haps Documentation

Piotr Karkut

Apr 22, 2021

Contents:

Haps *[χaps]* is a simple DI library, with IoC container included. It is written in pure Python with no external dependencies.

Look how easy it is to use:

```
from haps import Container as IoC, Inject, inject

# import interfaces
from my_application.core import IDatabase, IUserService


class MyApp:
    db: IDatabase = Inject()  # dependency as a property

    @inject  # or passed to the constructor
    def __init__(self, user_service: IUserService) -> None:
        self.user_service = user_service

IoC.autodiscover('my_application')  # find all interfaces and implementations

if __name__ == '__main__':
    app = MyApp()
    assert isinstance(app.db, IDatabase)
    assert isinstance(app.user_service, IUserService)
```

CHAPTER 1

Features

- IoC container
- No XML/JSON/YAML - pure python configuration
- No dependencies
- Based on the Python 3.6+ annotation system

CHAPTER 2

Installation

Install *haps* by running:

```
pip install haps
```

CHAPTER 3

Contribute

- Issue Tracker: github.com/ekiro/haps/issues
- Source Code: github.com/ekiro/haps

CHAPTER 4

Changelog

4.1 1.1.1 (2018-07-27)

- Fix bug with optional arguments for functions decorated with `@inject`

4.2 1.1.0 (2018-07-26)

- Add configuration module
- Add application class and runner
- Add profiles
- Minor fixes

4.3 1.0.5 (2018-07-12)

- `@egg` decorator can be used without function invocation

4.4 1.0.4 (2018-06-30)

- Add support for python 3.7
- Fix autodiscover sample

CHAPTER 5

Support

If you are having issues, ask a question on projects issue tracker.

CHAPTER 6

License

The project is licensed under the MIT license.

6.1 QuickStart

Here's a simple tutorial on how to write your first application using *haps*. Assuming you have already created an environment with python 3.6+ and *haps* installed, you can start writing some juicy code.

6.1.1 Application layout

Since *haps* doesn't enforce any project/code design (you can use it even as an addition to your existing Django or flask application!), this is just an example layout. You are going to create a simple user registration system.

```
quickstart/
├── setup.py
└── user_module/
    ├── app.py
    ├── core
    │   ├── implementations/
    │   │   ├── __init__.py
    │   │   ├── db.py
    │   │   └── others.py
    │   ├── __init__.py
    │   └── interfaces.py
    └── __init__.py
```

6.1.2 Interfaces

Let's start with creating some interfaces, so we can keep our code clean and readable:

```
# quickstart/user_module/core/interfaces.py
from haps import base


@base
class IUserService:
    def create_user(self, username: str) -> bool:
        raise NotImplementedError

    def delete_user(self, username: str) -> bool:
        raise NotImplementedError


@base
class IDatabase:
    def add_object(self, bucket: str, name: str, data: dict) -> bool:
        raise NotImplementedError

    def delete_object(self, bucket: str, name) -> bool:
        raise NotImplementedError


@base
class IMailer:
    def send(self, email: str, message: str) -> None:
        raise NotImplementedError
```

There are three interfaces:

- `IUserService`: High-level interface with methods to create and delete users
- `IDatabase`: Low-level data repository
- `IMailer`: One-method interface for mailing integration

You need to tell *haps* about your interfaces by using `@base` class decorator, so it can resolve dependencies correctly.

Note: Be aware that you don't have to create a fully-featured interface, instead you can just define a base type, that's enough for *haps*:

```
@base
class IUserService:
    pass
```

However, it's a good practice to do so.

6.1.3 Implementations

Every interface should have at least one implementation. So, we will start with UserService and Mailer implementation.

```
# quickstart/user_module/core/implementations/others.py
from haps import egg, Inject

from user_module.core.interfaces import IDatabase, IMailer, IUserService
```

(continues on next page)

(continued from previous page)

```
@egg
class DummyMailer(IMailer):
    def send(self, email: str, message: str) -> None:
        print(f'Mail to {email}: {message}')


@egg
class UserService(IUserService):
    db: IDatabase = Inject()
    mailer: IMailer = Inject()

    _bucket = 'users'

    def create_user(self, username: str) -> bool:
        email = f'{username}@my-service.com'
        created = self.db.add_object(self._bucket, username, {
            'email': email
        })
        if created:
            self.mailer.send(email, f'Hello {username}!')
        return created

    def delete_user(self, username: str) -> bool:
        return self.db.delete_object(self._bucket, username)
```

There are two classes, and the first one is quite simple, it inherits from `IMailer` and implements its only method `send`. The only new thing here is the `@egg` decorator. You can use it to tell *haps* about any callable (a class is also a callable) that returns the implementation of a base type. Now you can probably guess how *haps* can resolve right dependencies - it looks into inheritance chain.

