component-injector Documentation

Release 1.1.0

Ingmar Steen <iksteen@gmail.com>

Feb 18, 2019

Contents:

This library provides a framework-agnostic component (or dependency) injector that injects registered components into your function calls. The components to insert are identified by looking at the called function argument annotations.

When registering a component, all its base classes are registered as well unless you explicitly disable that. You can also choose to only register base classes that are not already registered with the injector.

It provides local scopes where you can register additional components or override existing components. After exiting the scope, the registry will return to the state it was in before entering the scope.

CHAPTER 1

Components

Adding components, which are instantiated classes, to the injector and injecting them in your function arguments are the most basic functionality the injector.

A small demonstration:

```
from component_injector import Injector

# Define a component to inject.
class MyFirstComponent:
   def __init__(self):
      print("Initializing MyFirstComponent.")

# Create an injector namespace.
injector = Injector()

# Instantiate the component and register it with the injector.
component = MyFirstComponent()
injector.register(component)

# Define a function that uses the component and connect it to
# the injector.
@injector.inject
def my_component_consumer(component: MyFirstComponent):
   print(component)

# Calling the consumer without specifying the `component`
# argument will trigger the injector to add it automatically.
my_component_consumer()
```

CHAPTER 2

Factories

If constructing your component is expensive and you only want to instantiate it if it's really necessary you can add a factory function to the injector which will only be called when the component is needed.

When registering a factory function, make sure the return type annotation matches the type of the component you want to inject.

Demonstration:

```
import time
from component_injector import Injector

injector = Injector()

class CheapComponent:
   pass

class ExpensiveComponent:
   pass

def expensive_factory() -> ExpensiveComponent:
   time.sleep(1)
   return ExpensiveComponent()

injector.register(CheapComponent())
injector.register_factory(expensive_factory)

@injector.inject
def consumer_1(c1: CheapComponent):
   pass

@injector.inject
def consumer_2(c1: CheapComponent, c2: ExpensiveComponent):
   pass

# This will not create the expensive components.
```

(continues on next page)

(continued from previous page)

```
consumer_1()

# ExpensiveComponent will not be instantiated until needed.
consumer_2()

# Needing it again will use the same instance created before.
consumer_2()
```

2.1 Asynchronous factories

If you register a factory that returns an *Awaitable*, you can use it to inject the resolved component into a coroutine:

```
import asyncio
from component_injector import Injector

class Component:
    pass

async def factory() -> Component:
    await asyncio.sleep(1)
    return Component()

injector = Injector()
injector.register_factory(factory)

@injector.inject
async def consumer(c: Component):
    pass

loop = asyncio.get_event_loop()
loop.run_until_complete(consumer())
```

CHAPTER 3

Scopes

A scope can be used to provide components (or factories) that are only valid in a certain context. f.e. If you use the injector with a web framework, you can add request-specific components to the injector in a separate scope as to not pollute the global scope.

Note that scopes are based on python's *contextvars*. This means they are thread safe when using the backported package for python 3.6. On python 3.7 and newer, they are also safe to use with *asyncio* tasks as well.

3.1 Basic usage

Basic example:

```
from component_injector import Injector

# The configuration is a global component and will be added and
# available from the root scope.
class Config:
   loglevel = "DEBUG"

# This class describes the current request and is request-specific.
class Request:
   def __init__(self, method, path):
      self.method = method
      self.path = path

injector = Injector()

@injector.inject
def handle_request(config: Config, request: Request):
   if config.loglevel == "DEBUG":
      print(request.method, request.path)

# Register our global configuration component.
```

(continues on next page)

(continued from previous page)

```
injector.register(Config())

# When receiving a request, set up a new scope:
with injector.scope():
    injector.register(Request("GET", "/index.html"))
    handle_request()

# This will fail, after leaving the scope the request was removed
# from the injector.
handle_request()
```

