Smart Injector

Release 0.0.6

May 27, 2020

Contents

CHAPTER 1

Overview

Smart-injector is an lightweight dependency injection framework for Python. It was designed to be an easy to use tool to implement dependency injection in your application in a pythonic way.

Key features are: - Ligtweight - pure python - leverages type annotations - scopes - non invasive

1.1 Project Status

docs	
tests	
package	

1.2 Installation

```
pip install smart-injector
```

You can also install the in-development version with:

```
pip install https://github.com/hlevering/python-smart-injector/archive/master.zip
```

1.3 Quickstart

Basic Usage:

```
>>> class A:
...     pass
...
>>> class B:
...     def __init__(self, a: A):
...         self.a = a
...
>>> container = create_container(StaticContainer)
>>> b = container.get(B)
>>> isinstance(b.a, A)
True
```

1.4 Documentation

Detailed documentation can be found here:

https://python-smart-injector.readthedocs.io/

Further usage examples can be found here: https://python-smart-injector.readthedocs.io/en/latest/usage.html

1.5 Development

To run the all tests run:

```
tox
```

Note, to combine the coverage data from all the tox environments run:

Windows	`set PYTEST_ADDOPTS=--cov-append` `tox`
Other	`PYTEST_ADDOPTS=--cov-append tox`

CHAPTER 2

Installation

At the command line:

```
pip install smart-injector
```

CHAPTER 3

Usage

Smart injector provides a bunch of easy-to-use functions and methods, which let you configure your container as you need it quickly.

3.1 Basic example:

```
from smart_injector import create_container

class A:
    pass

class B:
    def __init__(self, a: A):
        self.a = a

container = create_container()
b = container.get(B)
print(isinstance(b.a, A))
```

If you have only dependencies on concrete types, no further configuration will be needed and you can use the Di container as it is.

```
True
```

Smart-injector relies on type annotations to resolve dependencies. Therefore type annotated code is a must have, if you want to use smart-injector efficiently. For many cases smart-injector can resolve dependencies automatically. However, there are some limitations for the automatic depenendency resolving mechanism.

- dependencies on abstract types
- dependencies on builtin types
- explicitly provide arguments
- setting lifetime of a dependency

- set dependency to a specific instance

These cases need explicitly configuration. Basically, container configuration follows this pattern:

First you define a function which takes one parameter of type *smart_injector.Config*. Then you provide this function as a parameter to the factory function *smart_injector.create_container()*. In your function you can use the methods provided by the *Config* object to configure your container. In the next sections we will cover how this is done in detail.

3.2 Dependencies on abstract types

Abstract types (classes that inherit from `abc.ABC()` and have at least one `abc.abstractmethod()`) cannot be instantiated directly. There for it is impossible for smart-injector to resolve these kind of dependencies. An explicit binding of an abstract class to a concrete class must be configured. This is done by using *smart_injector.Config.bind()*.

```
from abc import ABC, abstractmethod

class A(ABC):
    @abstractmethod
    def do(self):
        pass

class ConcretA(A):
    def do(self):
        print("Hello")

from smart_injector import Config # not needed but lets you type annotate your␣
↪configure method

# create your own configuration function. This function must take a parameter of type␣
↪Config
def configure(config: Config):
    # use config's bind method to bind A to ConcretA
    config.bind(A, ConcretA)
    # now if there is a dependency on A ,then an instance of ConcretA will be injected

# create an instance of your new defined container
container = create_container(configure)
a = container.get(A)
a.do()
```

With the above configuration, the container will inject an instance of type *ConcreteA*, whenever there is a dependency on *A*.

```
Hello
```

Binding is not restricted to abstract classes. You can bind type A to type B as long as type B is a subclass of type A. Moreover, it is possible to chain bindings. Let's take the last example and add one more class.

```
from abc import ABC, abstractmethod

class A(ABC):
    @abstractmethod
    def do(self):
        pass
```

(continues on next page)

(continued from previous page)

```
class ConcretA(A):
    def do(self):
        print("Hello")

class ConcretB(ConcretA):
    def do(self):
        print("World")

def configure(config: Config):
    config.bind(A, ConcretA)
    config.bind(ConcretA, ConcretB)
    # now everytime when there is a dependency on A then ConcretB will be injected

# create an instance of your new defined container
container = create_container(configure)
a = container.get(A)
a.do()
```

Instead of *A* an instance of *ConcreteA* should be used, but since there is a binding from *ConcreteA* to *ConcreteB* effectively there will be inject an instance of *ConcreteB*.

```
World
```

