giveme Documentation

Release 1.0.0

Steinthor Palsson

Nov 27, 2017

Contents:

Giveme is a dependency injection framework for python. It gives you the tools to separate external services, such as database wrappers and web frameworks from your business logic.

CHAPTER 1

Quickstart

The *Injector* class is at the heart of giveme. It acts as a dependency registry and injects dependencies into function arguments.

You register a dependency factory using its *register()* decorator and inject a dependency into a another function or method using the *inject()* decorator.

```
from giveme import Injector

injector = Injector()

@injector.register
def magic_number():
    return 42

@injector.inject
def double_number(magic_number):
    return magic_number*2

double_number()
```

```
84
```

CHAPTER 2

Dependency cache (singleton and thread local dependencies)

By default, the injector calls the dependency factory every time its used. So in the example above, `magic_number()` is called every time you call `double_number` without arguments

In the real world, your dependencies are generally more complex objects that may involve network calls that are expensive to initalize or carry some kind of state that you want to persist between uses.

Using *`register()`* with the `singleton` argument achieves this by only calling the dependency factory the first time it's used, after that its return value is cached for subsequent uses. E.g.

```
@injector.register(singleton=True)
def number_list():
    return [1, 2, 3]

@injector.inject
def increment_list(number_list):
    for i in range(len(number_list)):
        number_list[i] += 1
    return number_list

print(increment_list())
print(increment_list())
```

```
[2, 3, 4]
[3, 4, 5]
```

Every call to `increment_list()` operates on the same instance of `number_list()`

`threadlocal` can also be used for the same effect, that makes the dependency cache use `Threading.local` storage behind the scenes so that each instance of a dependency is only available to the thread that created it.

CHAPTER 3

Naming dependencies

By default *register()* uses the name of the decorated function as the dependency name.

This can be overriden using the `name` keyword argument:

```
@injector.register(name='cache_wrapper')
def redis_cache():
    ...
```

When injecting *inject()* matches dependency names to the decorated function's arguments. This can also be overriden by passing any number of keyword arguments in the format of `argument_name='dependency_name'`

Example:

```
@injector.inject(cache='cache_wrapper')
def do_cache_stuff(cache):
    # cache receives the 'cache_wrapper' dependency
    ...
```

CHAPTER 4

Nested dependencies

A dependency may have its own dependencies. For instance you might have two database wrappers that share a database connection (pool). Luckily you can inject dependencies into other dependencies same as anything else, e.g.:

```
import redis


@injector.register(singleton=True)
def redis_client():
    return redis.Redis.from_url('my_redis_url')


@injector.register(singleton=True)
@injector.inject
def cache(redis_client):
    return MyRedisCache(redis_client)


@injector.register(singleton=True)
@injector.inject
def session_store(redis_client):
    return MyRedisSessionStore(redis_client)
```

You can now inject `cache` or `session_store` into other functions and both will use the same `Redis` instance behind the scenes.

CHAPTER 5

Argument binding

inject() handles any combination of injected and manually passed arguments and it only injects for arguments that are not explicitly passed in. Ordering does not matter beyond python's regular argument order rules.

E.g. This works as expected:

```
@injector.register
def something():
    return 'This is a dependency'


@injector.inject
def do_something(a, *args, something, b=100, c=200, **kwargs):
    return a, args, something, b, c, kwargs

do_something(1, 2, 3, 4, 5, b=200, c=300, x=55)
```

And to override the dependency

```
do_something(1, 2, 3, 4, 5, something='overriden dependency', b=200, c=300, x=55)
```

CHAPTER 6

Bypass injection

Dependency injection can always be bypassed by manually passing in replacement values for their respective arguments.

For instance in our `increment_list` function above:

```
print(increment_list())
print(increment_list([0, 0, 0])
```

```
[2, 3, 4]
[1, 1, 1]
```

CHAPTER 7

API reference

CHAPTER 8

giveme.injector

class giveme.injector.**Dependency**(*name*, *factory*, *singleton=False*, *threadlocal=False*)
Bases: object

__init__(*name*, *factory*, *singleton=False*, *threadlocal=False*)

factory

name

singleton

threadlocal

exception giveme.injector.**DependencyNotFoundError**
Bases: Exception

exception giveme.injector.**DependencyNotFoundWarning**
Bases: RuntimeWarning

class giveme.injector.**Injector**
Bases: object

__init__()

cache(*dependency: giveme.injector.Dependency*, *value*)
Store an instance of dependency in the cache. Does nothing if dependency is NOT a threadlocal or a singleton.

Parameters

- **dependency** (Dependency) – The Dependency to cache
- **value** – The value to cache for dependency

cached(*dependency*)
Get a cached instance of dependency.

Parameters **dependency** (Dependency) – The Dependency to retrievie value for

Returns The cached value

delete (*name*)
Delete (unregister) a dependency by name.

get (*name: str*)
Get an instance of dependency, this can be either a cached instance or a new one (in which case the factory is called)

inject (*function=None*, ***names*)
Inject dependencies into *funtion*'s arguments when called.

```
>>> @injector.inject
... def use_dependency(dependency_name):
        ...
>>> use_dependency()
```

The *Injector* will look for registered dependencies matching named arguments and automatically pass them to the given function when it's called.

Parameters

- **function** (*callable*) – The function to inject into
- ****names*** – in the form of `argument='name'` to override the default behavior which matches dependency names with argument names.

register (*function=None*, *, *singleton=False*, *threadlocal=False*, *name=None*)
Add an object to the injector's registry.

Can be used as a decorator like so:

```
>>> @injector.register
... def my_dependency(): ...
```

or a plain function call by passing in a callable injector.register(my_dependency)

Parameters

- **function** (*callable*) – The function or callable to add to the registry
- **name** (*string*) – Set the name of the dependency. Defaults to the name of *function*
- **singleton** (*bool*) – When True, register dependency as a singleton, this means that *function* is called on first use and its return value cached for subsequent uses. Defaults to False
- **threadlocal** (*bool*) – When True, register dependency as a threadlocal singleton, Same functionality as `singleton` except `Threading.local` is used to cache return values.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index

Symbols

C

D

F

G

I

N

R

S

T