Tunic Documentation

Release 1.3.0

TSH Labs

Dec 01, 2017

Contents

A Python library for deploying code on remote servers.

Tunic is designed so that you can make use of as much or as little of its functionality as you'd like, the choice is yours.

It only requires the Fabric library as a dependency and can be installed from the Python Package Index (PyPI) using the pip tool like so.

```
pip install tunic
```

You could then make use of it in your deploy process like so.

```
from fabric.api import task
from tunic.api import get_release_id, ReleaseManager, VirtualEnvInstallation

APP_BASE = '/srv/www/myapp'

@task
def deploy():
    stop_my_app()
    release = get_release_id()

    installer = VirtualEnvInstaller(APP_BASE, ['myapp'])
    release_manager = ReleaseManager(APP_BASE)

    installer.install(release)
    release_manager.set_current_release(release)

    start_my_app()
```

The above snippet is just the start, take a look around the code base for more methods that can save you work in your deploy process.

CHAPTER 1

Contents

1.1 Design

This section will go over some assumptions made by the Tunic library, the general design of the library, and some things to keep in mind when using it.

1.1.1 Purpose

Tunic is meant to supplement your existing Fabric based deploy process, not to replace it or the Fabric library / tool. Tunic is **not** an abstraction layer for Fabric. It's merely meant to let you avoid writing the same thing over and over again.

1.1.2 Directory structure

Tunic doesn't care about where on your server you deploy to. However, it expects that deployments are organized in a particular way and that each deployment of your project is named in a particular way. The following are **required** for Tunic functionality to work correctly.

For this example, let's assume you're deploying your project to `/srv/www/myapp`.

- This base directory for your project should be writable as the user or group that deploys are being performed by.
- The directory structure under `/srv/www/myapp` must organized as follows.

```
/
+-- srv
    +-- www
        +-- myapp
            |-- releases
            |   +-- 20141105123145-0.2.0
            |   +-- 20141002231442-0.1.0
            +-- current -> releases/20141105123145-0.2.0
```

- The `releases` directory must be under your project base directory and be writeable by the user or group that deploys are being performed by.
- Each deploy under the `releases` directory must be named starting with a timestamp corresponding to when the deploy was done. The timestamp component should built with the largest period of time first (the current year), followed by each smaller component down to the second. The timezone used to generate this timestamp component doesn't matter as long as you always use the same one.

  The name for each deploy will be generated for you (in UTC) if you use the *tunic.core.get_release_id()* function. This is required to ensure that we can determine the time deploys were done relative to each other.
- `current` must be a symlink to the active deployment in the `releases` directory. This symlink will be created for you automatically if you use the *tunic.core.ReleaseManager.set_current_release()* method as part of your deploy process.

If you've used Capistrano, this structure should feel pretty familiar ;)

1.1.3 Dependence on Fabric

Since Tunic is built on Fabric, it inherits the following behavior.

- Output from commands run by methods in `tunic.api` ends up being displayed just like output from commands run by Fabric. This can be changed through the use of Fabric context managers.
- Since Tunic requires Fabric and Fabric doesn't support Python 3 (yet), Tunic won't work with Python 3 at this time.
- Since Tunic makes heavy use of Fabric and Fabric is not thread safe, Tunic is also not thread safe.

1.1.4 Versions

Tunic uses semantic versioning of the form `major.minor.patch`. All backwards incompatible changes after version `1.0.0` will increment the major version number. All backwards incompatible changes prior to version `1.0.0` will increment the minor version number.

Since this is a Python project, only the subset of the semantic versioning spec that is compatible with PEP-440 will be used.

1.2 Usage

Tunic tries to reduce the amount of code you need to write for your deploy process. The major components of Tunic are designed so that they can be used together – or not. If you find a component doesn't fit well with your deploy process, don't use it!

This guide will go over each of the distinct components of the Tunic library and how to use them individually. Then we'll look at how to use them all together as part of the same deploy process.

Note: All functionality meant to be used by consumers of the Tunic library is exposed in the `tunic.api` module. Anything not contained in this module should be considered private and subject to change.

1.2.1 get_releases_path and get_current_path

These are the most basic parts of the Tunic library. Given a path to the base directory of your project, they'll give you paths to components of the directory structure that the rest of the Tunic library expects. They are code to enforce assumptions made by the library.

Below is an example of using the *tunic.core.get_releases_path()* method to find all releases of a particular project.

```
from fabric.api import run
from tunic.api import get_releases_path

APP_BASE = '/srv/www/myapp'

def get_myapp_releases():
    """Get all releases of the MyApp project as a list."""
    release_path = get_releases_path(APP_BASE)
    releases = run('ls -1r ' + releases_path)
    return releases.split()
```

Below is an example of using the *tunic.core.get_current_path()* method to find the deployment that is being actively served.

```
from fabric.api import run
from tunic.api import get_current_path

APP_BASE = '/srv/www/myapp'

def get_myapp_current():
    """Get the active deployment of MyApp."""
    current_path = get_current_path(APP_BASE)
    current = run('readlink ' + current_path)
    return current
```

1.2.2 get_release_id

The *tunic.core.get_release_id()* method is responsible for generating a unique name for each deployment of a project. It generates a timestamp based name, with an optional version component. The timestamp component is built with the largest period of time first (the current year), followed by each smaller component down to the second (similar to ISO 8601 dates).