Additionally, you can bind types to functions. For this to work, the function must return either an instance of that type or an instance of a subclass of that type.

```
from abc import ABC, abstractmethod

class A(ABC):
    @abstractmethod
    def do(self):
        pass

class ConcretA(A):
    def do(self):
        print("Hello")


class ADependency:
    pass

def concret_a_factory(dependency: ADependency)->ConcretA:
    return ConcretA()


def configure(config: Config):
    config.bind(A, concret_a_factory)
    # now everytime when there is a dependency on A then the object returned by␣
↪concret_a_factory will be injected

# create an instance of your new defined container
container = create_container(configure)
a = container.get(A)
a.do()
```

In the above example “concrete_a_factory” was called to get an instance of A. In addition, the dependencies of con-

rete_a_factory are injected automatically.

```
Hello
```

3.3 Dependencies on builtin types

Dependencies on builtin types are default constructed by default.

```
container = create_container()
print(container.get(int))
print(container.get(float))
print(container.get(str))
print(container.get(bytes))
print(container.get(bytearray))
```

```
0
0.0

b''
bytearray(b'')
```

This is rather useful. Therefore, a method for providing values for constructor or function parameters would be useful.

3.4 Explicitly provide arguments

You can provide arguments explicitly by configuring your container to do so. Either by specifying values for the arguments or by specifying a factory function for an argument, which will be called when resolving dependencies.

3.4.1 Values for arguments

Values for arguments can be set with *smart_injector.Config.arguments()*.

```
class MyClass:
    def __init__(self, a: str, b: int, c: float):
        self.a = a
        self.b = b
        self.c = c

def configure(config: Config):
    # use config's arguments method to provide some arguments
    config.arguments(MyClass, a="hello", b=42, c=1.0)
    # now everytime when there is a dependency on MyClass then MyClass(a="hello",␣
↪b=42, c=1.0) will be inserted

container = create_container(configure)
a = container.get(MyClass)
print(a.a)
print(a.b)
print(a.c)
```

In the above example *MyClass* will be created as *MyClass(a="hello", b=42, c=1.0)*.

```
hello
42
1.0
```

If arguments are provided explicitly, it is not necessary to provide all arguments. Arguments which are not specified, are resolved automatically by the DI container .

```
class Foo:
    pass

class MyClass:
    def __init__(self, a: str, foo: Foo, c: float):
        self.a = a
        self.foo = foo
        self.c = c

def configure(config: Config):
    # use config's arguments method to provide some arguments
    config.arguments(MyClass, a="hello", c=1.0)

container = create_container(configure)
a = container.get(MyClass)
print(a.a)
print(isinstance(a.foo, Foo))
print(a.c)
```

In the above example no argument for parameter *foo* was specified. Therefore, the dependency on *foo* is resolved by the container. In this case it is a default constructed *Foo()*.

```
hello
True
1.0
```

By explicitly providing arguments it is also possible to resolve dependencies without type annotations.

```
class MyClass:
    def __init__(self, a):
        self.a = a

def configure(config: Config):
    # use config's arguments method to provide some arguments
    config.arguments(MyClass, a="hello")
    # now everytime when there is a dependency on MyClass then MyClass(a=
↪"hello", b=42, c=1.0) will be inserted

container = create_container(configure)
a = container.get(MyClass)
print(a.a)
```

There is no type annotation for *MyClass* parameter *a*. Anyhow, the value "hello" is injected correctly for parameter *a*.

```
hello
```

Note: At the moment only keyword arguments can be provided with arguments. Moreover, you cannot provide the keyword argument "where" which is used to specify arguments in a specific context (see Context section for further information).

3.4.2 Setting factories for arguments

Instead of providing values for parameters, it is also possible to define a function which will be called to retrieve the value for the parameter. A factory for a parameter is set with *smart_injector.Config.arg_factory()*.

```
class MyClass:
    def __init__(self, a: str):
        self.a = a

def get_a()->str:
    return "hello"

def configure(config: Config):
    config.arg_factory(MyClass, a=get_a)

container = create_container(configure)
a = container.get(MyClass)
print(a.a)
```

