PyDio

Release 0.2.0

Maciej Wiatrzyk <maciej.wiatrzyk@gmail.com>

Nov 01, 2022

CONTENTS

CHAPTER

ONE

ABOUT PYDIO

PyDio is a dependency injection library for Python.

It aims to be simple, yet still powerful, allowing you to feed dependencies inside your application in a flexible way. PyDio design is based on simple assumption, that dependency injection can be achieved using simple **key-to-function** map, where **key** specifies **type of object** you want to inject and **function** is a **factory** function that creates **instances** of that type.

In PyDio, this is implemented using **providers** and **injectors**. You use providers to configure your **key-to-function** mapping, and then you use injectors to perform a **lookup** of a specific key and creation of the final object.

Here's a simple example:

```
import abc

from pydio.api import Provider, Injector

provider = Provider()

@provider.provides('greet')
def make_greet():
    return 'Hello, world!'

def main():
    injector = Injector(provider)
    greet_message = injector.inject('greet')
    print(greet_message)
```

And if you now call `main()` function, then the output will be following:

```
Hello, world!
```

CHAPTER

TWO

KEY FEATURES

- Support for any hashable keys: class objects, strings, ints etc.
- Support for any type of object factories: function, coroutine, generator, asynchronous generator.
- Automatic resource management via generator-based factories (similar to pytest's fixtures)
- Multiple environment support: testing, development, production etc.
- Limiting created object's lifetime to user-defined scopes: global, application, use-case etc.
- No singletons used, so there is no global state…
- …but you still can create global injector on your own if you need it :-)

CHAPTER

THREE

USER'S GUIDE

3.1 Installation

You can install PyDio using one of following methods:

1) From PyPI (for stable releases):

```
$ pip install PyDio
```

2) From test PyPI (for stable and development releases):

```
$ pip install -i https://test.pypi.org/simple/ PyDio
```

3) Directly from source code repository (for all releases):

```
$ pip install git+https://gitlab.com/zef1r/PyDio.git@[branch-or-tag]
```

3.2 Quickstart

3.2.1 Introduction

In this quickstart guide, we are going to write a simple TODO application that allows:

- creating items,
- listing items,
- marking items as completed,
- deleting items

3.2.2 Application's business logic

First, we need a data class to represent our todo items. Let's then start by creating a **TodoItem** entity:

```
import uuid
from datetime import datetime

class TodoItem:
    uid: uuid.UUID
```

(continues on next page)

(continued from previous page)

```
    created: datetime
    title: str
    description: str
    done: bool = False
```

Now we need some kind of storage where our todo items will be stored. We will do this formally, by designing interface. Of course we don't need it (it's a Python), but interfaces are pretty useful with annotations. Here's our TODO item storage interface:

```
import abc
from typing import Iterable, Optional

class ITodoItemStorage(abc.ABC):

    @abc.abstractmethod
    def create(self, item: TodoItem):
        pass

    @abc.abstractmethod
    def save(self, item: TodoItem):
        pass

    @abc.abstractmethod
    def get(self, item_uuid: uuid.UUID) -> Optional[TodoItem]:
        pass

    @abc.abstractmethod
    def delete(self, item_uuid: uuid.UUID):
        pass

    @abc.abstractmethod
    def list(self) -> Iterable[TodoItem]:
        pass
```

Finally, let's write our use case classes:

```
class CreateTodo:

    def __init__(self, todo_storage: ITodoItemStorage):
        self._todo_storage = todo_storage

    def invoke(self, title, description):
        item = TodoItem()
        item.uuid = uuid.uuid4()
        item.created = datetime.now()
        item.title = title
        item.description = description
        item.done = False
        self._todo_storage.create(item)

class ListTodos:
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, todo_storage: ITodoItemStorage):
        self._todo_storage = todo_storage

    def invoke(self):
        for item in self._todo_storage.list():
            yield {  # we don't want to expose our entity
                'uuid': item.uuid,
                'created': item.created,
                'title': item.title,
                'description': item.description,
                'done': item.done
            }

class CompleteTodo:

    def __init__(self, todo_storage: ITodoItemStorage):
        self._todo_storage = todo_storage

    def invoke(self, item_uuid: uuid.UUID):
        item = self._todo_storage.get(item_uuid)
        if item is None:
            raise ValueError("invalid item uuid: {}".format(item_uuid))
        item.done = True
        self._todo_storage.save(item)

class DeleteTodo:

    def __init__(self, todo_storage: ITodoItemStorage):
        self._todo_storage = todo_storage

    def invoke(self, item_uuid: uuid.UUID):
        self._todo_storage.delete(item_uuid)
```

And that's entire business logic of our simple TODO application. But so far, we were only using a suite of unit tests, with **ITodoItemStorage** interface mocked. Now, let's put some life into our application.