The purpose of generating a name for a deployment in this manor is to allow us to keep track of when each deployment was made. Thus we are able to easily figure out which deployments are the oldest, which particular deployment came before the 'current' one, etc.

Below is an example of using the *tunic.core.get_release_id()* method to set up a new deployment.

```
import os.path
from fabric.api import run
from tunic.api import get_release_id

APP_BASE = '/srv/www/myapp'

def create_new_release(version):
    """Create a new release virtualenv and return the path."""
    releases = os.path.join(APP_BASE, 'releases')     # '/srv/www/myapp/releases'
    release_id = get_release_id(version)              # '20140928223929-1.4.1'
```

```
    new_release = os.path.join(releases, release_id) # '/srv/www/myapp/releases/
↪20140928223929-1.4.1'
    run('virtualenv ' + new_release)
    return new_release
```

1.2.3 ReleaseManager

The *tunic.core.ReleaseManager* class is responsible for inspecting and manipulating previous deployments and the current deployment on a remote server.

In order manipulate deployments like this, the ReleaseManager requires that they are organized as described in *Design*.

Below is an example of getting all available deployments (current and past) from a server.

```
from tunic.api import ReleaseManager

APP_BASE = '/srv/www/myapp'

def get_all_releases():
    release_manager = ReleaseManager(APP_BASE)
    return release_manager.get_releases()
```

Below is an example of creating a "rollback" task in Fabric for switching to the previous deployment of your project that uses the *tunic.core.ReleaseManager.get_previous_release()* and *tunic.core.ReleaseManager.set_current_release()* methods.

```
from fabric.api import task, warn
from tunic.api import ReleaseManager

APP_BASE = '/srv/www/myapp'

@task
def rollback():
    release_manager = ReleaseManager(APP_BASE)
    previous = release_manager.get_previous_release()

    if previous is None:
        warn("No previous release, can't rollback!")
        return

    release_manager.set_current_release(previous)
```

The ReleaseManager can also remove old deployments. To do this, you must have named the deployments with a timestamp based prefix. If you've used *tunic.core.get_release_id()* to name your deployments, this is handled for you.

```
from fabric.api import task
from tunic.api import ReleaseManager

APP_BASE = '/srv/www/myapp'

@task
def cleanup(deployments_to_keep=5):
    release_manager = ReleaseManager(APP_BASE)
    release_manager.cleanup(keep=deployments_to_keep)
```

1.2.4 ProjectSetup

The *tunic.core.ProjectSetup* class is responsible for creating the required directory structure for a project and ensuring that permissions and ownership is consistent before and after a deploy.

The ProjectSetup class will create directories that are organized as described in *Design*.

The ProjectSetup class typically uses sudo for creation of the directory structure and changing of ownership and permissions of the project deploys. If the user doing the deploy will not have sudo permissions, the methods can be passed the `use_sudo=False` keyword argument to instruct them not to use sudo, but instead use the Fabric `run` command. When using the `run` command, the *tunic.core.ProjectSetup.set_permissions()* method will not attempt to change the owner of the project deploys, only the permissions.

As with most parts of the Tunic library, use of this class for project deploy process is optional. For example, if you use a configuration management system (such as Puppet, Chef, Ansible, etc.) to ensure the correct directories exist and have correct permissions on any server you deploy to, using the ProjectSetup class may not be needed.

An example of creating the required directory structure and ensuring permissions before and after a deploy, assuming the user doing the deploy has sudo permissions.

```
from fabric.api import task
from tunic.api import ProjectSetup
from .myapp import install_project

APP_BASE = '/srv/www/myapp'

@task
def deploy():
    setup = ProjectSetup(APP_BASE)
    setup.setup_directories()
    setup.set_permissions('root:www')

    install_project()

    setup.set_permissions('root:www')
```

1.2.5 LocalArtifactTransfer

The *tunic.install.LocalArtifactTransfer* class allows you to transfer locally built artifacts to a remote server and clean them up afterwards in the scope of a Python context manager. With more advanced deploy setups that use a centralized artifact repository, this class isn't usually needed. However, if you don't have a centralized repository, it can save you a bit of work.

An example of using it to transfer locally built artifacts is below.

```
from fabric.api import task
from tunic.api import LocalArtifactTransfer
from .myapp import install_project_from_artifacts

LOCAL_BUILD_DIRECTORY = '/tmp/myapp'

REMOTE_ARTIFACT_DIRECTORY = '/tmp/artifacts

@task
def deploy():
    transfer = LocalArtifactTransfer(
        LOCAL_BUILD_DIRECTORY, REMOTE_ARTIFACT_DIRECTORY)
```

```
    with transfer as remote_destination:
        install_project_from_artifacts(remote_destination)
```

In this example, the contents of the local directory `/tmp/myapp` are copied to the remote directory `/tmp/artifacts/myapp`. The value of `remote_destination` within the context manager is `/tmp/artifacts/myapp`. After the scope of the `transfer` context manager is exited, the directory `/tmp/artifacts/myapp` on the remote machine is removed.

