directory:

```
buildout setup . sdist
```

This command is useful when the Python environment you're using doesn't have setuptools installed. Normally today, setuptools *is* installed and you can just run setup scripts that use setuptools directly.

Note that if you want to build and upload a package to the standard package index you should consider using zest.releaser, which automates many aspects of software release including checking meta data, building releases and making version-control tags.

Buildout configuration options

The standard buildout options are shown below. Values of options with defaults shown can be used in *value substitutions*.

abi-tag-eggs A flag (true/false) indicating whether the eggs directory should be divided into subdirectories by ABI tag. This may be useful if you use multiple Python builds with different build options or different Python implementations. It's especially useful if you switch back and forth between PyPy and C Python.

allow-hosts, default: '*' Specify which hosts (as globs) you're willing to download distributions from when following *dependency links*.

allow-picked-versions, default: 'true' Indicate whether it should be possible to install requirements whose *versions aren't pinned <pinned-versions>*.

bin-directory, default: bin The directory where generated scripts should be installed. If this is a relative path, it's evaluated relative to the buildout directory.

develop One or more (whitespace-separated) paths to distutils setup scripts or (more commonly) directories containing setup scripts named `setup.py`.

See: *Python development projects*.

develop-eggs-directory, default: 'develop-eggs' The directory where *develop eggs* should be installed. If this is a relative path, it's evaluated relative to the buildout directory.

directory, default: directory containing top-level buildout configuration The top of the buildout. Other directories specified (or defaulting) with relative paths are created relative to this directory.

download-cache An optional directory in which to cache downloads. Python distributions are cached in the `dist` subdirectory of this directory. Recipes may also cache downloads in this directory, or in a subdirectory.

This is often set in a *User-default configuration* to share a cache between buildouts. See the section on *Optimizing buildouts with shared eggs and download caches*.

If the value is a relative path and doesn't contain value substitutions, it's interpreted relative to the directory containing the configuration file that defined the value. (If it contains value substitutions, and the result is a relative path, then it will be interpreted relative to the buildout directory.)

eggs-directory, default: 'eggs' The directory where *eggs* are installed.

This is often set in a *User-default configuration* to share eggs between buildouts. See the section on *Optimizing buildouts with shared eggs and download caches*.

If the value is a relative path and doesn't contain value substitutions, it's interpreted relative to the directory containing the configuration file that defined the value. (If it contains value substitutions, and the result is a relative path, then it will be interpreted relative to the buildout directory.)

executable, default: sys.executable, read-only The full path to the Python executable used to run the buildout.

extends The names, separated by whitespace, of one or more configurations that the configuration containing the `extends` option should *extend*. The names may be file paths, or URLs. If they are relative paths, they are interpreted relative to the configuration containing the `extends` option.

extends-cache An optional directory to cache remote configurations in. Remote configuration is configuration specified using a URL in an *extends option* or as the argument to the *-C buildout command-line option*. How the extends-cache behaves depends on the buildout mode:

Mode	Behavior
install-from-cache or *offline*	Configuration is retrieved from cache if possible. If configuration isn't cached, the buildout fails.
non-newest	Configuration is retrieved from cache if possible. If configuration isn't cached, then it is downloaded and saved in the cache.
Default (*newest*)	Configuration is downloaded and saved in the cache, even if it is already cached, and the previously cached value is replaced.

If the value is a relative path and doesn't contain value substitutions, it's interpreted relative to the directory containing the configuration file that defined the value. (If it contains value substitutions, and the result is a relative path, then it will be interpreted relative to the buildout directory.)

find-links, default: '' Extra locations to search for distributions to download.

These may be file paths or URLs. These may name individual distributions or directories containing distributions. Subdirectories aren't searched.

index An alternate index location.

This can be a local directory name or an URL. It can be a flat collection of distributions, but should be a "simple" index, with subdirectories for distribution project names containing distributions for those projects.

If this isn't set, then `https://pypi.python.org/simple/` is used.

install-from-cache, default: 'false' Enable install-from-cache mode.

In install-from-cache mode, no network requests should be made.

It's a responsibility of recipes to adhere to this. Recipes that would need to download files may use the *download cache*.

The original purpose of the install-from-cache mode was to support source-distribution of buildouts that could be built without making network requests (mostly for security reasons).

This mode may only be used if a *download-cache* is specified.

installed, default: '.installed.cfg' The name of the file used to store information about what's installed.

Buildout keeps information about what's been installed so it can remove files created by parts that are removed and so it knows whether to update or install new parts from scratch.

If this is a relative path, then it's interpreted relative to the buildout directory.

log-format, default: '' Format to use for log messages.

If `log-format` is blank, the default, Buildout will use the format:

```
%(message)s
```

for its own messages, and:

```
%(name)s: %(message)s
```

for the root logger[1].