3.2.3 Application's API

To make our business logic running we cannot use mocks any longer - now we need a real implementation of **ITodoItemStorage** interface. Since we are still doing development of our application, we still don't have to use any SQL databases - just a simple in-memory store will do. Here's a very basic implementation:

```
class InMemoryTodoStorage(ITodoItemStorage):

    def __init__(self):
        self._todos = {}

    def create(self, item):
        self._todos[item.uuid] = item

    def save(self, item):
```

(continues on next page)

(continued from previous page)

```
        self._todos[item.uuid] = item

    def delete(self, item_uuid):
        del self._todos[item_uuid]

    def get(self, item_uuid):
        return self._todos.get(item_uuid)

    def list(self):
        for item in self._todos.values():
            yield item
```

Now we can use it in our application. It will be represented by **TodoApplication** class, with all use cases exposed as methods:

```
from typing import List

class TodoApplication:

    def __init__(self):
        self._todo_storage = InMemoryTodoStorage()

    def create(self, title: str, description: str):
        CreateTodo(self._todo_storage).invoke(title, description)

    def complete(self, item_uuid: uuid.UUID):
        CompleteTodo(self._todo_storage).invoke(item_uuid)

    def list(self) -> List[dict]:
        return [x for x in ListTodos(self._todo_storage).invoke()]

    def delete(self, item_uuid: uuid.UUID):
        DeleteTodo(self._todo_storage).invoke(item_uuid)
```

And here's how it works:

```
>>> app = TodoApplication()
>>> app.create('shopping', 'buy some milk')
>>> items = app.list()
>>> items
[{'uuid': ..., 'created': ..., 'title': 'shopping', 'description': 'buy some milk', 'done
↪': False}]
>>> app.complete(items[0]['uuid'])
>>> app.list()
[{'uuid': ..., 'created': ..., 'title': 'shopping', 'description': 'buy some milk', 'done
↪': True}]
>>> app.delete(items[0]['uuid'])
>>> app.list()
[]
```

3.2.4 Adding another environment

Okay, so we have our basic scenario working in development environment. But to make it work in production, we need some non-volatile storage. Therefore, we need another implementation. Let it be a some kind of SQL database:

```
import sqlite3

class SQLiteDatabase:

    def __init__(self, db_name):
        self._db_name = db_name

    def connect(self):
        connection = sqlite3.connect(self._db_name)
        c = connection.cursor()
        c.execute("""CREATE TABLE IF NOT EXISTS todos (
            uuid UUID PRIMARY KEY,
            created DATETIME,
            title TEXT,
            description TEXT,
            done BOOLEAN)""")
        connection.commit()
        return connection

class SQLiteTodoStorage(ITodoItemStorage):

    def __init__(self, connection):
        self._conn = connection

    def create(self, item):
        c = self._conn.cursor()
        c.execute(
            "INSERT INTO todos VALUES (?, ?, ?, ?, ?)",
            [str(item.uuid), item.created, item.title, item.description,
            item.done])

    def save(self, item):
        c = self._conn.cursor()
        c.execute("UPDATE todos SET done=?", [item.done])  # Just for our case
        self._conn.commit()

    def delete(self, item_uuid):
        c = self._conn.cursor()
        c.execute("DELETE FROM todos WHERE uuid=?", [str(item_uuid)])

    def get(self, item_uuid):
        c = self._conn.cursor()
        c.execute("SELECT * FROM todos WHERE uuid=?", [str(item_uuid)])
        row = c.fetchone()
        return self._make_todo(row)

    def list(self):
        c = self._conn.cursor()
```

(continues on next page)

(continued from previous page)

```
        c.execute("SELECT * FROM todos")
        for row in c.fetchmany():
            yield self._make_todo(row)

    def _make_todo(self, row):
        item = TodoItem()
        item.uuid = row[0]
        item.created = row[1]
        item.title = row[2]
        item.description = row[3]
        item.done = True if row[4] else False
        return item
```