1.2.6 LocalArtifactInstallation

The *tunic.install.LocalArtifactInstallation* class is used to install a single local file (Go binary, Java JAR or WAR) on a remote server. Optionally, the artifact can be renamed when it is installed on the remote server.

The `LocalArtifactInstallation` class assumes that directories for a project are setup as described in *Design*.

Below is an example of using the `LocalArtifactInstallation` class to install a single Java JAR file to a release directory on a remote server.

```
from fabric.api import task
from tunic.api import LocalArtifactInstallation

APP_BASE = '/srv/www/app.example.com'

LOCAL_FILE = '/tmp/build/myapp/target/myapp-0.1.0.jar'

@task
def install():
    installation = LocalArtifactInstallation(
        APP_BASE, LOCAL_FILE, remote_name='myapp.jar')
    installation.install('20141002111442')
```

After running the `install` task above, the JAR would be installed to `/srv/www/app.example.com/releases/20141002111442/myapp.jar`.

1.2.7 HttpArtifactInstallation

The *tunic.install.HttpArtifactInstallation* class is used to install a single file (Go binary, Java JAR or WAR) on a remote server after downloading it from an HTTP or HTTPS URL. Optionally the artifact can be renamed when it is installed on the remote server.

The `HttpArtifactInstallation` class assumes that directories for a project are setup as described in *Design*.

By default downloads are performed with a `wget` call on the remote server.

Below is an example of using the `HttpArtifactInstallation` class to install a single Java JAR file to a release directory on a remote server.

```
from fabric.api import task
from tunic.api import HttpArtifactInstallation

APP_BASE = '/srv/www/app.example.com'

ARTIFACT_URL = 'https://www.example.com/builds/myapp-0.1.0.jar'

@task
```

```
def install():
    installation = HttpArtifactInstallation(
        APP_BASE, ARTIFACT_URL, remote_name='myapp.jar')
    installation.install('20141002111442')
```

After running the `install` task above, the JAR would be installed to `/srv/www/app.example.com/releases/20141002111442/myapp.jar`.

Up next is an example of using the `HttpArtifactInstallation` class with an alternate downloader. For this example we'll define a download function with the following signature (this is the interface required by *tunic.install.HttpArtifactInstallation*).

```
def download(url, destination):
    pass
```

```
from fabric.api import run, task
from tunic.api import HttpArtifactInstallation

APP_BASE = '/srv/www/app.example.com'

ARTIFACT_URL = 'https://www.example.com/builds/myapp-0.1.0.jar'

def my_downloader(url, destination):
    return run("curl --output '{path}' '{url}'".format(
        url=url, path=destination))

@task
def install():
    installation = HttpArtifactInstallation(
        APP_BASE, ARTIFACT_URL, remote_name='myapp.jar', downloader=my_downloader)
    installation.install('20141002111442')
```

After running the `install` task above, the JAR would be installed to `/srv/www/app.example.com/releases/20141002111442/myapp.jar`.

1.2.8 StaticFileInstallation

The *tunic.install.StaticFileInstallation* class is used to install static files (maybe HTML and CSS files created by a static site generator, like Nikola).

The `StaticFileInstallation` class assumes that directories for a project are setup as described in *Design*.

Below is an example of using the `StaticFileInstallation` class to install a directory of static files to a release directory on a remote server.

```
from fabric.api import task
from tunic.api import StaticFileInstallation

APP_BASE = '/srv/www/blog.example.com'

LOCAL_FILES = '/home/user/myblog/output'

@task
def install():
    installation = StaticFileInstallation(APP_BASE, LOCAL_FILES)
    installation.install('20141002111442')
```

After running the `install` task above, the contents of `~/myblog/output` would be in `/srv/www/blog.example.com/releases/20141002111442`.

1.2.9 VirtualEnvInstallation

The *`tunic.install.VirtualEnvInstallation`* class is used to install one or multiple packages into a Python virtual environment. The virtual environment is typically a particular deployment of your project.

The `VirtualEnvInstallation` class assumes that directories for a project are setup as described in *Design*.

Usage of this installer requires that the `virtualenv` tool is installed on the remote server and is on the `PATH` of the user performing the deploy or the location of the `virtualenv` tool is provided to the `VirtualEnvInstallation` class when instantiated.

