zanna Documentation

Release latest

May 21, 2021

Contents

Contents:

CHAPTER 1

zanna

Simple Dependency Injection library. Supports python 3.5+ and makes full use of the typing annotations. The design is pythonic but inspired by Guice in many aspects.

- Free software: BSD license
- Documentation: https://zanna.readthedocs.io.

1.1 Motivation

Zanna is meant to be a modern (3.5+), well maintained injection library for Python.

1.2 Features

- Support for typing annotations
- Decorators are not mandatory: all the injection logic can be outside your modules
- Supports injection by name
- Instances can be bound directly, useful when testing (i.e. by override bindings with mocks)
- No autodiscover for performance reasons and to avoid running into annoying bugs

1.3 Usage

1.3.1 Injecting by variable name

The basic form of injection is performed by variable name. The injector expects a list of modules (any callable that takes a Binder as argument). You can get the bound instance by calling get_instance

```
from zanna import Injector, Binder

def mymodule(binder: Binder) -> None:
    binder.bind_to("value", 3)

injector = Injector(mymodule)
assert injector.get_instance("value") == 3
```

Zanna will automatically inject the value into arguments with the same name:

```
from zanna import Injector, Binder

def mymodule(binder: Binder) -> None:
    binder.bind_to("value", 3)

class ValueConsumer:
    def __init__(self, value):
        self.value = value

injector = Injector(mymodule)
assert injector.get_instance(ValueConsumer).value == 3
```

1.3.2 Injecting by type annotation

Zanna also makes use of python typing annotations to find the right instance to inject.

```
from zanna import Injector, Binder

class ValueClass:
    def __init__(self, the_value: int):
        self.the_value = the_value

class ValueConsumer:
    def __init__(self, value_class_instance: ValueClass):
        self.value_class_instance = value_class_instance

def mymodule(binder: Binder) -> None:
    binder.bind_to("the_value", 3)
    binder.bind(ValueClass)

injector = Injector(mymodule)
assert injector.get_instance(ValueConsumer).value_class_instance.the_value == 3
```

1.3.3 Singleton or not singleton?

Instances provided by the injector are always singletons, meaning that the __init__ method of the class will be called only the first time, and every subsequent call of get_instance will return the same instance:

```
from zanna import Injector

class MyClass:
    pass
injector = Injector(lambda binder: binder.bind(MyClass))
assert injector.get_instance(MyClass) is injector.get_instance(MyClass)
```

1.3.4 Use providers for more complex use cases

Binder instances can be used to bind providers. A provider is any callable that takes any number of arguments and returns any type. The injector will try to inject all the necessary arguments. Providers can be bound explicitely or implicitely (in which case zanna will use the return annotation to bind by type).

```
from zanna import Injector, Binder

class AValueConsumer:
    def __init__(self, value: int):
        self.value = value

def explicit_provider(a_value: int) -> int:
    return a_value + 100

def implicit_provider(value_plus_100: int) -> AValueConsumer:
    return AValueConsumer(value_plus_100)

def mymodule(binder: Binder) -> None:
    binder.bind_to("a_value", 3)
    binder.bind_provider("value_plus_100", explicit_provider)
    binder.bind_provider(implicit_provider)

injector = Injector(mymodule)
assert injector.get_instance(AValueConsumer).value == 103
```

1.3.5 Override existing bindings

Bindings can be overridden. Overriding a non-existent binding will result in a ValueError being raised.

Override bindings is extremely useful when testing, as any part of your stack can be replaced with a mock.

```
from zanna import Injector, Binder
from unittest.mock import MagicMock

class ValueClass:
    def __init__(self):
        pass
    def retrieve_something(self):
        return ['some', 'thing']

class ValueConsumer:
    def __init__(self, value: ValueClass):
        self.value = value


def mymodule(binder: Binder) -> None:
```

(continues on next page)

(continued from previous page)

```
    binder.bind(ValueClass)

injector = Injector(mymodule)
assert injector.get_instance(ValueConsumer).value.retrieve_something() == ['some',
↪'thing']

def module_overriding_value_class(binder: Binder) -> None:
    mock_value_class = MagicMock(ValueClass)
    mock_value_class.retrieve_something.return_value = ['mock']
    binder.override_binding(ValueClass, mock_value_class)

injector = Injector(mymodule, module_overriding_value_class)
assert injector.get_instance(ValueConsumer).value.retrieve_something() == ['mock']
```

1.3.6 Using the decorators

One of the advantages of using Zanna over other solutions is that it doesn't force you to pollute your code by mixing in the injection logic.

If you are working on a small project and would like to handle part (or all) of the injection logic using decorators instead of modules, Zanna supports that as well.

Internally, Zanna creates a module that sets up the bindings as indicated by the decorators (in a random order).

All Injectors initialized with use_decorators=True will run that module first on their Binder.

Zanna supports the following decorators:

- decorators.provider, which takes a provided annotated with an appropriate return type
- decorators.provider_for, which can be given the name or the class of the instance provided
- decorators.inject, to annotate class to be bound/injected

Here's an example:

```
from zanna import Injector
from zanna import decorators
class Thing:
    pass

@decorators.provider_for("value")
def provide_value():
    return 3

@decorators.provider
def provide_thing() -> Thing:
    return Thing()


@decorators.inject
class OtherThing:
    def __init__(self, value, thing:Thing):
        self.value = value
        self.thing = thing

inj = Injector(use_decorators=True)
otherthing = inj.get_instance(OtherThing)
```

(continues on next page)

(continued from previous page)

```
assert otherthing.value == 3
assert isinstance(otherthing.thing, Thing)
assert isinstance(otherthing, OtherThing)
```

1.3.7 Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER 2

Installation

2.1 Stable release

To install zanna, run this command in your terminal:

```
$ pip install zanna
```

This is the preferred method to install zanna, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for zanna can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/MirkoRossini/zanna
```

Or download the tarball:

```
$ curl  -OL https://github.com/MirkoRossini/zanna/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 3

Usage

To use zanna in a project:

```
import zanna
```

CHAPTER 4

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

4.1 Types of Contributions

4.1.1 Report Bugs

Report bugs at https://github.com/MirkoRossini/zanna/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

4.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

4.1.4 Write Documentation

zanna could always use more documentation, whether as part of the official zanna docs, in docstrings, or even on the web in blog posts, articles, and such.

4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/MirkoRossini/zanna/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

4.2 Get Started!

Ready to contribute? Here's how to set up *zanna* for local development.

1. Fork the *zanna* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/zanna.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv zanna
$ cd zanna/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 zanna tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, 3.4 and 3.5, and for PyPy. Check https://travis-ci.org/MirkoRossini/zanna/pull_requests and make sure that the tests pass for all supported Python versions.

4.4 Tips

To run a subset of tests:

```
$ py.test tests.test_zanna
```

CHAPTER 5

Indices and tables

- genindex
- modindex
- search