

---

# Punq Documentation

**Bob Gregory**

**Mar 08, 2022**



---

## Contents

---

<b>1</b>	<b>Punq</b>	<b>1</b>
1.1	Installation . . . . .	1
1.2	Quick Start . . . . .	1
1.3	API . . . . .	3
	<b>Index</b>	<b>7</b>



# CHAPTER 1

---

## Punq

---

An unintrusive library for dependency injection in modern Python. Inspired by [Funq](#), Punq is a dependency injection library you can understand.

- No global state
- No decorators
- No weird syntax applied to arguments
- Small and simple code base with 100% test coverage and developer-friendly comments.

## 1.1 Installation

Punq is available on the [cheese shop](#).

```
pip install punq
```

Documentation is available on [Read the docs](#).

## 1.2 Quick Start

Punq avoids global state, so you must explicitly create a container in the entrypoint of your application:

```
import punq

container = punq.Container()
```

Once you have a container, you can register your application's dependencies. In the simplest case, we can register any arbitrary object with some key:

```
container.register("connection_string", instance="postgresql://...")
```

We can then request that object back from the container:

```
conn_str = container.resolve("connection_string")
```

Usually, though, we want to register some object that implements a useful service.:

```
class ConfigReader:
    def get_config(self):
        pass

class EnvironmentConfigReader(ConfigReader):
    def get_config(self):
        return {
            "logging": {
                "level": os.getenv("LOGGING_LEVEL", "debug")
            },
            "greeting": os.getenv("GREETING", "Hello world")
        }

container.register(ConfigReader, EnvironmentConfigReader)
```

Now we can *resolve* the *ConfigReader* service, and receive a concrete implementation:

```
config = container.resolve(ConfigReader).get_config()
```

If our application's dependencies have their *own* dependencies, Punq will inject those, too:

```
class Greeter:
    def greet(self):
        pass

class ConsoleGreeter(Greeter):
    def __init__(self, config_reader: ConfigReader):
        self.config = config_reader.get_config()

    def greet(self):
        print(self.config['greeting'])

container.register(Greeter, ConsoleGreeter)
container.resolve(Greeter).greet()
```

If you just want to resolve an object without having any base class, that's okay:

```
class Greeter:
    def __init__(self, config_reader: ConfigReader):
        self.config = config_reader.get_config()

    def greet(self):
        print(self.config['greeting'])

container.register(Greeter)
container.resolve(Greeter).greet()
```

And if you need to have a singleton object for some reason, we can tell punq to register a specific instance of an object:

```
class FileWritingGreeter:
    def __init__(self, path, greeting):
        self.path = path
        self.message = greeting
        self.file = open(self.path, 'w')

    def greet(self):
        self.file.write(self.message)

one_true_greeter = FileWritingGreeter("/tmp/greetings", "Hello world")
container.register(Greeter, instance=one_true_greeter)
```

You might not know all of your arguments at registration time, but you can provide them later:

```
container.register(Greeter, FileWritingGreeter)
greeter = container.resolve(Greeter, path="/tmp/foo", greeting="Hello world")
```

Conversely, you might want to provide arguments at registration time, without adding them to the container:

```
container.register(Greeter, FileWritingGreeter, path="/tmp/foo", greeting="Hello world"
→)
```

Fuller documentation is available on [Read the docs](#).

Github workflows, nox configuration, and linting gratefully stolen from [Hypermodern Python](#)

## 1.3 API

### **class** punq.Container

Provides dependency registration and resolution.

This is the main entrypoint of the Punq library. In normal scenarios users will only need to interact with this class.

**register** (service, factory=<punq.Empty object>, instance=<punq.Empty object>, scope=<Scope.transient: 0>, \*\*kwargs)  
Register a dependency into the container.

Each registration in Punq has a “service”, which is the key used for resolving dependencies, and either an “instance” that implements the service or a “factory” that understands how to create an instance on demand.

