SunnyDl Documentation

Release 0.0.0

Justin Smith

May 23, 2017

Contents

SunnyDI is an IoC container for managing and injecting dependencies in Python.

It is inspired by Autofac for .NET and Guice for java.

CHAPTER 1

How to Use

For our example, we will create an IoC module for our HelloService.:

```
class HelloService(object):
    def hello(self):
        return 'hello'
```

Create a new configuration module that extends *sunnydi.ioc.Module*. A module defines how objects will be created, destroyed and provided to other object instances in the IoC object graph. In the most simple configuration, we can just bind a string name to our HelloService class type:

```
class HelloModule(Module):
    def configure(self):
        self.bind('hello_service')
            .to(HelloService)
```

We can then create the injector and resolve our HelloService like this:

```
>>> hello_module = HelloModule()
>>> injector = hello_module.create_injector()
>>> hello_service = injector.get('hello_service')

>>> hello_service.hello()
'hello'
```

Resolved instances are provided via constructor arguments to consuming classes. For instance, given the following class:

```
class MyClass(object):

    def __init__(self, hello_service):
        self._hello_service = hello_service

    def do_hello(self):
        return self._hello_service.hello()
```

An instance of *MyClass* can be resolved with an instance of *HelloService* due to the service's binding name matching the parameter defined in the *MyClass* constructor:

```
>>> my_class_instance = injector.get(MyClass)
>>> my_class_instance.do_hello()
'hello'
```

For advanced usage, checkout the docs

Installation

Binaries are available via pip. Source code is available on github.

Pip Install

From the command line:

```
$ pip install sunnydi
```

Get the Source Code

Clone the public repository from github:

```
$ git clone git@github.com:thomasstreet/sunnydi.git
```

Advanced Usage

Advanced features and load balancer configuration options.

Configuring the LoadBalancer

The `LoadBalancer` takes a number of configuration options in order to suit different environments. At its most basic, it just requires a `ServerList` implementation:

```
import ballast
from ballast.discovery.static import StaticServerList

servers = StaticServerList(['127.0.0.1', '127.0.0.2'])
load_balancer = ballast.LoadBalancer(servers)
```

Now, you can configure a `Service` with the load balancer:

```
my_service = ballast.Service(load_balancer)
```

Make an HTTP request as you would with *requests* - using a relative path instead of an absolute URL:

```
response = my_lb_service.get('/v1/path/to/resource')
# <Response[200]>
```

The following options are enabled by default when no other options are specified:

```
from ballast import rule, ping

ballast.LoadBalancer(
    servers, # required
    rule=rule.RoundRobinRule(),
    ping_strategy=ping.SerialPingStrategy(),
    ping=ping.SocketPing()
)
```

Dynamic Server Discovery

Servers can be discovered dynamically by configuring one of the dynamic `ServerList` implementations (or creating your own) on the `LoadBalancer`. The `ServerList` is periodically queried by the `LoadBalancer` for updated `Server` objects.

DNS

NOTE: Using DNS features requires additional dependencies. From the command line, install the DNS dependencies from pip:

```
$ pip install ballast[dns]
```

To use DNS to query `Server` instances, configure a `LoadBalancer` with either a `DnsARecordList` to query *A* records

```
import ballast
from ballast.discovery.ns import DnsARecordList

servers = DnsARecordList('my.service.internal.')
load_balancer = ballast.LoadBalancer(servers)
```

Or use `DnsServiceRecordList` to query *SRV* records

```
import ballast
from ballast.discovery.ns import DnsServiceRecordList

servers = DnsServiceRecordList('my.service.internal.')
load_balancer = ballast.LoadBalancer(servers)
```

Consul REST API

To use Consul (via HTTP REST API) to query `Server` instances, configure a `LoadBalancer` with `ConsulRestRecordList`

```
import ballast
from ballast.discovery.consul import ConsulRestRecordList

servers = ConsulRestRecordList('http://my.consul.url:8500', 'my-service')
load_balancer = ballast.LoadBalancer(servers)
```

Load-Balancing Rules

The logic of how to choose the next server in the load-balancing pool is configurable by specifying a `Rule` implementation.