And now, let's modify our original application. But this time, we need both storages at once! We'll decide which one to use by giving environment name to **TodoApplication**'s constructor:

```
from typing import List

class TodoApplication:

    def __init__(self, env):
        if env == 'production':
            self._database = SQLiteDatabase(':memory:')
            self._todo_storage = SQLiteTodoStorage(self._database.connect())
        else:
            self._todo_storage = InMemoryTodoStorage()

    def create(self, title: str, description: str):
        CreateTodo(self._todo_storage).invoke(title, description)

    def complete(self, item_uuid: uuid.UUID):
        CompleteTodo(self._todo_storage).invoke(item_uuid)

    def list(self) -> List[dict]:
        return [x for x in ListTodos(self._todo_storage).invoke()]

    def delete(self, item_uuid: uuid.UUID):
        DeleteTodo(self._todo_storage).invoke(item_uuid)
```

As you can see, the code gets more complicated. And this is only one interface with just only two implementations! Let's see how this works:

```
>>> app = TodoApplication('production')
>>> app.create('shopping', 'buy some milk')
>>> items = app.list()
>>> items
[{'uuid': ..., 'created': ..., 'title': 'shopping', 'description': 'buy some milk', 'done
↪': False}]
>>> app.complete(items[0]['uuid'])
>>> app.list()
[{'uuid': ..., 'created': ..., 'title': 'shopping', 'description': 'buy some milk', 'done
↪': True}]
```

(continues on next page)

(continued from previous page)

```
>>> app.delete(items[0]['uuid'])
>>> app.list()
[]
```

3.2.5 Introducing providers

As you can see, when implementing additional storages, our business logic was not affected at all, however configuration part of our application was getting more complicated. Now let's do some refactoring with PyDio.

First, we need to create **providers**. Providers are used to wrap user-defined factory functions and give it a key that can be referenced later. Here are providers for our two previously created storages:

```
from pydio.api import Provider

provider = Provider()

@provider.provides(ITodoItemStorage)
def make_in_memory_todo_storage():  # (1)
    return InMemoryTodoStorage()

@provider.provides(ITodoItemStorage, env='production')
def make_sqlite_todo_storage():  # (2)
    database = SQLiteDatabase(':memory:')
    return SQLiteTodoStorage(database.connect())
```

We have created two object factories with a key set in both to **ITodoItemStorage** - our interface created earlier. Object factory (1) will be used as a default for that key, while (2) will only be used for production environment. Of course, environment names are not predefined - you can set it to anything you like. The only requirement is to use same name later.

3.2.6 Introducing injectors

Now let me introduce second element of PyDio library - the **injector**. Here's our TODO application from earlier example refactored to use injector:

```
from pydio.api import Injector  # (1)

class TodoApplication:

    def __init__(self, env):
        self._injector = Injector(provider, env=env)  # (2)

    @property
    def _todo_storage(self):
        return self._injector.inject(ITodoItemStorage)  # (3)

    def create(self, title: str, description: str):
        CreateTodo(self._todo_storage).invoke(title, description)

    def complete(self, item_uuid: uuid.UUID):
        CompleteTodo(self._todo_storage).invoke(item_uuid)
```

(continues on next page)

(continued from previous page)

```
    def list(self) -> List[dict]:
        return [x for x in ListTodos(self._todo_storage).invoke()]

    def delete(self, item_uuid: uuid.UUID):
        DeleteTodo(self._todo_storage).invoke(item_uuid)
```

And now a brief explanation:

- First, we need to import *pydio.injector.Injector* class (1)
- Now we have to create instance of that class. We need to pass provider created earlier and environment given from the outside (2). Our newly created injector will later use given provider and environment to find matching factory.
- And finally (3), we use *pydio.injector.Injector.inject()* method to perform injections. We use same key as previously in provider, and environment passed in constructor will be used implicitly to find matching variant of our factory.

As you can see, the code of our application is much simpler after refactoring. Moreover, we can easily attach another implementation of our storage - we just need to create another factory, and decorate it with same key, but different environment. Here's an example that uses mock this time:

```
from mockify.mock import ABCMock

@provider.provides(ITodoItemStorage, env='testing')
def make_storage_mock():
    return ABCMock('storage_mock', ITodoItemStorage)
```

And now, let's run our **unchanged** application code, but giving it an environment we've just used:

```
>>> app = TodoApplication('testing')
>>> app.create('shopping', 'buy some milk')
Traceback (most recent call last):
    ...
mockify.exc.UninterestedCall: No expectations recorded for mock:

at <doctest default[0]>:13
--------------------------
Called:
  storage_mock.create(<TodoItem object at ...>)
```

As you can see, our mock was now triggered - not in-memory, neither SQLite storage.