The `UserService` implementation is a way more interesting. Besides the parts we've already seen in the `DummyMailer` implementation, it uses the `Inject` descriptor to provide dependencies. Yes, it's that simple. You only need to define class-level field `Inject` with proper annotation, and *haps* will take care of everything else. It means creating and binding the proper instance.

Warning: With this method, the instance of an injected class, e.g., DummyMailer, is created (or fetched from the container) at the time of first property access, and then is assigned to the current `UserService` instance.

So:

```
us = UserService()
assert us.mailer is us.mailer  # it's always true
# but
assert us.mailer is UserService().mailer  # not necessarily
# (but it can, as you will see later)
```

Now let's move to our repository. We need to implement some data storage for our project. For now, it'll be in-memory storage, but, thanks to haps, you can quickly switch between many implementations. Creation of the database repository may be more complicated, so we'll use a factory function.

```
# quickstart/user_module/core/implementations/db.py
from collections import defaultdict
```

(continues on next page)

(continued from previous page)

```
from haps import egg, scope, SINGLETON_SCOPE

from user_module.core.interfaces import IDatabase


class InMemoryDb(IDatabase):
    storage: dict

    def __init__(self):
        self.storage = defaultdict(dict)

    def add_object(self, bucket: str, name: str, data: dict) -> bool:
        if name in self.storage[bucket]:
            return False
        else:
            self.storage[bucket][name] = data
            return True

    def delete_object(self, bucket: str, name) -> bool:
        try:
            del self.storage[bucket][name]
        except KeyError:
            return False
        else:
            return True


@egg
@scope(SINGLETON_SCOPE)
def database_factory() -> IDatabase:
    db = InMemoryDb()
    # Maybe do some stuff, like reading configuration
    # or create some kind of db-session.
    return db
```

`InMemoryDb` is a simple implementation of `IDatabase` that uses defaultdict to store users. It could be file-based storage or even SQL storage. However, notice there's no `@egg` decorator on this implementation. Instead, we've created a function decorated with it which have `IDatabase` declared as the return type.

In this case, when injecting, haps calls `database_factory` function and injects the result.

> **Warning:** Be aware that *haps* by design WILL NOT validate function output in any way. So if your function returns a type that's not compatible with declared one, it could lead to hard to catch errors.

6.1.4 Scope

As you can see in the previous file, `database_factory` function is also decorated with `scope` decorator.

A scope in *haps* determines object life-cycle. The default scope is `INSTANCE_SCOPE`, and you don't have to declare it explicitly. There are also two scopes that ships with haps, `SINGLETON_SCOPE`, and `THREAD_SCOPE`. You can also create your own scopes. You can read about scopes in another chapter, but for the clarity: `SINGLETON_SCOPE` means that *haps* creates only one instance, and injects the same object every time. On the other hand, dependencies with `INSTANCE_SCOPE` (which is default), are instantiated on every injection.

6.1.5 Run the code!

Now we have configured our interfaces and dependencies, and we're ready to run our application:

```
# quickstart/user_module/app.py
from haps import Container as IoC, inject

from user_module.core.interfaces import IUserService


class UserModule:
    @inject
    def __init__(self, user_service: IUserService) -> None:
        self.user_service = user_service

    def register_user(self, username: str) -> None:
        if self.user_service.create_user(username):
            print(f'User {username} created!')
        else:
            print(f'User {username} already exists!')

    def delete_user(self, username: str) -> None:
        if self.user_service.delete_user(username):
            print(f'User {username} deleted!')
        else:
            print(f'User {username} does not exists!')


IoC.autodiscover(['user_module.core'])

if __name__ == '__main__':
    um = UserModule()
    um.register_user('Kiro')
    um.register_user('John')
    um.register_user('Kiro')
    um.delete_user('Kiro')
    um.delete_user('Kiro')
    another_um_instance = UserModule()
    another_um_instance.register_user('John')
```

The main class `UserModule` takes `IUserService` in the constructor, and thanks to the `@inject` decorator, haps will create and pass `UserService` instance to it.

After that, we have to call `autodiscover` method from *haps*, which scans all modules under given path and configures all dependencies.

Running our application should give following output:

```
Mail to Kiro@my-service.com: Hello Kiro!
User Kiro created!
Mail to John@my-service.com: Hello John!
User John created!
User Kiro already exists!
User Kiro deleted!
User Kiro does not exists!
User John already exists!
```

6.2 Profiles

Haps allows you to attach dependencies to configuration profile. It helps with development, testing, and some other stuff. You can set active profiles using *Configuration*.