Below is an example of using the `VirtualEnvInstallation` class to install a project and WSGI server from the default Python Package Index (PyPI).

```
from fabric.api import task
from tunic.api import VirtualEnvInstallation

APP_BASE = '/srv/www/myapp'

@task
def install():
    installation = VirtualEnvInstallation(APP_BASE, ['myapp', 'gunicorn'])
    installation.install('20141002111442-1.4.1')
```

The example above is simple, but not ideal. If you want a robust deploy process you probably don't want to rely on PyPI being available and you probably don't want to install whatever happens to be the latest version of a dependency. An example that installs only packages from a directory on the filesystem of the remote server is below. Presumably the packages in this directory have been created by some part of your build process or copied there by a different step in your deploy process.

```
from fabric.api import task
from tunic.api import VirtualEnvInstallation

APP_BASE = '/srv/www/myapp'

LOCAL_PACKAGES = '/tmp/build/myapp'

@task
def install():
    installation = VirtualEnvInstallation(
        APP_BASE, ['myapp', 'gunicorn'], sources=[LOCAL_PACKAGES])
    installation.install('20141002111442-1.4.1')
```

Better still, you may want to run your own local build artifact repository. In this case you'd simply include a URLs to index pages on the repository as sources. An example is below.

```
from fabric.api import task
from tunic.api import VirtualEnvInstallation

APP_BASE = '/srv/www/myapp'

MY_PACKAGES = 'https://artifacts.example.com/myapp/1.4.1/'

THIRD_PARTY = 'https://artifacts.example.com/3rd-party/1.4.1/'
```

```
@task
def install():
    installation = VirtualEnvInstallation(
        APP_BASE, ['myapp', 'gunicorn'], sources=[MY_PACKAGES, THIRD_PARTY])
    installation.install('20141002111442-1.4.1')
```

1.2.10 Putting it all together

Alright, you've seen how each individual component can be used. How does it all work together in a real deploy process? Take a look at the example below!

```
from fabic.api import hide, task, warn
from tunic.api import (
    get_current_path,
    get_releases_path,
    get_release_id,
    ProjectSetup,
    ReleaseManager,
    VirtualEnvInstallation)

APP_BASE = '/srv/www/myapp'

DEPLOY_OWNER = 'root:www'

# URLs to download artifacts from. Notice that we don't
# include version numbers in these URls. We'll use the
# version specified as part of the deploy to build source
# URLs below specific to our version.
MY_PACKAGES = 'https://artifacts.example.com/myapp/'
THIRD_PARTY = 'https://artifacts.example.com/3rd-party/'

@task
def deploy(version):
    # Ensure that the correct directory structure exists on
    # the remote server and attempt to set the permissions of
    # it to something reasonable.
    setup = ProjectSetup(APP_BASE)
    setup.setup_directories()
    setup.set_permissions(DEPLOY_OWNER)

    # Come up with a new release ID and build source URLs that
    # include the particular version of our project that we want
    # to deploy.
    release_id = get_release_id(version)
    versioned_package_sources = MY_PACKAGES + version
    versioned_third_party_sources = THIRD_PARTY + version

    # Install the 'myapp' and 'gunicorn' packages into a new
    # virtualenv on a remote server using our own custom internal
    # artifact sources, ignoring the default Python Package Index.
    installation = VirtualEnvInstallation(
        APP_BASE, ['myapp', 'gunicorn'],
        sources=[versioned_package_sources,
            versioned_third_party_sources])
```

```
    with hide('stdout'):
        # Installation output can be quite verbose, so we suppress
        # it here.
        installation.install(release_id)

    # Use the release manager to mark the just installed release as
    # the 'current' release and remove all but the N newest releases.
    release_manager = ReleaseManager(APP_BASE)
    release_manager.set_current_release(release_id)
    release_manager.cleanup()

    # Ensure that permissions and ownership of the deploys are
    # correct after the new deploy before exiting.
    setup.set_permissions(DEPLOY_OWNER)

@task
def rollback():
    release_manager = ReleaseManager(APP_BASE)
    previous = release_manager.get_previous_release()

    # If the previous version couldn't be determined for some reason,
    # we can't rollback so we just given up now. This can happen when
    # there's only a single deployment, when the 'current' symlink
    # doesn't exist, when deploys aren't named correctly, etc.
    if previous is None:
        warn("No previous release, can't rollback!")
        return

    # Atomically swap the 'current' symlink to another release.
    release_manager.set_current_release(previous)
```

1.3 API

The public API of the Tunic library is maintained in the `tunic.api` module. This is done for the purposes of clearly identifying which parts of the library are public and which parts are internal.

Functionality in the *`tunic.core`* and *`tunic.install`* modules is included in this module under a single, flat namespace. This allows a simple and consistent way to interact with the library.

1.3.1 tunic.core

Core Tunic functionality.

class tunic.core.**ProjectSetup**(*base*, *runner=None*)

Functionality for performing the initial creation of project directories and making sure their permissions are reasonable on a remote server.

Note that by default methods in this class rely on being able to execute commands with the Fabric `sudo` function. This can be disabled by passing the `use_sudo=False` flag to methods that accept it.

See *Design* for more information about the expected directory structure for deployments.

__init__(*base*, *runner=None*)

Set the base path to the project that will be setup and an optional `FabRunner` implementation to use for running commands.

Parameters

- **base** (*str*) – Absolute path to the root of the code deploy
- **runner** (*FabRunner*) – Optional runner to use for executing remote commands to set up the deploy.

Raises **ValueError** – If the base directory isn't specified

Changed in version 0.3.0: `ValueError` is now raised for empty `base` values.

set_permissions(*owner*, *file_perms='u+rw, g+rw, o+r'*, *dir_perms='u+rwx, g+rws, o+rx'*, *use_sudo=True*)

Set the owner and permissions of the code deploy.