3.2 Persistent factories

By default, the components created by factories are added to the current scope and removed when exiting the scope. It is however possible to instruct the injector to store the component in the same scope as the factory. You can do this by setting the *persistent* flag to *True* when adding the factory:

```
from component_injector import Injector

class Component:
    pass

injector = Injector()

@injector.inject
def consume(c: Component):
    return c

injector.register_factory(Component)

# Set up a new scope:
with injector.scope():
    # Call the consumer, triggering the factory.
    c1 = consume()
    # Ensure the same component is injected again.
    assert c1 is consume()

# After exiting the scope, the component will be cleaned up. Calling
# the consumer again will trigger the factory once more.
with injector.scope():
    assert c1 is not consume()

# Now, let's re-add the factory but this time make it persistent.
injector.register_factory(Component, persistent=True)

# Set up a new scope and call the consumer. This will create the
# component and insert it into the root scope as that is where the
# factory is located.
with injector.scope():
    c1 = consume()

# Even after leavig the scope the component was created in, the
# component persists because the factory is part of the root scope.
assert c1 is consume()
```

3.3 Re-entering scopes

If needed, you can re-enter a specific scope as well:

```
from component_injector import Injector

class Component:
   def __init__(self, msg):
      self.msg = msg

injector = Injector()

@injector.inject
def consumer(c: Component):
   return c.msg

with injector.scope() as ctx:
   injector.register(Component("Initial scope"))

   assert consumer() == "Initial scope"

   with injector.scope():
      injector.register(Component("Secondary scope"))

      assert consumer() == "Secondary scope"

      # Re-enter initial scope
      with ctx:
          assert consumer() == "Initial scope"

      # We're now back in the secondary scope.
      assert consumer() == "Secondary scope"
```

CHAPTER 4

component_injector

class component_injector.**Injector**

Provides a basic injector namespace. It's common to use one injector per project.

get_component(*type_: Type[T]*) → T

Get a component from the injector's current scope. Materialize it using a factory if necessary.

Note that it is an error to use this function to get a component that has a factory that returns an *Awaitable*.

Parameters **type** – The type of the component to return.

Returns The materialized component.

get_component_async(*type_: Type[T]*) → T

Get a component from the injector's current scope. Materialize it using a factory if necessary.

Use this method if the component's factory function returns an *Awaitable*.

Parameters **type** – The type of the component to return.

Returns The materialized component.

inject(*f: Callable[[...], T]*) → Callable[[...], T]

This decorator will connect the injector to a function or method. When the resulting function is called, the provided arguments will be checked against the function's signature and any missing arguments the injector has a component or factory those arguments will be filled in.

Parameters **f** – The function or method to inject components into.

Returns The decorated function.

register(*component: Any*, *, *bases: bool = True*, *overwrite_bases: bool = True*) → None

Register a new component with the injector.

Parameters

- **component** – The component to register with the injector.
- **bases** – Besides registering the exact component type, also register for all of the component's base classes. Defaults to *True*.

- **overwrite_bases** – If any of the component's base classes are already registered with the injector, overwrite those registrations. Defaults to *True*.

register_factory(*factory: Callable[[], Any], *, bases: bool = True, overwrite_bases: bool = True, persistent: bool = False*) → None

Register a new factory function with the injector. Not that the factory function's return type annotation should be set to the type of the component you want to inject.

Parameters

- **factory** – The factory function. Will be called without arguments and should return the instantiated component. If the factory returns an Awaitable it can only used to inject into coroutine functions.
- **bases** – Besides registering the exact component type, also register for all of the component's base classes. Defaults to *True*.
- **overwrite_bases** – If any of the component's base classes are already registered with the injector, overwrite those registrations. Defaults to *True*.
- **persistent** – When materializing the component using the factory, insert the component into the scope where the factory is registered instead of the current scope. Defaults to *False*.

scope() → component_injector.Context

Return a context manager that you can use to enter a new scpoe. When leaving the scope, any components or factories added to the injector will be forgotten.

Returns The scope context object. You can use this to re-enter this scope at a later time if needed.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

Index