6.2.1 Example

One of many good use cases for profiles is mailing. Imagine you have to implement mailer class. Your production environment uses AWS SES, stage uses an internal SMTP system, on your local env every mail is printed to stdout, and mailer for tests do nothing. You may ask, how to implement this without nasty ifs? Well, it's quite easy with profiles:

```
from haps import base, egg


@base
class IMailer
    def send(self, to: str, message: str) -> None:
        raise NotImplementedError


@egg(profile='production')
class SESMailer(IMailer):
    def send(self, to: str, message: str) -> None:
        # SES implementation


@egg(profile='stage')
class SMTPMailer(IMailer):
    def send(self, to: str, message: str) -> None:
        # SMTP implementation


@egg(profile='tests')
class DummyMailer(IMailer):
    def send(self, to: str, message: str) -> None:
        pass


@egg  # missing profile means default
class LogMailer(IMailer):
    def send(self, to: str, message: str) -> None:
        print(f"Mail to {to}: {message}")
```

And that's it. Now you only need to run your app with `HAPS_PROFILES=production` (or any other profile) and haps will choose proper dependency. You can set more than one profile separating them by a comma: `HAPS_PROFILES=stage,local-static,sqlite`

If there is more than one egg for the given profiles list, the order decides about priority. e.g. for `HAPS_PROFILES=tests,production` the `DummyMailer` class is chosen.

Profiles can be configured programmatically, **before** Container configuration:

```
from haps import PROFILES
from haps.config import Configuration

```

(continues on next page)

(continued from previous page)

```
Configuration().set(PROFILES, ('tests', 'tmp-static', 'sqlite'))
# Container config / autodiscover
```

Note: Profiles that are set directly by *Configuration* overrides profiles from the environment variable.

6.3 API

6.3.1 Container

class haps.**Container**
Dependency Injection container class

Container is a heart of *haps*. For now, its implemented as a singleton that can only be used after one-time configuration.

```
from haps import Container

Container.autodiscover(['my.package'])  # configuration, once in the app lifetime
Container().some_method()  # Call method on the instance
```

That means, you can create instances of classes that use injections, only after *haps* is properly configured.

classmethod Container.**autodiscover**(*module_paths: List[str], subclass: Optional[haps.container.Container] = None*) → None
Load all modules automatically and find bases and eggs.

Parameters

- **module_paths** – List of paths that should be discovered
- **subclass** – Optional Container subclass that should be used

static Container.**configure**(*config: List[haps.container.Egg], subclass: Optional[haps.container.Container] = None*) → None
Configure haps manually, an alternative to *autodiscover()*

Parameters

- **config** – List of configured Eggs
- **subclass** – Optional Container subclass that should be used

Container.**get_object**(*base_: Type, qualifier: str = None*) → Any
Get instance directly from the container.

If the qualifier is not None, proper method to create/retrieve instance is used.

Parameters

- **base** – *base* of this object
- **qualifier** – optional qualifier

Returns object instance

Container.**register_scope**(*name: str, scope_class: Type[haps.scopes.Scope]*) → None
Register new scopes which should be subclasses of *Scope*

Parameters

- **name** – Name of new scopes
- **scope_class** – Class of new scopes

6.3.2 Egg

class haps.**Egg**(*base_: Optional[Type], type_: Type, qualifier: Optional[str], egg_: Callable, profile: str = None*)

Configuration primitive. Can be used to configure *haps* manually.

Egg.**__init__**(*base_: Optional[Type], type_: Type, qualifier: Optional[str], egg_: Callable, profile: str = None*) → None

Parameters

- **base** – *base* of dependency, used to retrieve object
- **type** – *type* of dependency (for functions it's a return type)
- **qualifier** – extra qualifier for dependency. Can be used to register more than one type for one base.
- **egg** – any callable that returns an instance of dependency, can be a class or a function
- **profile** – dependency profile name

6.3.3 Injection

class haps.**Inject**(*qualifier: str = None*)

A descriptor for injecting dependencies as properties

```
class SomeClass:
    my_dep: DepType = Inject()
```

Important: Dependency is injected (created/fetched) at the moment of accessing the attribute, not at the moment of instance creation. So, even if you create an instance of *SomeClass*, the instance of *DepType* may never be created.

---

haps.**inject**(*fun: Callable*) → Callable

A decorator for injection dependencies into functions/methods, based on their type annotations.

```
class SomeClass:
    @inject
    def __init__(self, my_dep: DepType) -> None:
        self.my_dep = my_dep
```

Important: On the opposite to *Inject*, dependency is injected at the moment of method invocation. In case of decorating *__init__*, dependency is injected when *SomeClass* instance is created.