RoundRobinRule

The `RoundRobinRule` chooses each server in the load-balancing pool an equal number of times by simply looping through the collection of servers in the pool:

```
import ballast
from ballast import rule

servers = ... # defined earlier

my_rule = rule.RoundRobinRule()
load_balancer = ballast.LoadBalancer(servers, my_rule)
```

PriorityWeightedRule

The `PriorityWeightedRule` chooses each server in the load-balancing pool based on a combination of *priority* and *weight*.

Given a pool of 5 servers with the following priority/weight values, this rule will choose priority *1* servers exclusively (unless/until all priority *1* servers are down, in which case it will move on to priority *2* servers):

```
# priority 1
Server(address='127.0.0.1', priority=1, weight=60)
Server(address='127.0.0.2', priority=1, weight=20)
Server(address='127.0.0.3', priority=1, weight=20)

# priority 2 (backups)
Server(address='127.0.0.4', priority=2, weight=1)
Server(address='127.0.0.5', priority=2, weight=1)
```

Of the current priority *1* servers, the choice of server will be determined by its *weight* as a ratio. 60% of the traffic will go to *127.0.0.1* while the remaining 40% will be split evently between *127.0.0.2* and *127.0.0.3* (both have the same weight):

```
Server(address='127.0.0.1', priority=1, weight=60)
```

If all priority *1* servers are down, this rule will split traffic between *127.0.0.4* and *127.0.0.5* equally (both have the same weight).

For this rule to work correctly, it must be paired with a `ServerList` that provides *priority* and *weight* as part of its discovery (e.g. `DnsServiceRecordList`):

```
import ballast
from ballast import rule
from ballast.discovery.ns import DnsServiceRecordList

# use a ServerList that provides 'priority' and 'weight'
servers = DnsServiceRecordList('my.service.internal.')

my_rule = rule.PriorityWeightedRule()
load_balancer = ballast.LoadBalancer(servers, my_rule)
```

Pinging Servers

The `LoadBalancer` periodically queries for servers as well as attempts to *ping* each server to ensure it's up, running and responding. This can be configured via the following standard `Ping` implementations (or you can create your own):

DummyPing

`DummyPing` doesn't actually ping any servers, it just assumes the server is active - useful for testing or when otherwise not wanting to actually ping servers in the load balancing pool. Not recommended for production.

SocketPing

`SocketPing` attempts to open a socket connection to the server. If the connection was successful, the ping is considered successful.

UrlPing

`UrlPing` attempts to make a *GET* request to the server. If the request returns a *2xx* status code, the ping is considered successful.

Ping Strategies

The `LoadBalancer` initiates its periodic ping using a configurable `PingStrategy`. The following strategies are available (or you can create your own):

SerialPingStrategy

The `SerialPingStrategy` iterates through each `Server` attempting to ping each one sequentially. The time it takes for this strategy to complete is *ping time x number of servers*. It is recommended to use this strategy only when there are a (known) small number of servers.

ThreadPoolPingStrategy

The `ThreadPoolPingStrategy` iterates through each `Server` attempting to ping each server in parallel using a `ThreadPool`. The time it takes for this strategy to complete is not much longer than the time it takes for a single ping to complete.

NOTE: this class does not play well when using gevent. It's recommended to use the `GeventPingStrategy` instead for gevent-based systems.

MultiprocessingPoolPingStrategy

The `MultiprocessingPoolPingStrategy` iterates through each `Server` attempting to ping each server in parallel using a `Pool`. The time it takes for this strategy to complete is not much longer than the time it takes for a single ping to complete, however, on systems where a large number of servers are queried, it's recommended to use `ThreadPoolPingStrategy` instead.

NOTE: this class does not play well when using gevent. It's recommended to use the `GeventPingStrategy` instead for gevent-based systems.