Note: The call failed with exception, because we did not record any expectations - that's default behaviour for Mockify. Please proceed to https://mockify.readthedocs.io/en/latest/ if you want to read more about Mockify - my other project.

3.2.7 Using nested injections

Our example is rather trivial. In real life projects there are often much more dependencies to be injected, and sometimes it is event necessary to inject dependencies to the object that is being injected as well (nested injections). To show how this works, let's first extract our use case class constructors out of the application and use provider to provide those as well. Of course, our use cases will still need a storage, so we will have to use nested injections:

```
provider = Provider()

@provider.provides(ITodoItemStorage)
def make_in_memory_todo_storage():
    return InMemoryTodoStorage()

@provider.provides(ITodoItemStorage, env='testing')
def make_storage_mock():
    return ABCMock('storage_mock', ITodoItemStorage)

@provider.provides(ITodoItemStorage, env='production')
def make_sqlite_todo_storage():
    database = SQLiteDatabase(':memory:')
    return SQLiteTodoStorage(database.connect())

@provider.provides(CreateTodo)
def make_create_todo(injector: Injector): # (1)
    return CreateTodo(injector.inject(ITodoItemStorage))  # (2)

@provider.provides(CompleteTodo)
def make_complete_todo(injector: Injector):
    return CompleteTodo(injector.inject(ITodoItemStorage))

@provider.provides(ListTodos)
def make_list_todos(injector: Injector):
    return ListTodos(injector.inject(ITodoItemStorage))

@provider.provides(DeleteTodo)
def make_delete_todos(injector: Injector):
    return DeleteTodo(injector.inject(ITodoItemStorage))
```

And now some explanation:

- First, we need to add argument for passing current injector to our factory function. All supported arguments are:
  - **injector** - for passing current injector (the one that owns that object factory)
  - **key** - for passing key assigned to that factory (**CreateTodo** in this case)
  - **env** - for passing environment name

  These names are reserved currently, however the order may be changed - you can pick from 0-3 arguments out of that predefined ones depending on your needs. In other words, this works similarly to PyTest's fixtures.
- And finally (2), we use **injector** just like in our application class earlier.

Okay, we have our provider configured, so let's now rewrite our application again. This time we'll use injector to inject use case classes only:

```
class TodoApplication:

    def __init__(self, env):
        self._injector = Injector(provider, env=env)

    def create(self, title: str, description: str):
        self._injector.inject(CreateTodo).invoke(title, description)

    def complete(self, item_uuid: uuid.UUID):
        self._injector.inject(CompleteTodo).invoke(item_uuid)

    def list(self) -> List[dict]:
        return [x for x in self._injector.inject(ListTodos).invoke()]

    def delete(self, item_uuid: uuid.UUID):
        self._injector.inject(DeleteTodo).invoke(item_uuid)
```

3.2.8 Using scopes

The solution we've prepared so far would not work in real situations unless we create different application object for every action. That is due to the fact, that each object factory is **called only once** per injector's lifetime. And since we create injector in application's constructor, we would have to call it (the constructor) again for every method call - otherwise we would start sharing our objects between API calls, and that may not be expected behavior.

To solve this issue, PyDio provides **scopes**. Scopes are implemented by creating new injector from given one, and giving the new one access to user-defined scope, plus its ancestors. Such created injectors can have shorter lifetime than the root one.

But we also need to set scopes when factory functions are registered to provider - just like we did for environments:

```
provider = Provider()

@provider.provides(ITodoItemStorage, scope='app')
def make_in_memory_todo_storage():
    return InMemoryTodoStorage()

@provider.provides(ITodoItemStorage, env='testing', scope='app')
def make_storage_mock():
    return ABCMock('storage_mock', ITodoItemStorage)

@provider.provides(ITodoItemStorage, env='production', scope='app')
def make_sqlite_todo_storage():
    database = SQLiteDatabase(':memory:')
    return SQLiteTodoStorage(database.connect())

@provider.provides(CreateTodo, scope='action')
def make_create_todo(injector: Injector):
    return CreateTodo(injector.inject(ITodoItemStorage))

@provider.provides(CompleteTodo, scope='action')
def make_complete_todo(injector: Injector):
    return CompleteTodo(injector.inject(ITodoItemStorage))
```

(continues on next page)

(continued from previous page)

```
@provider.provides(ListTodos, scope='action')
def make_list_todos(injector: Injector):
    return ListTodos(injector.inject(ITodoItemStorage))

@provider.provides(DeleteTodo, scope='action')
def make_delete_todos(injector: Injector):
    return DeleteTodo(injector.inject(ITodoItemStorage))
```