The owner will be set recursively for the entire code deploy.

The directory permissions will be set on only the base of the code deploy and the releases directory. The file permissions will be set recursively for the entire code deploy.

If not specified default values will be used for file or directory permissions.

By default the Fabric `sudo` function will be used for changing the owner and permissions of the code deploy. Optionally, you can pass the `use_sudo=False` argument to skip trying to change the owner of the code deploy and to use the `run` function to change permissions.

This method performs between three and four network operations depending on if `use_sudo` is false or true, respectively.

Parameters

- **owner** (*str*) – User and group in the form 'owner:group' to set for the code deploy.
- **file_perms** (*str*) – Permissions to set for all files in the code deploy in the form 'u+perms,g+perms,o+perms'. Default is `u+rw,g+rw,o+r`.
- **dir_perms** (*str*) – Permissions to set for the base and releases directories in the form 'u+perms,g+perms,o+perms'. Default is `u+rwx,g+rws,o+rx`.
- **use_sudo** (*bool*) – If `True`, use `sudo()` to change ownership and permissions of the code deploy. If `False` try to change permissions using the `run()` command, do not change ownership.

Changed in version 0.2.0: `use_sudo=False` will no longer attempt to change ownership of the code deploy since this will just be a no-op or fail.

setup_directories(*use_sudo=True*)

Create the minimal required directories for deploying multiple releases of a project.

By default, creation of directories is done with the Fabric `sudo` function but can optionally use the `run` function.

This method performs one network operation.

Parameters **use_sudo** (*bool*) – If `True`, use `sudo()` to create required directories. If `False` try to create directories using the `run()` command.

class tunic.core.**ReleaseManager**(*base*, *runner=None*)

Functionality for manipulation of multiple releases of a project deployed on a remote server.

Note that functionality for managing releases relies on them being named with a timestamp based prefix that allows them to be naturally sorted – such as with the *get_release_id()* function.

See *Design* for more information about the expected directory structure for deployments.

__init__(*base*, *runner=None*)
Set the base path to the project that we will be managing releases of and an optional `FabRunner` implementation to use for running commands.

Parameters

- **base** (*str*) – Absolute path to the root of the code deploy
- **runner** (*FabRunner*) – Optional runner to use for executing remote commands to manage releases.

Raises **ValueError** – If the base directory isn't specified

Changed in version 0.3.0: `ValueError` is now raised for empty `base` values.

cleanup(*keep=5*)
Remove all but the `keep` most recent releases.

If any of the candidates for deletion are pointed to by the 'current' symlink, they will not be deleted.

This method performs N + 2 network operations where N is the number of old releases that are cleaned up.

Parameters **keep** (*int*) – Number of old releases to keep around

get_current_release()
Get the release ID of the "current" deployment, None if there is no current deployment.

This method performs one network operation.

Returns Get the current release ID

Return type str

get_previous_release()
Get the release ID of the deployment immediately before the "current" deployment, `None` if no previous release could be determined.

This method performs two network operations.

Returns The release ID of the release previous to the "current" release.

Return type str

get_releases()
Get a list of all previous deployments, newest first.

This method performs one network operation.

Returns Get an ordered list of all previous deployments

Return type list

set_current_release(*release_id*)
Change the 'current' symlink to point to the given release ID.

The 'current' symlink will be updated in a way that ensures the switch is done atomically.

This method performs two network operations.

Parameters **release_id** (*str*) – Release ID to mark as the current release

tunic.core.**get_current_path**(*base*)
Construct the path to the 'current release' symlink based on the given project base path.

Note that this function does not ensure that the 'current' symlink exists or points to a valid release, it only returns the full path that it should be at based on the given project base directory.

See *Design* for more information about the expected directory structure for deployments.

Changed in version 1.1.0: `ValueError` is now raised for empty `base` values.

Parameters **base** (*str*) – Project base directory (absolute path)

Raises **ValueError** – If the base directory isn't specified

Returns Path to the 'current' symlink

Return type str

`tunic.core.`**get_release_id**(*version=None*)

Get a unique, time-based identifier for a deployment that optionally, also includes some sort of version number or release.

If a version is supplied, the release ID will be of the form '$timestamp-$version'. For example:

```
>>> get_release_id(version='1.4.1')
'20140214231159-1.4.1'
```

If the version is not supplied the release ID will be of the form '$timestamp'. For example:

```
>>> get_release_id()
'20140214231159'
```

The timestamp component of this release ID will be generated using the current time in UTC.

Parameters **version** (*str*) – Version to include in the release ID

Returns Unique name for this particular deployment

Return type str

`tunic.core.`**get_releases_path**(*base*)

Construct the path to the directory that contains all releases based on the given project base path.

Note that this function does not ensure that the releases directory exists, it only returns the full path that it should be at based on the given project base directory.

See *Design* for more information about the expected directory structure for deployments.

Changed in version 1.1.0: `ValueError` is now raised for empty `base` values.