Result:

```
hello
```

You can provide any callable as a factory. If necessessary, dependencies of the factory function are injected automatically by smart_injector. Additionally, if you provide a method of a factory function, smart_injector will create a class instance and then call that method. (smart_injector will also create and inject all dependencies to create that instance automatically)

```
class MyInt:
    def get_int(self) -> int:
        return 42


class ProvidesInt:
    def __init__(self, a_int: MyInt):
        self._a_int = a_int

    def get_int(self) -> int:
        return self._a_int.get_int()


class NeedsInt:
    def __init__(self, a_int: int):
        self.a_int = a_int


def configure(config: Config):
    config.arg_factory(NeedsInt, a_int=ProvidesInt.get_int)

container = create_container(configure)
needs_int = container.get(NeedsInt)
print(needs_int.a_int)
```

Smart_injector creates an instance of ProvidesInt automatically (and it will inject an instance of MyInt into it). Then it calls method "get_int" of that previously created instance.

```
42
```

For this kind of factory methods it is impossible to set arguments for the method explicitly with *smart_injector.Config.arguments()*.

3.5 Setting dependency's lifetime

By default all injected objects have a transient lifetime. That means, that every time when an object is needed a new instance of that object is created.

```
class A:
    pass

class B:
    pass

from smart_injector import Lifetime

def configure(config: Config):
    # use config's lifetime method to specify an objects lifetime
    config.lifetime(A, lifetime=Lifetime.SINGLETON)
    # now there will be only one object of type A, which will be inserted wherever an␣
↪object A is needed
    config.lifetime(B, lifetime=Lifetime.TRANSIENT)
    # everytime a new object B is created. This is the default behaviour for all types

container = create_container(configure)
a1 = container.get(A)
a2 = container.get(A)
b1 = container.get(B)
b2 = container.get(B)
print(a1 is a2)
print(b1 is b2)
```

b1 and *b2* refer to the same object since lifetime of *B* was defined as *SINGLETON*.

```
True
False
```

It is possible to override the default lifetime for objects created by a container. This must be done when the container is created.

```
class A:
    pass


from smart_injector import Lifetime

container = create_container(default_lifetime=Lifetime.SINGLETON)
a1 = container.get(A)
a2 = container.get(A)
print(a1 is a2)
```

```
True
```

3.6 Specify a specific instance

If you want a specific instance to be used for a type, you can do that, too. You have specify the instance with *smart_injector.Config.instance()*.

```
class A:
    def __init__(self, a: str):
        self.a = a

my_a = A("foo")

def configure(config: Config):
    # use config's instance method to specify that a particular instance shall be used
    config.instance(A, my_a)
    # every time an object of type A is needed, the instance my_a will be returned

container = create_container(configure)
a1 = container.get(A)
print(a1 is my_a)
```

```
True
```

TODO explanation for contexts and *where* parameter

3.7 Get a configured object from the container

When you ask the container to provide you an object of type *T* by calling *smart_injector.StaticContainer.get()* with *T*, the container will provide and configure the object in a specific way.

3.7.1 Resolving Order

First of all, the container determines, which real type is requested and if a new instance has to be created:

1. If an instance of T was set with *smart_injector.Config.instance()* method, use this instance of T.
2. If a binding was specified for T with *smart_injector.Config.bind()*, use the bounded type instead of T and start again with a new request with the bounded type.
3. If T's lifetime is singleton(with *smart_injector.Config.lifetime()* or *smart_injector.create_container()* and default_lifetime = `smart_injector.Lifetime.SINGLETON`, create a new instance of T at the first request. Return the same instance on every subsequent request.
4. If T is a builtin type, than use the type's default constructor.
5. Create a new instance of T.

3.7.2 New Instance Creation

When a new instance of T must be created. The container will resolve all dependencies of T via the following schema:

1. Determine all dependencies of T. This means all argument of T's constructor, if is a class or of T itself if it is a function.
2. Remove all dependencies of T, which were already set with *smart_injector.Config.arguments()* or *smart_injector.Config.arg_factory()*.

3. Resolve the remaining dependencies by asking the container to resolve each dependency.
4. For every argument, for which there was set a factory with *Config.arg_factory*, call that factory function. This is done by asking the container to resolve the factory function by calling `smart_injector.StaticContainer.get()`. Therefore, all dependencies of that factory function are resolved automatically, too.
5. Create the new oject of type T with the former resolved dependencies injected.

CHAPTER 4

Reference

4.1 smart_injector

smart_injector.**create_container**(*configure: Optional[Callable[[smart_injector.config.user.Config], None]] = None, default_lifetime=<Lifetime.TRANSIENT: 1>, dependencies: Optional[List[object]] = None*) → smart_injector.container.container.StaticContainer

Use this function to create a DI container.