SunnyDI api reference

Inversion of Control

Framework for configuring and composing object graphs injecting their associated dependencies.

Using inversion-of-control rather than manually building object graphs can reduce an application's maintenance burden.

For the philosophical reasoning behind such an architecture, see Martin Fowler's [article](http://martinfowler.com/articles/injection.html).

Getting Started

In order to create an injector, we must first create and configure a *sunnydi.ioc.Module*. A module defines how instances will be built and provided to other instances in the object graph.

For our example, we will create a module for our HelloService.

#!python class HelloService(object):

def hello(self): return 'hello'

Now, we create a custom configuration module that extends *sunnydi.ioc.Module*. In the most simple configuration, we just bind a contract name to our HelloService class type:

#!python class HelloModule(Module):

def configure(self):

self.bind('hello_service') .to(HelloService)

We can then create an injector and resolve our HelloService like this:

#!python hello_module = HelloModule() injector = hello_module.create_injector() hello_service = injector.get('hello_service')

```
>>> hello_service.hello()
'hello'
```

Advanced Configuration

More often than not, classes will have dependencies on other classes, and those classes will have additional dependencies. This results in potentially large object graphs that becomes very difficult to manage manually. On top of that, we probably only need to create some classes once for the lifetime of the application.

The below configuration illustrates how to accomplish this with the IoC configuration Module:

#!python class GoodbyeService(object):

param name matches our binding contract name def __init__(self, hello_service):

self._hello_service = hello_service

def goodbye(self): return '%s, goodbye' % self._hello_service.hello()

class HelloModule(Module):

def configure(self):

we only ever need one instance of this service self.bind('hello_service')

.to(HelloService) .as_singleton()

we only ever need one instance of this service self.bind('goodbye_service')

.to(GoodbyeService) .as_singleton()

...

hello_module = HelloModule() injector = hello_module.create_injector()

resolving the service multiple times # returns the same instance due to *as_singleton()* goodbye_service = injector.get('goodbye_service') goodbye_service2 = injector.get('goodbye_service')

```
>>> assert goodbye_service == goodbye_service2
True
```

```
>>> goodbye_service.goodbye()
'hello, goodbye'
```

Occasionally, manual configuration of a class is necessary in whole or in part. In these cases, the module can configure a factory method to provide the instance, or provide an instance as-is.

#!python class HelloModule(Module):

def configure(self):

new up an instance on our own # this instance is de facto singleton hello_service = HelloService() # additional configuration ... self.bind('hello_service')

.to_instance(hello_service)

this service uses a factory to create the instance # factory can be static, instance, or global function # factory can also be marked as singleton self.bind('goodbye_service')

.to_factory(self._create_goodbye_service) .as_singleton()

@staticmethod def _create_goodbye_service(hello_service):

goodbye_service = GoodbyeService(hello_service) # additional configuration ... return goodbye_service

Resolving Instances

Class instances can be resolved directly from the injector via their contract name(s) or class type(s). Multiple contracts may be resolved by adding additional parameters to the *get()* call.

#!python

get one goodbye_service = injector.get('goodbye_service')

get many (hello_service, goodbye_service) = injector.get('hello_service', 'goodbye_service')

get can also take a class type goodbye_service = injector.get(GoodbyeService)

For CLI applications, resolving the main application class should be the only call to *get()* necessary (the remaining object graph should be populated via the injector).

For non-CLI or other applications in which object lifecycle isn't fully controlled, the *sunnydi.ioc.inject* decorator may be used on a class's *__init__()* method (*MyClass* does _not_ need to be configured in the module).

The *sunnydi.ioc.inject* decorator is not necessary for classes resolved via the injector (only for classes outside the injection context).

#!python class MyClass(object):

@inject def __init__(self, hello_service, goodbye_service):

self._hello_service = hello_service self._goodbye_service = goodbye_service

same as calling my_class = injector.get(MyClass)

Global Injector

In some cases, there may be a need to import and use the *sunnydi.ioc.Injector* from the global context.