If `log-format` is non-blank, then it will be used for the root logger[1] (and for Buildout's messages).

newest, default: 'true' If true, check for newer distributions. If false, then only look for distributions when installed distributions don't satisfy requirements.

The goal of non-newest mode is to speed Buildout runs by avoiding network requests.

offline, default: 'false' If true, then offline mode is enabled.

> **Warning:** Offline mode is deprecated.
>
> Its purpose has evolved over time and the end result doesn't make much sense, but it is retained (indefinitely) for backward compatibility.
>
> If you think you want an offline mode, you probably want either the *non-newest mode* or the *install-from-cache mode* instead.

In offline mode, no network requests should be made. It's the responsibility of recipes to adhere to this. Recipes that would need to download files may use the *download cache*.

No distributions are installed in offline mode. If installed distributions don't satisfy requirements, the the buildout will error in offline mode.

parts-directory, default: 'parts' The directory where generated part artifacts should be installed. If this is a relative path, it's evaluated relative to the buildout directory.

If a recipe creates a file or directory, it will normally create it in the parts directory with a name that's the same as the part name.

prefer-final, default: 'true' If true, then only final distribution releases will be used unless no final distributions satisfy requirements.

show-picked-versions, default: 'false' If true, when Buildout finds a newest distribution for a requirement that *wasn't pinned <pinned-versions>*, it will print lines it would write to a versions configuration if the *update-versions-file* option was used.

socket-timeout, default: '' Specify a socket timeout[2], in seconds, to use when downloading distributions and other artifacts. If non-blank, the value must be a positive non-zero integer. If left blank, the socket timeout is system dependent.

This may be useful if downloads are attempted from very slow sources.

update-versions-file, default: '' If non-blank, this is the name of a file to write versions to when selecting a distribution for a requirement whose version *wasn't pinned <pinned-versions>*. This file, typically `versions.cfg`, should end with a `versions` section (or whatever name is specified by the `versions` option).

use-dependency-links, default: true Distribution meta-data may include URLs, called dependency links, of additional locations to search for distribution dependencies. If this option is set to `false`, then these URLs will be ignored.

versions, default 'versions' The name of a section that contains *version pins*.

[1] Generally, the root logger format is used for all messages unless it is overridden by a lower-level logger.
[2] This timeout reflects how long to wait on individual socket operations. A slow request may take much longer than this timeout.

Configuration file syntax

Buildout configurations use an INI file format.

A configuration is a collection of named sections containing named options.

Section names

A section begins with a section and and, optionally, a condition in square braces (`[` and `]`).

A name can consist of any characters other than whitespace, square braces, curly braces (`{` or `}`), pound signs (`#`), colons (`:`) or semi-colons (`;`). The name may be surrounded by leading and trailing whitespace, which is ignored.

An optional condition is separated from the name by a colon and is a Python expression. It may not contain a pound sign or semi-colon. See the section on *conditional sections* for an example and more details.

A comment, preceded by a pound sign or semicolon may follow the section name, as in:

```
[buildout] # This is the buildout section
```

Options

Options are specified with an option name followed by an equal sign and a value:

```
parts = py
```

Option names may have any characters other than whitespace, square braces, curly braces, equal signs, or colons. There may be and usually is whitespace between the name and the equal sign and the name and equal sign must be on the same line. Names starting with < are reserved for Buildout's use.

Option values may contain any characters. A consequence of this is that there can't be comments in option values.

Option values may be continued on multiple lines, and may contain blank lines:

```
parts = py

        test
```

Whitespace in option values

Trailing whitespace is stripped from each line in an option value. Leading and trailing blank lines are stripped from option values.

Handling of leading whitespace and blank lines internal to values depend on whether there is data on the first line (containing the option name).

data on the first line Leading whitespace is stripped and blank lines are omitted.

The resulting option value in the example above is:

```
py
test
```

no data on the first line Internal blank lines are retained and common leading white space is stripped.

For example, the value of the option:

```
code =
    if x == 1:
        y = 2 # a comment

        return
```

is:

```
if x == 1:
    y = 2 # a comment

    return
```

Special "implication" syntax for the <part-dependencies> option

An exception to the normal option syntax is the use of => as a short-hand for the <part-dependencies> option:

```
=> part1 part2
   part3
```

This is equivalent to:

```
<part-dependencies> = part1 part2
   part3
```

and declares that the named parts are dependencies of the part in which this option appears.

Comments and blank lines

Lines beginning with pound signs or semi-colons (# or ;) are comments:

```
# This is a comment
; This too
```

As mentioned earlier, comments can also appear after section names.

Blank lines are ignored unless they're within option values that only have data on continuation lines.