Parameters

- **configure** –
- **default_lifetime** –
- **dependencies** –

Returns

class smart_injector.**StaticContainer**(*resolver: smart_injector.resolver.resolver.Resolver*)

DI Container. Used by the user to get instances of types.

To get your own container. Create a new class inherited from this class and override configure method

get(*a_type: Callable[[...], T]*) → T

Get an instance of type *T*

Parameters **a_type** – either a class *T* or a function returning a *T*

Returns an instance of *T*

class smart_injector.**Lifetime**

Specifies the lifetime for objects created by the container

Lifetime.SINGLETON *smart_injector.StaticContainer.get()* returns the same every instance on every call

Lifetime.TRANSIENT *smart_injector.StaticContainer.get()* returns a new instance on every call

class smart_injector.**Config**(*backend: smart_injector.config.backend.ConfigBackend*)
Used by the user to configure DI container injection behaviour

arg_factory(*a_type: Callable[[...], T], where: Optional[Type[T]] = None, **kwargs*)
In difference to *arguments()*: Instead of providing a value for *parameter* directly, *factory* is called to get the value for the parameter.

Parameters

- **a_type** –
- **where** –
- **kwargs** –

Returns

arguments(*a_type: Callable[[...], T], where: Optional[Type[T]] = None, **kwargs*)
When creating an object of type *T*, the provided arguments will be inserted in *T*'s constructor (if it is a class) or a_type will be called with this arguments if it is a function.

Parameters

- **a_type** –
- **where** –
- **kwargs** –

Returns

Note: Only keyword arguments are supported

---

bind(*a_type: Callable[[...], T], to_type: Callable[[...], S], where: Optional[Type[T]] = None*)
Specify a binding. Whenever an object of type a_type is required, then an object of type to_type will be provided. For example you can configure, which concrete class shall be used for an abstract base class

Parameters

- **a_type** – will be replaced by to_type
- **to_type** – will be used when an object of type a_type is required. to_type must be a subclass of a_type
- **where** –
- **kwargs** –

Returns

dependency(*a_type: Callable[[...], T]*)
declare that *T* is a dependency for the container. When creating the container with *smart_injector.create_container()* an instance must be provided for every dependency which was declared.

Parameters **a_type** –

Returns

instance(*a_type: Callable[[...], T], instance: T, where: Optional[Type[T]] = None*)
set an instance of type *T* which is returned whenever an object of type *T* is requested

Parameters

- **a_type** –

- **instance** –
- **where** –

Returns

lifetime(*a_type: Callable[[...], T], lifetime: smart_injector.lifetime.Lifetime, where: Optional[Type[T]] = None*)

Specify the lifetime for an object of type *T*. See *smart_injector.Lifetime()*

Parameters

- **a_type** –
- **lifetime** –
- **where** –

Returns None

CHAPTER 5

Contributing

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

5.1 Bug reports

When 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.

5.2 Documentation improvements

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

5.3 Feature requests and feedback

The best way to send feedback is to file an issue at https://github.com/hlevering/python-smart-injector/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 code contributions are welcome :)

5.4 Development

To set up *python-smart-injector* for local development:

1. Fork python-smart-injector (look for the "Fork" button).
2. Clone your fork locally:

```
git clone git@github.com:hlevering/python-smart-injector.git
```

3. Create a branch for local development:

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

   Now you can make your changes locally.

4. When you're done making changes run all the checks and docs builder with tox one command:

```
tox
```

5. 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
```

6. Submit a pull request through the GitHub website.

5.4.1 Pull Request Guidelines

If you need some code review or feedback while you're developing the code just make the pull request.

For merging, you should:

1. Include passing tests (run `tox`)[1].
2. Update documentation when there's new API, functionality etc.
3. Add a note to `CHANGELOG.rst` about the changes.
4. Add yourself to `AUTHORS.rst`.

5.4.2 Tips

To run a subset of tests:

```
tox -e envname -- pytest -k test_myfeature
```

To run all the test environments in *parallel* (you need to `pip install detox`):

```
detox
```

[1] If you don't have all the necessary python versions available locally you can rely on Travis - it will run the tests for each change you add in the pull request.
It will be slower though ...

CHAPTER 6

Authors

- Hendrik Levering - https://github.com/HLevering

CHAPTER 7

Changelog

7.1 0.0.0 (2019-11-20)

- First release on PyPI.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index