We've registered our factories using two scopes: *app* and *action*. Now, let's change our application class to something like this:

```
injector = Injector(provider)  # (1)

class TodoApplication:

    def __init__(self, env):
        self._injector = injector.scoped('app', env=env)  # (2)

    def create(self, title: str, description: str):
        with self._injector.scoped('action') as injector:  # (3)
            injector.inject(CreateTodo).invoke(title, description)

    def complete(self, item_uuid: uuid.UUID):
        with self._injector.scoped('action') as injector:
            injector.inject(CompleteTodo).invoke(item_uuid)

    def list(self) -> List[dict]:
        with self._injector.scoped('action') as injector:
            return [x for x in injector.inject(ListTodos).invoke()]

    def delete(self, item_uuid: uuid.UUID):
        with self._injector.scoped('action') as injector:
            injector.inject(DeleteTodo).invoke(item_uuid)

    def shutdown(self):
        self._injector.close()
```

And now some explanation:

- We've created a root injector at (1)
- Then, in our application, we've created a **scoped** injector from our root and named it *app* - it will be application-wide. This injector will be able to use object factories:
  - that does not have scope assigned,
  - that has *app* scope assigned.

  All other will not be accessible from there.
- Finally, in our actions we've created another scoped injector, from our application's one, and named it with a scope *action* (3). This injector will be able to use object factories:
  - that does not have scope assigned,
  - that have *app* scope assigned (as it is a child of *app* scoped injector),

  - that have *action* scope assigned.

  And - like previously - all other will not be accessible.

- The lifetime of each injector is:
  - Same as for process (root injector)
  - Until `shutdown()` is called (*app* injector)
  - Until we are under context manager (each *action* injector)

3.2.9 Using generator-based object factories

We are still missing one important thing in our application - database sessions. Of course, that is not needed for a in-memory storage, but definitely will have to be used for SQL-based storage. And the session scope should be limited only to actions. How to do that using PyDio? Here's a solution:

```
provider = Provider()

@provider.provides('database', env='production', scope='app')  # (1)
def make_database():
    return SQLiteDatabase(':memory:').connect()

@provider.provides(ITodoItemStorage, env='production', scope='action')
def make_sqlite_todo_storage(injector):
    connection = injector.inject('database')  # (2)
    try:
        yield SQLiteTodoStorage(connection)  # (3)
    except Exception:
        connection.close()
    else:
        connection.commit()
```

This time, we've extracted making database to a separate factory function (1) and changed the scope for `make_sqlite_todo_storage` function to *action*. Notice, that the scope of `make_database` function is still set to *app*, so database object will be bound to *app* injector and reused by all *action* injectors. There is one more important thing: we've used a **generator** in (3). Thanks to this, we were able to customize cleanup behavior for that particular factory to either do a commit, or a rollback - in similar way as in PyTest fixtures.

That will work with unchanged application code from previous example.

3.2.10 Using multiple providers

Sometimes single provider object may not be good enough. Especially, when there are dozens of object factory functions to be registered, possible in several separate modules. For example, based on our application, different module for storages and different for use cases may be needed at some point in time. So now let's rewrite our application to use two different provider objects.

We'll start by creating module for our storage provider. It will look like this:

```
from pydio.api import Provider

storage_provider = Provider()

```

(continues on next page)

(continued from previous page)

```
@storage_provider.provides(ITodoItemStorage, scope='app')
def make_in_memory_todo_storage():
    return InMemoryTodoStorage()

@storage_provider.provides(ITodoItemStorage, env='testing', scope='app')
def make_storage_mock():
    return ABCMock('storage_mock', ITodoItemStorage)

@storage_provider.provides('database', env='production', scope='app')
def make_database():
    return SQLiteDatabase(':memory:').connect()

@storage_provider.provides(ITodoItemStorage, env='production', scope='action')
def make_sqlite_todo_storage(injector):
    connection = injector.inject('database')  # (2)
    try:
        yield SQLiteTodoStorage(connection)  # (3)
    except Exception:
        connection.close()
    else:
        connection.commit()
```