#!python from sunnydi.ioc import get_injector

The *sunnydi.ioc.Injector* can also be resolved from the *sunnydi.ioc.inject* decorator:

#!python @inject def __init__(self, injector):

pass

same as calling injector = injector.get('injector')

In order to use the global *sunnydi.ioc.get_injector* or the *sunnydi.ioc.inject* decorator, we must register the configuration module:

#!python hello_module = HelloModule() injector = hello_module.create_injector() hello_module.register(injector)

or (if we don't need the *sunnydi.ioc.Injector* instance right away):

#!python hello_module = HelloModule() hello_module.register()

exception `sunnydi.ioc.`**`DependencyResolutionException`**
Raised when an item could not be resolved from an *sunnydi.ioc.Injector*.

exception `sunnydi.ioc.`**`ComponentNotRegisteredException`**
Raised when a component binding was not registered with an *sunnydi.ioc.Injector*.

exception `sunnydi.ioc.`**`ScopeDisposedException`**
Raised when an *sunnydi.ioc.InjectionScope* has been disposed, but a client has attempted to resolve a component from it.

class `sunnydi.ioc.`**`Module`**(*name=None*)
Configuration module for defining dependency-injection bindings.

`add_module`(*module*)
Add a configuration sub-module to this module.

•module (*sunnydi.ioc.Module*): A sub-module to add.

`bind`(*contract*)
Create a new binding with the specified contract name.

•contract (basestring): The contract name to bind to.

A binding builder.

`configure`()
Configure the IoC module, creating any necessary injection bindings.

`create_injector`()
Create the dependency *sunnydi.ioc.Injector* using the configured bindings.

A configured *sunnydi.ioc.Injector*.

name
The module name (or *DEFAULT_INJECTOR*).

register(*injector=None*)
Register the specified *sunnydi.ioc.Injector* with the global scope. If *injector* parameter is not specified, create a new *sunnydi.ioc.Injector* and register it.

- **injector (*sunnydi.ioc.Injector*):** The injector to register or create and register a new *sunnydi.ioc.Injector* if None.

class sunnydi.ioc.**Injector**(*bindings=None*)
Dependency injector used for resolving dependencies.

This class is typically not created explicitly, rather it is created by configuring a *sunnydi.ioc.Module*.

get(**args*, ***kwargs*)
Resolve an instance or instances of the specified contracts.

- contract (basestring): One or more contract names to resolve.
- **class_type (type): One or more classes to resolve with** constructor parameters injected.
- **class_args (tuple): (Optional) Collection of arguments to pass** to a resolving class's positional arguments (**args*) instead of resolving parameters via the injector.
- **class_kwargs (dict): (Optional) Collection of key-word arguments** to pass to a resolving class's keyword arguments (**kwargs*) instead of resolving parameters via the injector.

The resolved object instance or tuple of instances if multiple parameters specified.

- ***sunnydi.ioc.DependencyResolutionException*:** If no contracts or types are specified or if *None* type is specified.
- ***sunnydi.ioc.ComponentNotRegisteredException*:** If a binding for the specified contract could not be found.

is_scope(*scope_id*)
Whether or not a child *sunnydi.ioc.InjectionScope* with the specified scope id exists for this injector.

Will only return *True* for scopes created directly from this injector (not child scopes).

- scope_id (basestring): The unique scope identifier.

True if a scope with the specified id exists, *False* if no scope exists.

scope(*scope_id=None*)
Create a new child *sunnydi.ioc.InjectionScope* with the specified scope id or return a previously created child scope.

It's recommended to use this method as a context manager in order to properly dispose of the scope when it's finished being used.

#!python with injector.scope('my-scope-id') as my_scope:

obj = my_scope.get('my_contract_name')

If not being used as a context manager, it is mandatory to manually dispose of the scope via the *sunnydi.ioc.InjectionScope.dispose()* method when finished using the scope. Failure to do so will result in memory leaks within the application.