Parameters **fun** – callable with annotated parameters

Returns decorated callable

6.3.4 Dependencies

haps.**base**(*cls: T*) → T
A class decorator that marks class as a base type.

Parameters **cls** – Some base type

Returns Not modified *cls*

haps.**egg**(*qualifier: Union[str, Type] = ''*, *profile: str = None*)
A function that returns a decorator (or acts like a decorator) that marks class or function as a source of *base*.

If a class is decorated, it should inherit from *base* type.

If a function is decorated, it declared return type should inherit from some *base* type, or it should be the *base* type.

```
@egg
class DepImpl(DepType):
    pass

@egg(profile='test')
class TestDepImpl(DepType):
    pass

@egg(qualifier='special_dep')
def dep_factory() -> DepType:
    return SomeDepImpl()
```

Parameters

- **qualifier** – extra qualifier for dependency. Can be used to register more than one type for one base. If non-string argument is passed, it'll act like a decorator.
- **profile** – An optional profile within this dependency should be used

Returns decorator

haps.**scope**(*scope_type: str*) → Callable
A function that returns decorator that set scopes to some class/function

```
@egg()
@scopes(SINGLETON_SCOPE)
class DepImpl:
    pass
```

Parameters **scope_type** – Which scope should be used

Returns

6.3.5 Configuration

class haps.config.**Configuration**
Configuration container, a simple object to manage application config variables. Variables can be set manually, from the environment, or resolved via custom function.

Configuration.**get_var**(*var_name: str*, *default: Optional[Any] = <object object>*) → Any
Get a config variable. If a variable is not set, a resolver is not set, and no default is given *UnknownConfigVariable* is raised.

Parameters

- **var_name** – Name of variable
- **default** – Default value

Returns Value of config variable

classmethod Configuration.**resolver**(*var_name: str*) → function
Variable resolver decorator. Function or method decorated with it is used to resolve the config variable.

Note: Variable is resolved only once. Next gets are returned from the cache.

Parameters **var_name** – Variable name

Returns Function decorator

classmethod Configuration.**env_resolver**(*var_name: str*, *env_name: str = None*, *default: Any = <object object>*) → haps.config.Configuration
Method for configuring environment resolver.

Parameters

- **var_name** – Variable name
- **env_name** – An optional environment variable name. If not set haps looks for *HAPS_var_name*
- **default** – Default value for variable. If it's a callable, it is called before return. If not provided *UnknownConfigVariable* is raised

Returns *Configuration* instance for easy chaining

classmethod Configuration.**set**(*var_name: str*, *value: Any*) → haps.config.Configuration
Set the variable

Parameters

- **var_name** – Variable name
- **value** – Value of variable

Returns *Configuration* instance for easy chaining

class haps.config.**Config**(*var_name: str = None*, *default=<object object>*)
Descriptor providing config variables as a class properties.

```
class SomeClass:
    my_var: VarType = Config()
    custom_property_name: VarType = Config('var_name')
```

Config.**__init__**(*var_name: str = None*, *default=<object object>*) → None

Parameters

- **var_name** – An optional variable name. If not set the property name is used.
- **default** – Default value for variable. If it's a callable, it is called before return. If not provided *UnknownConfigVariable* is raised

6.4 Scopes

A scope is a special object that controls dependency creation. It decides if new dependency instance should be created, or some cached instance should be returned.

By default, there are two scopes registered in haps: `InstanceScope` and `SingletonScope` as `haps.INSTANCE_SCOPE` and `haps.SINGLETON_SCOPE`. The `haps.INSTANCE_SCOPE` is used as a default.

You can register any other scope by calling *haps.Container.register_scope()*. New scopes should be a subclass of *haps.scopes.Scope*.

class haps.scopes.**Scope**
Base scope class. Every custom scope should subclass this.

class haps.scopes.instance.**InstanceScope**
Dependencies within InstanceScope are created at every injection.

class haps.scopes.singleton.**SingletonScope**
Dependencies within SingletonScope are created only once in the application context.

class haps.scopes.thread.**ThreadScope**
Dependencies within ThreadScope are created only once in a thread context.

6.5 Exceptions

exception haps.exceptions.**AlreadyConfigured**

exception haps.exceptions.**ConfigurationError**

exception haps.exceptions.**NotConfigured**

exception haps.exceptions.**UnknownDependency**

exception haps.exceptions.**UnknownScope**

exception haps.exceptions.**CallError**

exception haps.exceptions.**UnknownConfigVariable**

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Index

Symbols

A

B

C

E

G

I

N

R

S

T

U