Parameters **base** (*str*) – Project base directory (absolute path)

Raises **ValueError** – If the base directory isn't specified

Returns Path to the releases directory

Return type str

1.3.2 tunic.install

Perform installations on remote machines.

class `tunic.install.`**HttpArtifactInstallation**(*base*, *artifact_url*, *remote_name=None*, *retries=None*, *retry_delay=None*, *downloader=None*, *runner=None*)

Download and install a single file into a remote release directory.

This is useful for installing an application that is typically bundled as a single file, e.g. Go binaries or Java JAR files, after downloading it from some sort of artifact repository (such as a company-wide file server or artifact store like Artifactory).

Downloads are performed over HTTP or HTTPS using a call to `wget` on the remote machine by default. An alternate download method may be specified when creating a new instance of this installer by providing an alternate implementation. The download method is expected to conform to the following interface.

```
>>> def download(url, destination, retries=None, retry_delay=None):
...     pass
```

Where `url` is a URL to the artifact that should be downloaded, `destination` is the absolute path on the remote machine that the artifact should be downloaded to, `retries` is the number of download attempts made after a failure, and `retry_delay` is the number of seconds between retries. The function should return the result of the Fabric command run(e.g. calling 'curl' or 'wget' with `fabric.api.run()`).

If the remote release directory does not already exist, it will be created during the install process.

See *Design* for more information about the expected directory structure for deployments.

New in version 1.2.0.

__init__(*base*, *artifact_url*, *remote_name=None*, *retries=None*, *retry_delay=None*, *downloader=None*, *runner=None*)

Set the project base directory on the remote server, URL to the artifact that should be installed remotely, and optional file name to rename the artifact to on the remote server.

Parameters

- **base** (*str*) – Absolute path to the root of the code deploy on the remote server
- **artifact_url** (*str*) – URL to the artifact to be downloaded and installed on the remote server.
- **remote_name** (*str*) – Optional file name for the artifact after it has been installed on the remote server. For example, if the artifact should always be called 'application.jar' on the remote server but might be named differently ('application-1.2.3.jar') locally, you would specify `remote_name='application.jar'` for this parameter.
- **retries** (*int*) – Max number of times to retry downloads after a failure. Default is to retry once after a failure.
- **retry_delay** (*float*) – Number of seconds between download retries. Default is not to wait between a failure and subsequent retry.
- **downloader** (*callable*) – Function to download the artifact with the interface specified above. This is primarily for unit testing but may be useful for users that need to be able to customize how the artifact HTTP store is accessed.
- **runner** (*FabRunner*) – Optional runner to use for executing remote and local commands to perform the installation.

Raises **ValueError** – If the base directory or artifact URL isn't specified.

Changed in version 1.3.0: Added the `retries` and `retry_delay` parameters

install(*release_id*)

Download and install an artifact into the remote release directory, optionally with a different name the the artifact had.

If the directory for the given release ID does not exist on the remote system, it will be created. The directory will be created according to the standard Tunic directory structure (see *Design*).

Parameters **release_id** (*str*) – Timestamp-based identifier for this deployment.

Returns The results of the download function being run. This return value should be the result of running a command with Fabric. By default this will be the result of running `wget`.

class tunic.install.**LocalArtifactInstallation**(*base*, *local_file*, *remote_name=None*, *runner=None*)

Install a single local file into a remote release directory.

This can be useful for installing applications that are typically bundled as a single file, e.g. Go binaries or Java JAR files, etc.. The artifact can optionally be renamed as part of the installation process.

If the remote release directory does not already exist, it will be created during the install process.

See *Design* for more information about the expected directory structure for deployments.

New in version 1.1.0.

__init__(*base*, *local_file*, *remote_name=None*, *runner=None*)

Set the project base directory on the remote server, local artifact (a single file) that should be installed remotely, and optional file name to rename the artifact to on the remote server.

Parameters

- **base** (*str*) – Absolute path to the root of the code deploy on the remote server
- **local_file** (*str*) – Relative or absolute path to the local artifact to be installed on the remote server.
- **remote_name** (*str*) – Optional file name for the artifact after it has been installed on the remote server. For example, if the artifact should always be called 'application.jar' on the remote server but might be named differently ('application-1.2.3.jar') locally, you would specify `remote_name='application.jar'` for this parameter.
- **runner** (*FabRunner*) – Optional runner to use for executing remote and local commands to perform the installation.

Raises **ValueError** – If the base directory or local file isn't specified.

install(*release_id*)

Install the local artifact into the remote release directory, optionally with a different name than the artifact had locally.

If the directory for the given release ID does not exist on the remote system, it will be created. The directory will be created according to the standard Tunic directory structure (see *Design*).

Parameters

- **release_id** (*str*) – Timestamp-based identifier for this deployment.
- **retries** (*int*) – Max number of times to retry downloads after a failure
- **retry_delay** (*float*) – Number of seconds between download retries

Returns The results of the `put` command using Fabric. This return value is an iterable of the paths of all files uploaded on the remote server.

class tunic.install.**LocalArtifactTransfer**(*local_path*, *remote_path*, *runner=None*)

Transfer a local artifact or directory of artifacts to a remote server when entering a context manager and clean the transferred files up on the remote server after leaving the block.