#!python my_scope = injector.scope('my-scope-id') obj = my_scope.get('my_contract_name') my_scope.dispose()

- **scope_id (basestring): (Optional) The unique scope identifier.** If no scope id is specified, a random *uuid.UUID* is used to create a new scope id.

An *sunnydi.ioc.InjectionScope*

class sunnydi.ioc.**InjectionScope**(*scope_id*, *bindings*, *parent_scope*)
Dependency injector used for resolving dependencies within a limited scope.

This class is typically not created explicitly, rather it is created from a parent *sunnydi.ioc.Injector* by calling *scope()*.

dispose()

get(**args*, ***kwargs*)
Resolve an instance or instances of the specified contracts within the specified scope.

•contract (basestring): One or more contract names to resolve.

•**class_type (type): One or more classes to resolve with** constructor parameters injected.

•**class_args (tuple): (Optional) Collection of arguments to pass** to a resolving class's positional arguments (**args*) instead of resolving parameters via the injector.

•**class_kwargs (dict): (Optional) Collection of key-word arguments** to pass to a resolving class's keyword arguments (**kwargs*) instead of resolving parameters via the injector.

The resolved object instance or tuple of instances if multiple parameters specified.

•***sunnydi.ioc.DependencyResolutionException*:** If no contracts or types are specified or if *None* type is specified.

•***sunnydi.ioc.ComponentNotRegisteredException*:** If a binding for the specified contract could not be found.

•***sunnydi.ioc.ScopeDisposedException*:** If the scope has been disposed.

scope_id

sunnydi.ioc.**inject**(*f*)
Inject the dependencies for the decorated function.

Each of the function's parameter names should match a configured binding name.

If a parameter is included directly, injection is skipped for that parameter.

sunnydi.ioc.**get**(**args*, ***kwargs*)
Resolve an instance or instances of the specified contracts.

•contract (basestring): One or more contract names to resolve.

•**class_type (type): One or more classes to resolve with** constructor parameters injected.

•**class_args (tuple): (Optional) Collection of arguments to pass** to a resolving class's positional arguments (**args*) instead of resolving parameters via the injector.

•**class_kwargs (dict): (Optional) Collection of key-word arguments** to pass to a resolving class's keyword arguments (**kwargs*) instead of resolving parameters via the injector.

The resolved object instance or tuple of instances if multiple parameters specified.

Raises:

•***sunnydi.ioc.DependencyResolutionException*:** If no contracts or types are specified or if *None* type is specified.

•***sunnydi.ioc.ComponentNotRegisteredException*:** If a binding for the specified contract could not be found.

sunnydi.ioc.**scope**(*scope_id=None*)

Create a new child *sunnydi.ioc.InjectionScope* from the default injector, with the specified scope id or return a previously created child scope.

It's recommended to use this method as a context manager in order to properly dispose of the scope when it's finished being used.

#!python import ioc with ioc.scope('my-scope-id') as my_scope:

obj = my_scope.get('my_contract_name')

If not being used as a context manager, it is mandatory to manually dispose of the scope via the *sunnydi.ioc.InjectionScope.dispose()* method when finished using the scope. Failure to do so will result in memory leaks within the application.

#!python my_scope = ioc.scope('my-scope-id') obj = my_scope.get('my_contract_name') my_scope.dispose()

- **scope_id (basestring): (Optional) The unique scope identifier.** If no scope id is specified, a random *uuid.UUID* is used to create a new scope id.

An *sunnydi.ioc.InjectionScope*

sunnydi.ioc.**get_injector**(*name='Default'*)

Get the *sunnydi.ioc.Injector* registered with the specified name, the default injector if no name is specified, or None if no injector is globally registered.

- name (basestring): The injector name (or *DEFAULT_INJECTOR*).

The named *sunnydi.ioc.Injector* (or the default injector if *name* is not specified)

- genindex
- modindex
- search

Python Module Index

b

s

Index