Mediapills Dependency Injection

Release 0.1.1

mediapills

Oct 05, 2021

TABLE OF CONTENT

Warning: This Document Page Under Construction

Dependency Injection is a technique in which an object receives other objects that it depends on, called dependencies. Typically, the receiving object is called a client and the passed-in ('injected') object is called a service. The code that passes the service to the client is called the injector. Instead of the client specifying which service it will use, the injector tells the client what service to use. The 'injection' refers to the passing of a dependency (a service) into the client that uses it.

CHAPTER

ONE

OVERVIEW

Warning: This Document Page Under Construction

Dependency Injection is a technique in which an object receives other objects that it depends on, called dependencies. Typically, the receiving object is called a client and the passed-in ('injected') object is called a service. The code that passes the service to the client is called the injector. Instead of the client specifying which service it will use, the injector tells the client what service to use. The 'injection' refers to the passing of a dependency (a service) into the client that uses it.

Dependency injection solves the following problems:

- How can a class be independent of how the objects on which it depends are created?
- How can the way objects are created be specified in separate configuration files?
- How can an application support different configurations?

Creating objects directly within the class commits the class to particular implementations. This makes it difficult to change the instantiation at runtime, especially in compiled languages where changing the underlying objects can require re-compiling the source code.

Dependency injection separates the creation of a client's dependencies from the client's behavior, which promotes loosely coupled programs and the dependency inversion and single responsibility principles. Fundamentally, dependency injection is based on passing parameters to a method.

Dependency injection is an example of the more general concept of inversion of control.

1.1 Roles

Dependency injection involves four roles:

- the service objects to be used
- the client object, whose behavior depends on the services it uses
- the interfaces that define how the client may use the services
- the injector, which constructs the services and injects them into the client

Any object that may be used can be considered a **service**. Any object that uses other objects can be considered a **client**. The names relate only to the role the objects play in an injection.

The **interfaces** are the types the client expects its dependencies to be. The client should not know the specific implementation of its dependencies, only know the interface's name and API. As a result, the client will not need to change even if what is behind the interface changes. Dependency injection can work with true interfaces or abstract classes,

but also concrete services, though this would violate the dependency inversion principle and sacrifice the dynamic decoupling that enables testing. It is only required that the client never treats its interfaces as concrete by constructing or extending them. If the interface is refactored from a class to an interface type (or vice versa) the client will need to be recompiled. This is significant if the client and services are published separately.

The **injector** introduces services to the client. Often, it also constructs the client. An injector may connect a complex object graph by treating the same object as both a client at one point and as a service at another. The injector itself may actually be many objects working together, but may not be the client (as this would create a circular dependency). The injector may be referred to as an assembler, provider, container, factory, builder, spring, or construction code.

1.2 Pros

A basic benefit of dependency injection is decreased coupling between classes and their dependencies. By removing a client's knowledge of how its dependencies are implemented, programs become more reusable, testable and maintainable.

This also results in increased flexibility: a client may act on anything that supports the intrinsic interface the client expects.

Many of dependency injection's benefits are particularly relevant to unit-testing.

For example, dependency injection can be used to externalize a system's configuration details into configuration files, allowing the system to be reconfigured without recompilation. Separate configurations can be written for different situations that require different implementations of components. This includes testing. Similarly, because dependency injection does not require any change in code behavior it can be applied to legacy code as a refactoring. The result is clients that are more independent and that are easier to unit test in isolation using stubs or mock objects that simulate other objects not under test. This ease of testing is often the first benefit noticed when using dependency injection.

More generally, dependency injection reduces boilerplate code, since all dependency creation is handled by a singular component.

Finally, dependency injection allows concurrent development. Two developers can independently develop classes that use each other, while only needing to know the interface the classes will communicate through. Plugins are often developed by third party shops that never even talk to the developers who created the product that uses the plugins.

1.3 Cons

Creates clients that demand configuration details, which can be onerous when obvious defaults are available.

Make code difficult to trace because it separates behavior from construction.

Is typically implemented with reflection or dynamic programming. This canhinder IDE automation.

Typically requires more upfront development effort.

Forces complexity out of classes and into the links between classes which might be harder to manage.

Encourage dependence on a framework.

CHAPTER

TWO

INSTALL

These instructions will install Dependency Injection package. **Dependency Injection** is a Python *package* that supports Python 3 on Linux, MacOS and Windows. We recommend using Python 3.6 or higher.