The value yielded when entering the context manager will be the full path to the transferred file or directory on the remote server. The value yielded will be made up of `remote_path` combined with the right most component of `local_path`.

For example, if `/tmp/myapp` is a local directory that contains several files, the example below will have the following effect.

```
>>> transfer = LocalArtifactTransfer('/tmp/myapp', '/tmp/artifacts')
>>> with transfer as remote_dest:
...     pass
```

The directory `myapp` and its contents would be copied to `/tmp/artifacts/myapp` on the remote machine within the scope of the context manager and the value of `remote_dest` would be `/tmp/artifacts/myapp`. After the context manager exits `/tmp/artifacts/myapp` on the remote machine will be removed.

If `/tmp/myartifact.zip` is a single local file, the example below will have the following effect.

```
>>> transfer = LocalArtifactTransfer('/tmp/myartifact.zip', '/tmp/artifacts')
>>> with transfer as remote_dest:
...     pass
```

The file `myartifact.zip` would be copied to `/tmp/artifacts/myartifact.zip` on the remote machine within the scope of the context manager and the value of `remote_dest` would be `/tmp/artifacts/myartifact.zip`. After the context manager exits `/tmp/artifacts/myartifact.zip` on the remote machine will be removed.

The destination of the artifacts must be a directory that is writable by the user running the deploy or that the user has permission to create.

The path yielded by the context will be removed when the context manager exits. The local artifacts are not modified or removed on exit.

New in version 0.4.0.

__enter__()
Transfer the local artifacts to the appropriate place on the remote server (ensuring the path exists first) and return the remote destination path.

The remote destination path is the remote path joined with the right-most component of the local path.

Returns The path artifacts were transferred to on the remote server

Return type str

__exit__(*exc_type*, *exc_val*, *exc_tb*)
Remove the directory containing the build artifacts on the remote server.

__init__(*local_path*, *remote_path*, *runner=None*)
Set the local directory that contains the artifacts and the remote directory that they should be transferred to.

Both the local and remote paths should not contain trailing slashes. Any trailing slashes will be removed.

Parameters

- **local_path** (*str*) – Directory path on the local machine that contains the build artifacts to be transferred (without a trailing slash) or the path on the local machine of a single file.
- **remote_path** (*str*) – Directory on the remote machine that the build artifacts should be transferred to (without a tailing slash).
- **runner** (*FabRunner*) – Optional runner to use for executing commands to transfer artifacts.

Changed in version 0.5.0: Trailing slashes are now removed from `local_path` and `remote_path`.

class tunic.install.**StaticFileInstallation**(*base*, *local_path*, *runner=None*)
Install the contents of a local directory into a remote release directory.

If the remote release directory does not already exist, it will be created during the install process.

See *Design* for more information about the expected directory structure for deployments.

New in version 0.5.0.

__init__(*base*, *local_path*, *runner=None*)
Set the project base directory on the remote server and path to a directory of static content to be installed into a remote release directory.

Parameters

- **base** (*str*) – Absolute path to the root of the code deploy on the remote server
- **local_path** (*str*) – Absolute or relative path to a local directory whose contents will be copied to a remote release directory.
- **runner** (*FabRunner*) – Optional runner to use for executing remote and local commands to perform the installation.

Raises **ValueError** – If the base directory or local path isn't specified.

install(*release_id*)
Install the contents of the local directory into a release directory.

If the directory for the given release ID does not exist on the remote system, it will be created. The directory will be created according to the standard Tunic directory structure (see *Design*).

Note that the name and path of the local directory is irrelevant, only the contents of the specified directory will be transferred to the remote server. The contents will end up as children of the release directory on the remote server.

Parameters **release_id** (*str*) – Timestamp-based identifier for this deployment. If this ID corresponds to a directory that already exists, contents of the local directory will be copied into this directory.

Returns The results of the `put` command using Fabric. This return value is an iterable of the paths of all files uploaded on the remote server.

class tunic.install.**VirtualEnvInstallation**(*base*, *packages*, *sources=None*, *venv_path=None*, *runner=None*)
Install one or multiple packages into a remote Python virtual environment.

If the remote virtual environment does not already exist, it will be created during the install process.

The installation can use the standard package index (PyPI) to download dependencies, or it can use one or multiple alternative installation sources (such as a local PyPI instance, Artifactory, the local file system, etc.) and ignore the default index. These two modes are mutually exclusive.

See *Design* for more information about the expected directory structure for deployments.

New in version 0.3.0.

__init__(*base*, *packages*, *sources=None*, *venv_path=None*, *runner=None*)
Set the project base directory, packages to install, and optionally alternative sources from which to download dependencies and path to the virtualenv tool.

Parameters

- **base** (*str*) – Absolute path to the root of the code deploy on the remote server
- **packages** (*list*) – A collection of package names to install into a remote virtual environment.