And now, let's make separate module for our use case provider:

```
from pydio.api import Provider

use_case_provider = Provider()

@use_case_provider.provides(CreateTodo, scope='action')
def make_create_todo(injector: Injector):
    return CreateTodo(injector.inject(ITodoItemStorage))

@use_case_provider.provides(CompleteTodo, scope='action')
def make_complete_todo(injector: Injector):
    return CompleteTodo(injector.inject(ITodoItemStorage))

@use_case_provider.provides(ListTodos, scope='action')
def make_list_todos(injector: Injector):
    return ListTodos(injector.inject(ITodoItemStorage))

@use_case_provider.provides(DeleteTodo, scope='action')
def make_delete_todos(injector: Injector):
    return DeleteTodo(injector.inject(ITodoItemStorage))
```

To make a use of those two distinct providers we just need to create yet another provider and attach previously created two providers to it using *pydio.provider.Provider.attach()* method:

```
provider = Provider()
provider.attach(storage_provider)
provider.attach(use_case_provider)

injector = Injector(provider)
```

(continues on next page)

(continued from previous page)

```
class TodoApplication:

    def __init__(self, env):
        self._injector = injector.scoped('app', env=env)

    def create(self, title: str, description: str):
        with self._injector.scoped('action') as injector:
            injector.inject(CreateTodo).invoke(title, description)

    def complete(self, item_uuid: uuid.UUID):
        with self._injector.scoped('action') as injector:
            injector.inject(CompleteTodo).invoke(item_uuid)

    def list(self) -> List[dict]:
        with self._injector.scoped('action') as injector:
            return [x for x in injector.inject(ListTodos).invoke()]

    def delete(self, item_uuid: uuid.UUID):
        with self._injector.scoped('action') as injector:
            injector.inject(DeleteTodo).invoke(item_uuid)

    def shutdown(self):
        self._injector.close()
```

3.3 API Reference

3.3.1 pydio.api - All core classes in one place

An all-in-one module for making imports easier.

You can use this in your code to create one-line imports. For example, instead of adding multiple PyDio imports to your application, you can do this instead:

```
from pydio.api import Injector, Provider
```

class pydio.api.**Injector**(*provider:* IUnboundFactoryRegistry, *env: Optional[Hashable] = None*)

Dependency injector main class.

Parameters

- **provider** – Unbound factory provider to work on
- **env** – Name of the environment this injector will use when making queries to `IUnboundFactoryRegistry` object given via **provider**.

  This can be obtained f.e. from environment variable. Once injector is created you will not be able to change this.

  See `IUnboundFactoryRegistry.get()` for more details.

exception AlreadyClosedError(***kwargs*)

Raised when operation on a closed injector was performed.

exception NoProviderFoundError(*key*, *env*)

Raised when there was no matching provider found for given key.

Parameters

- **key** – Searched key
- **env** – Searched environment

exception OutOfScopeError(*key*, *scope*, *required_scope*)

Raised when there was attempt to create object that was registered for different scope.

Parameters

- **key** – Searched key
- **scope** – Injector's own scope
- **required_scope** – Required scope

close() → Optional[Awaitable[None]]

See *pydio.base.IInjector.close()*.

property env: Optional[Hashable]

Environment assigned to this injector.

inject(*key*)

See IInjector.inject.

is_closed() → bool

Return True if this injector was closed or False otherwise.

scoped(*scope: Hashable*, *env: Optional[Hashable] = None*) → *IInjector*

See *pydio.base.IInjector.scoped()*.

class pydio.api.**Provider**

Used to record user-defined object factories or instances and bind them with particular key, that can later be used by IInjector.inject().

exception DoubleRegistrationError(*key*, *env*)

Raised when same (key, env) tuple was used twice during registration.

Parameters

- **key** – Registered key
- **env** – Registered environment

attach(*provider: Provider*)

Attach given provider to this provider.

This effectively extends current provider with object factories registered to the other one.

Use this if you need to split your providers across multiple modules.

get(*key*, *env=None*)

See IUnboundFactoryRegistry.get().

has_awaitables()

See IUnboundFactoryRegistry.has_awaitables().

provides(*key*, *scope=None*, *env=None*)

Same as *register_func()*, but to be used as a decorator.

Here's an example:

```
from pydio.api import Provider

provider = Provider()

@provider.provides('spam')
def make_spam():
    return 'give me more spam'
```

register_func(*key*, *func*, *scope=None*, *env=None*)

Register user factory function.

Parameters

- **key** – Key to be used for **func**.

  See `IInjector.inject()` for more info.

- **func** – User-defined function to be registered.

  This can be normal function, coroutine, generator or async denerator.

- **scope** – Optional scope to be assigned.