2.1 Linux and Mac OS

To install Dependency Injection package run:

```
python3 -m venv mediapills

source mediapills/bin/activate

pip install mediapills.dependency_injection
```

2.2 Windows (CMD/PowerShell)

To install Dependency Injection package on **Windows** (CMD/PowerShell)

To install Dependency Injection package run:

```
python3 -m venv mediapills

./mediapills/bin/activate

pip install mediapills.dependency_injection
```

CHAPTER

THREE

USING THE CONTAINER

Warning: This Document Page Under Construction

This documentation describes the API of the **Container** object itself.

3.1 __setitem__

You can set entries directly on the container:

```
>>> from mediapills.dependency_injection import Container

>>> di = Container()

>>> di['key'] = 'value'

>>> di['key']

'value'
```

3.2 __getitem__

You can get entries from the container:

```
>>> from mediapills.dependency_injection import Container

>>> di = Container({"key": "value"})

>>> di['key']

'value'

>>> di.get('key')

'value'

>>> di.get(None, 'default')
```

(continues on next page)

(continued from previous page)

```
'default'
```

CHAPTER

FOUR

DEFINITIONS

Warning: This Document Page Under Construction

4.1 Lazy Loading

Lazy loading (also known as *asynchronous loading*) is a **design pattern** commonly used in computer programming and mostly in web design and development to defer initialization of an object until the point at which it is needed. It can contribute to efficiency in the program's operation if properly and appropriately used.

`Container` loads the definitions you have written and uses them like instructions on how to create objects.

However those objects are only created when/if they are requested from the `Container`, for example through *container.get(...)* or when they need to be injected in another object. That means you can have a large amount of definitions, `Container` will not create all the objects unless asked to.

4.2 Definition types

This definition format is the most powerful of all. There are several kind of entries you can define:

- *Scalars*
- *Generic Container Type Objects*
- *Objects*
- *Aliases*
- *Environment Variables*
- *String Expressions*
- arrays

4.2.1 Scalars

Scalars are simple Python values:

```
>>> from mediapills.dependency_injection import Container

>>> di = Container({
...   'database.driver': 'mysql',
...   'database.host': '127.0.0.1',
...   'database.port': 80,
...   'database.auth': False,
...   'database.user': 'root',
... })

>>> di['database.host']

'127.0.0.1'
```

You can also define object entries by creating them directly:

```
>>> from mediapills.dependency_injection import Container

>>> di = Container()

>>> di['key'] = 'value'

>>> di['key']

'value'
```

However this is **not recommended** as that object will be created for every entry invocation, even if not used (it will not be lazy loaded like explained at this section).

4.2.2 Generic Container Type Objects

`Container` supports any object that holds an arbitrary number of other objects. *Examples* of containers include **tuple**, **list**, **set**, **dict**; these are the built-in containers.

```
>>> from mediapills.dependency_injection import Container

>>> di = Container()

>>> di['parameters'] = {
...   'database.host': '127.0.0.1',
...   'database.port': '80',
...   'database.user': 'root',
... }

>>> di['parameters']

{'database.host': '127.0.0.1', 'database.port': '80', 'database.user': 'root'}
```

4.2.3 Factories

Warning: This Page Section Under Construction

4.2.4 Objects

Services are defined by **anonymous functions** that return an instance of an object:

```
# define some services
container['session_storage'] = lambda di: (
    SessionStorage('SESSION_ID')
)

container['session'] = lambda di: (
    Session(di['session_storage'])
)
```

Notice that the anonymous function has access to the current container instance, allowing references to other services or parameters.

As objects are only created when you get them, the order of the definitions does not matter.

Using the defined services is also very easy:

```
# get the session object
session = injector['session']

# the above call is roughly equivalent to the following code:
# storage = SessionStorage('SESSION_ID')
# session = Session(storage)
```

4.2.5 Autowired Objects

Warning: This Page Section Under Construction

4.2.6 Aliases

You can alias an entry to another using the `Container`:

```
# define arguments container
container['arguments'] = lambda _: sys.argv

# define arguments container alias with name properties
container['properties'] = lambda di: di['arguments']
```

Allows the interface of an existing location to be used as another name.