- **sources** (*list*) – A collection of alternative sources from which to install dependencies. These sources should be strings that are either URLs or file paths. E.g. 'http://pypi.example.com/simple/' or '/tmp/build/mypackages'. Paths and URLs may be mixed in the same list of sources.
- **venv_path** (*str*) – Optional absolute path to the virtualenv tool on the remote server. Required if the virtualenv tool is not in the PATH on the remote server.
- **runner** (*FabRunner*) – Optional runner to use for executing remote and local commands to perform the installation.

Raises **ValueError** – If the base directory isn't specified, if no packages are given, packages is not an iterable collection of some kind, or if sources is specified but not an iterable collection of some kind.

Changed in version 0.4.0: Allow the path to the `virtualenv` script on the remote server to be specified.

install(*release_id*, *upgrade=False*)

Install target packages into a virtual environment.

If the virtual environment for the given release ID does not exist on the remote system, it will be created. The virtual environment will be created according to the standard Tunic directory structure (see *Design*).

If `upgrade=True` is passed, packages will be updated to the most recent version if they are already installed in the virtual environment.

Parameters

- **release_id** (*str*) – Timestamp-based identifier for this deployment. If this ID corresponds to a virtual environment that already exists, packages will be installed into this environment.
- **upgrade** (*bool*) – Should packages be updated if they are already installed in the virtual environment.

Returns The results of running the installation command using Fabric. Note that this return value is a decorated version of a string that contains additional meta data about the result of the command, in addition to the output generated.

Return type str

1.4 Change Log

1.4.1 1.3.0 - 2017-08-31

- Add retry settings to *tunic.install.HttpArtifactInstallation* to allow more robust deploys over unreliable networks. Fixes #8.

1.4.2 1.2.3 - 2017-06-20

- **Bug** - Fix bug where local permissions were not mirrored on the remote side when using *LocalArtifactInstallation*.

1.4.3 1.2.2 - 2016-05-06

- **Bug** - Fix bug where newer version of `cryptography` module being pulled in than supported by the version of Fabric we depended on. Fixed by updating to Fabric 1.11.1. Fixes #5.

1.4.4 1.2.1 - 2016-02-25

- **Bug** - Fix bug when running with older versions of Fabric that didn't define a `warn_only` context manager. Fixes #4.

1.4.5 1.2.0 - 2016-02-25

- Add *tunic.install.HttpArtifactInstallation* for installation of a single artifact downloaded from an HTTP or HTTPS URL. Useful when installation artifacts are stored in some central file store or repository (like Artifactory).
- Minor documentation fixes.

1.4.6 1.1.0 - 2015-06-03

- **Bug** - Fix bug in *tunic.core.get_current_path()* and *tunic.core.get_releases_path()* where input was not being checked to ensure it was a valid base directory. `ValueError` will now be raised for invalid (blank or `None`) values.
- Added fuzz testing to some parts of the Tunic unit test suite.
- Added *tunic.install.LocalArtifactInstallation* for installation of single local artifact onto a remote server. Useful for Go binaries or Java JARs and WARs.

1.4.7 1.0.1 - 2015-04-04

- This is the first stable release of Tunic. From this point on, all breaking changes will only be made in major version releases.

  This release is almost functionally equivalent to the `0.5.0` release. There are only minor changes to the build process and project documentation.
- Packaging fixes (use `twine` for uploads to PyPI, stop using the setup.py `register` command).
- Assorted documentation updates.

1.4.8 0.5.0 - 2014-10-11

- **Breaking change** - Change behavior of *tunic.install.LocalArtifactTransfer* to yield the final destination path on the remote server (a combination of the remote path and right-most component of the local path). This new value will be the only path removed on the remote server when the context manager exits.
- **Breaking change** - Trailing slashes on `local_path` and `remote_path` constructor arguments to *tunic.install.LocalArtifactTransfer* are now removed before being used.
- Add *tunic.install.StaticFileInstallation* class for installation of static files into a release on a remote server.

1.4.9 0.4.0 - 2014-10-02

- Allow override of the `virtualenv` script location on the remote server when using the *tunic.install.VirtualEnvInstallation* class.
- Add *Usage* section to the documentation that explains how to use each part of the library at a higher level than just the *API* section.
- Add *tunic.install.LocalArtifactTransfer* class for transferring locally built artifacts to a remote server and cleaning them up after deployment has completed.

1.4.10 0.3.0 - 2014-09-28

- Test coverage improvements
- *tunic.core.ReleaseManager* and *tunic.core.ProjectSetup* now throw `ValueError` for invalid `base` values in their `__init__` methods.
- Fix bug where we attempted to split command output by `\n\r` instead of `\r\n`.
- Add *tunic.install.VirtualEnvInstallation* class for performing remote virtualenv installations.

1.4.11 0.2.0 - 2014-09-26

- Add initial documentation for Tunic API
- Add design decision documentation for library
- Change behavior of *tunic.core.ProjectSetup.set_permissions()* to not attempt to change the ownership of the code deploy unless it is using the `sudo` function

1.4.12 0.1.0 - 2014-09-22

- Initial release

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

Symbols

C

G

H

I

L

P

R

S

T

V