- **env** – Optional environment to be assigned

register_instance(*key*, *value*, *scope=None*, *env=None*)

Same as *register_func()*, but for registration of constant objects.

If your application has some global configuration data you want to inject using PyDio - that's the method you should use.

class pydio.api.**Variant**(*key: Hashable*, ***kwargs: Hashable*)

A special form of key that can have user-defined parameters attached.

This class can be used if you need to use same key twice, but return different objects depending on additional parameters given (which can be accessed by object factory)

Parameters

- **key** – The key to be wrapped
- **kwargs** – Additional parameters to be bound with given key

property key

Wrapped key.

property kwargs: dict

Dict with parameters given in constructor.

3.3.2 pydio.base - Interface definitions

Interface definitions.

class pydio.base.**IFactory**

Interface for bound factories.

Bound factories are managed by *IInjector* objects and are responsible for construction of target object that is later returned by *IInjector.inject()*. Each factory should wrap one kind of object factory function provided by user (f.e. normal function or a coroutine, but never both).

abstract close()

Close this factory.

When called, underlying instance is cleared and calling *get_instance()* again will return None. This method may also invoke some additional custom-defined clearing actions (if supported by implementation).

abstract get_instance() → Optional[Union[T, Awaitable[T]]]

Create and return target object.

Value returned by this method is later also returned by *IInjector.inject()* method.

class pydio.base.**IInjector**

Definition of injector interface.

abstract close() → Optional[Awaitable[None]]

Close this injector.

Closing injector invalidates injector and makes it unusable.

It also cleans up internal cache by calling *IFactory.close()* for each factory being in use by this injector. If this injector has children injectors (created by calling *scoped()* method) then those are closed as well (recursively).

abstract inject(*key: Hashable*) → Union[T, Awaitable[T]]

Create and return object for given hashable key.

On success, this method returns created object or awaitable pointing to created object. On failure, it raises *pydio.exc.InjectorError*.

Parameters

key – Identifier of underlying object factory to be used.

This can be either class object (f.e. base class or interface), a hashable (f.e. string or a number), or a special key wrapper from *pydio.keys*.

Please be aware that same key has to be used in provider during registration of object factory.

abstract scoped(*scope: Hashable*, *env: Optional[Hashable] = None*) → *IInjector*

Create scoped injector that is a child of current one.

Scoped injectors can only operate on *pydio.base.IUnboundFactory* objects with *pydio.base.IUnboundFactory.scope* attribute being equal to given scope.

Parameters

- **scope** – User-defined scope name.
- **env** – User-defined environment name for newly created injector and all its descendants.

  This option is applicable only if none of the ancestors of newly created injector has environment set. Otherwise, setting this will cause `ValueError` exception.

class pydio.base.**IUnboundFactory**

Interface for unbound factories.

Unbound factories are created and managed by *IUnboundFactoryRegistry* objects. The role of this class is to wrap user-specified factory functions that are being registered to providers.

abstract bind(*injector:* IInjector) → *IFactory*

Create *IFactory* object to be owned by given injector.

Parameters
injector – The owner of bound factory object to be created

abstract is_awaitable() → bool

Return True if this factory produces awaitable *IFactory* instances or False otherwise.

abstract property scope: Optional[Hashable]

Name of the scope assigned to this factory.

Factories with scopes defined can only be used by injectors with same scope set.

class pydio.base.**IUnboundFactoryRegistry**

Interface for *IUnboundFactory* objects registry.

Factory registries are used by *IInjector* objects to find *IUnboundFactory* object that matches key that was given to *IInjector.inject()* call.

abstract get(*key: Hashable*, *env: Optional[Hashable] = None*) → Optional[*IUnboundFactory*]

Get *IUnboundFactory* registered for given key and environment (if given).

If no factory was found, then return None.

Parameters

- **key** – See *IInjector.inject()*
- **env** – Environment name.

  Same **key** can be reused by multiple environments, but none can have that key duplicated. This is used to provide several different implementations of one key that depend on environment on which application is executed (f.e. different database may be needed in testing, and different in production)

abstract has_awaitables() → bool

Return True if this factory registry contains awaitable factories or False otherwise.

Behind the scenes, this will check if there is at least one unbound factory for which *IUnboundFactory.is_awaitable()* returns True.

3.3.3 pydio.exc - Base exceptions

Base exception classes for PyDio.

exception pydio.exc.**Base**(***kwargs*)

Bases: Exception, ABC

Common base class for all PyDio exceptions.