4.2.7 Environment Variables

You can get an environment variable's value using the `Container`:

```
>>> container['env'] = lambda _: os.environ

>>> di['env'].get("LANGUAGE")

'en_US'
```

4.2.8 String Expressions

> **Warning:** This Page Section Under Construction

4.2.9 Wildcards

> **Warning:** This Page Section Under Construction

CHAPTER

FIVE

HOW TO CONTRIBUTE

There are lots of ways to contribute to the project. People with different expertise can help to improve the web content, the documentation, the code, and the tests. Even this README file is a result of collaboration of multiple people.

unit tests : A software project is as good as its tests are. Following this simple idea, we are trying to cover the integration capabilities with automated tests. If you're passionate about the quality - please consider chiming in.

doc : Have you seen a project that doesn't need to improve its documentation?!

website : We are using GH pages to build and manage the site's content. If you're interested in making it better, check-out *gh-pages* branch and dig in. If you are not familiar with the Github Pages - check it out, it's pretty simple yet powerful!

giving feedback : Tell us how you use maediapills.dependency-injection, what was great and what was not so much. Also, what are you expecting from it and what would you like to see in the future? Opening an issue will grab our attention. Seriously, this is the great way to contribute!

5.1 Roles

Much like projects in ASF, Mediapills recognizes a few roles. Unlike ASF's projects, our structure is a way simpler. There are only two types:

A Contributor is a user who contributes to a project in the form of code or documentation. Developers take extra steps to participate in a project,are active on the developer forums, participate in discussions, provide PRs (patches), documentation, suggestions, and criticism. Contributors are also known as developers.

A Committer is a developer that was given write access to the code repository. Not needing to depend on other people to commit their patches, they are actually making short-term decisions for the project. By submitting your code or other content to the project via PR or a patch, a Committer agrees to transfer the contribution rights to the Project. From time to time, the project's committership will go through the list of contributions and make a decision to invite new developers to become a project committer.

5.2 RTC model

Mediapills supports Review-Then-Commit model of development. The following rules are used in the RTC process:

- a developer should seek peer-review and/or feedback from other developers

through the PR mechanism (aka code review).

- a developer should make a reasonable effort to test the changes before

submitting a PR for review.

- any non-document PR is required to be opened for at least 24 hours for

community feedback before it gets committed unless it has an explicit +1 from a committer

- any non-document PR needs to address all the comment and reach consensus

before it gets committed without a +1 from other committers

- a committer can commit documentation patches without explicit review

process. However, we encourage you to seek the feedback.

CHAPTER

SIX

CHANGELOG

Warning: This Document Page Under Construction

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog and this project adheres to Semantic Versioning.

6.1 v0.1.0 (2021-08-23)

Minor release

6.1.1 Added

- Added module `src.mediapills.dependency_injection.exceptions` class `RecursionError`
- Added codecov integration
- Added badges: requires.io, codecov, actions, py_versions, license, downloads, wheel and codeclimate
- Added files: `LICENSE.md` and `CONTRIBUTING.md`
- Added **classifiers** and **project_urls** sections in file `setup.cfg`
- Added *py36*, *py37* and *py39* into section **envlist** in `tox.ini` file

6.1.2 Other

- Changed **mypy**, **pytest-cov** and **build** modules version
- Changed `README.rst`
- Changed value in **python_requires** section in `setup.cfg` file from *3.8* to *3.5*
- Changed `code-analysis.yml` workflow file

6.2 v0.0.2 (2021-08-22)

Patch release

6.2.1 Added

- Created decorator `handle_unknown_identifier()`
- Created module `mediapills.dependency_injection` class `Container` methods: `__getitem__()`, `__setitem__()`, `values()`, `items()`, `copy()`, `update()` and `protect()`
- Created `TestInjector` unit test case

6.2.2 Other

- Changed module `mediapills.dependency_injection` class name from `Container` to `Injector`
- Changed name from `TestContainer` to `TestContainerBase` unit test case

6.3 v0.0.1 (2021-08-21)

Minor release

6.3.1 Added

- Created `.coveragerc` file specifies python coverage configuration
- Created `.gitignore` file specifies intentionally untracked files
- Created `.pre-commit-config.yaml` file specifies pre-commit configuration
- Created *Makefile* the make utility
- Created *pyrightconfig.json* the Pyright flexible configuration
- Created python package builder `setup.py` and `setup.cfg`
- Created module `mediapills.dependency_injection` class `Container`
- Created module `src.mediapills.dependency_injection.exceptions` classes: `BaseContainerException`, `ExpectedInvokableException`, `FrozenServiceException`, `InvalidServiceIdentifierException`, `UnknownIdentifierException` and `RecursionInfiniteLoopError`
- Created unit tests case `TestContainer`
- Created virtualenv management file `tox.ini`