### Examples

If we have an object that is expensive to construct, or that wraps a resource that must not be shared, we might choose to use a singleton instance.

```
>>> import sqlalchemy
>>> from punq import Container
>>> container = Container()
```

```
>>> class DataAccessLayer:
...     pass
```

(continues on next page)

(continued from previous page)

```

...
>>> class SqlAlchemyDataAccessLayer(DataAccessLayer):
...     def __init__(self, engine: sqlalchemy.engine.Engine):
...         pass
...
>>> dal = SqlAlchemyDataAccessLayer(sqlalchemy.create_engine("sqlite:///"))
>>> container.register(
...     DataAccessLayer,
...     instance=dal
... )
<punq.Container object at 0x...>
>>> assert container.resolve(DataAccessLayer) is dal

```

If we need to register a dependency, but we don't need to abstract it, we can register it as concrete.

```

>>> class FileReader:
...     def read(self):
...         # Assorted legerdemain and rigmarole
...         pass
...
>>> container.register(FileReader)
<punq.Container object at 0x...>
>>> assert type(container.resolve(FileReader)) == FileReader

```

In this example, the EmailSender type is an abstract class and SmtplibEmailSender is our concrete implementation.

```

>>> class EmailSender:
...     def send(self, msg):
...         pass
...
>>> class SmtplibEmailSender(EmailSender):
...     def send(self, msg):
...         print("Sending message via smtp")
...
>>> container.register(EmailSender, SmtplibEmailSender)
<punq.Container object at 0x...>
>>> instance = container.resolve(EmailSender)
>>> instance.send("beep")
Sending message via smtp

```

**resolve\_all** (service, *\*\*kwargs*)

Return all registrations for a given service.

Some patterns require us to use multiple implementations of an interface at the same time.

## Examples

In this example, we want to use multiple Authenticator instances to check a request.

```

>>> class Authenticator:
...     def matches(self, req):
...         return False
...

```

(continues on next page)



(continued from previous page)

```

...     def authenticate(self, req):
...         return False
...
>>> class BasicAuthenticator(Authenticator):
...     def matches(self, req):
...         head = req.headers.get("Authorization", "")
...         return head.startswith("Basic ")
...
>>> class TokenAuthenticator(Authenticator):
...     def matches(self, req):
...         head = req.headers.get("Authorization", "")
...         return head.startswith("Bearer ")
...
>>> def authenticate_request(container, req):
...     for authn in req.resolve_all(Authenticator):
...         if authn.matches(req):
...             return authn.authenticate(req)

```

**resolve** (*service\_key*, *\*\*kwargs*)

Build an return an instance of a registered service.

**instantiate** (*service\_key*, *\*\*kwargs*)

Instantiate an unregistered service.

**exception** `punq.MissingDependencyError`

Raised when a service, or one of its dependencies, is not registered.

## Examples

```

>>> import punq
>>> container = punq.Container()
>>> container.resolve("foo")
Traceback (most recent call last):
punq.MissingDependencyError: Failed to resolve implementation for foo

```

**exception** `punq.InvalidRegistrationError`

Raised when a registration would result in an unresolvable service.

**exception** `punq.InvalidForwardReferenceError`

Raised when a registered service has a forward reference that can't be resolved.

## Examples

In this example, we register a service with a string as a type annotation. When we try to inspect the constructor for the service we fail with an `InvalidForwardReferenceError`

```

>>> from dataclasses import dataclass
>>> from punq import Container
>>> @dataclass
... class Client:
...     dep: 'Dependency'
>>> container = Container()
>>> container.register(Client)
Traceback (most recent call last):

```

(continues on next page)

(continued from previous page)

```
...  
punq.InvalidForwardReferenceError: name 'Dependency' is not defined
```

This error can be resolved by first registering a type with the name ‘Dependency’ in the container.

```
>>> class Dependency:  
...     pass  
...  
>>> container.register(Dependency)  
<punq.Container object at 0x...>  
>>> container.register(Client)  
<punq.Container object at 0x...>  
>>> container.resolve(Client)  
Client(dep=<punq.Dependency object at 0x...>)
```

Alternatively, we can register a type using the literal key ‘Dependency’.

```
>>> class AlternativeDependency:  
...     pass  
...  
>>> container = Container()  
>>> container.register('Dependency', AlternativeDependency)  
<punq.Container object at 0x...>  
>>> container.register(Client)  
<punq.Container object at 0x...>  
>>> container.resolve(Client)  
Client(dep=<punq.AlternativeDependency object at 0x...>)
```

## C

`Container` (*class in punq*), 3

## I

`instantiate()` (*punq.Container method*), 5

`InvalidForwardReferenceError`, 5

`InvalidRegistrationError`, 5

## M

`MissingDependencyError`, 5

## R

`register()` (*punq.Container method*), 3

`resolve()` (*punq.Container method*), 5

`resolve_all()` (*punq.Container method*), 4