You can use this class to catch all exceptions that this library may raise.

abstract property message_template: *str*

Specify message template.

This must be defined in subclass and provides template to render exception message. This template can use **self** to access exception data, for example:

```
class MyException(Base):
    message_template = 'Failed: foo={self.foo!r}'
```

property params: *dict*

Dictionary containing all keyword args given in constructor.

In subclass, this can be used as source of data when adding another properties.

exception pydio.exc.**InjectorError**(***kwargs*)

Bases: *Base*

Base class for exceptions that can be raised by *pydio.base.IInjector* instances.

exception pydio.exc.**ProviderError**(***kwargs*)

Bases: *Base*

Base class for exceptions that can be raised by *pydio.base.IUnboundFactoryRegistry* instances.

3.3.4 pydio.injector - Dependency injector

class pydio.injector.**Injector**(*provider:* IUnboundFactoryRegistry, *env: Optional[Hashable] = None*)

Bases: *IInjector*

Dependency injector main class.

Parameters

- **provider** – Unbound factory provider to work on
- **env** – Name of the environment this injector will use when making queries to `IUnboundFactoryRegistry` object given via **provider**.

  This can be obtained f.e. from environment variable. Once injector is created you will not be able to change this.

  See `IUnboundFactoryRegistry.get()` for more details.

exception AlreadyClosedError(***kwargs*)

Bases: *InjectorError*

Raised when operation on a closed injector was performed.

exception NoProviderFoundError(*key*, *env*)

Bases: *InjectorError*

Raised when there was no matching provider found for given key.

Parameters

- **key** – Searched key
- **env** – Searched environment

exception OutOfScopeError(*key*, *scope*, *required_scope*)

Bases: *InjectorError*

Raised when there was attempt to create object that was registered for different scope.

Parameters

- **key** – Searched key
- **scope** – Injector's own scope
- **required_scope** – Required scope

close() → Optional[Awaitable[None]]

See *pydio.base.IInjector.close()*.

property env: Optional[Hashable]

Environment assigned to this injector.

inject(*key*)

See IInjector.inject.

is_closed() → bool

Return True if this injector was closed or False otherwise.

scoped(*scope: Hashable*, *env: Optional[Hashable] = None*) → *IInjector*

See *pydio.base.IInjector.scoped()*.

3.3.5 pydio.keys - Key wrappers for special purposes

class pydio.keys.**Variant**(*key: Hashable*, ***kwargs: Hashable*)

Bases: Hashable

A special form of key that can have user-defined parameters attached.

This class can be used if you need to use same key twice, but return different objects depending on additional parameters given (which can be accessed by object factory)

Parameters

- **key** – The key to be wrapped
- **kwargs** – Additional parameters to be bound with given key

property key

Wrapped key.

property kwargs: dict

Dict with parameters given in constructor.

3.3.6 pydio.provider - Object factory provider

class pydio.provider.**Provider**

Bases: *IUnboundFactoryRegistry*

Used to record user-defined object factories or instances and bind them with particular key, that can later be used by `IInjector.inject()`.

exception DoubleRegistrationError(*key*, *env*)

Bases: *ProviderError*

Raised when same `(key, env)` tuple was used twice during registration.

Parameters

- **key** – Registered key
- **env** – Registered environment

attach(*provider:* Provider)

Attach given provider to this provider.

This effectively extends current provider with object factories registered to the other one.

Use this if you need to split your providers across multiple modules.

get(*key*, *env=None*)

See `IUnboundFactoryRegistry.get()`.

has_awaitables()

See `IUnboundFactoryRegistry.has_awaitables()`.

provides(*key*, *scope=None*, *env=None*)

Same as *register_func()*, but to be used as a decorator.

Here's an example:

```
from pydio.api import Provider

provider = Provider()

@provider.provides('spam')
def make_spam():
    return 'give me more spam'
```

register_func(*key*, *func*, *scope=None*, *env=None*)

Register user factory function.

Parameters

- **key** – Key to be used for **func**.

  See `IInjector.inject()` for more info.

- **func** – User-defined function to be registered.

  This can be normal function, coroutine, generator or async denerator.

- **scope** – Optional scope to be assigned.
- **env** – Optional environment to be assigned

register_instance(*key*, *value*, *scope=None*, *env=None*)

Same as *register_func()*, but for registration of constant objects.

If your application has some global configuration data you want to inject using PyDio - that's the method you should use.

3.4 Changelog

3.5 License

PYTHON MODULE INDEX

p

INDEX

R

S

V