yaak.inject Documentation

Release 0.2.1

Sylvain Prat

March 23, 2012

CONTENTS

YAAK stands for Yet Another Application Kit. It's a set of tools that help developing enterprise applications in python.

`yaak.inject` is a package from the YAAK toolkit that provides dependency injection to your applications. See this Martin Fowler's article for an explanation of dependency injection and its usefulness when developing enterprise application.

CHAPTER

ONE

INSTALLATION

You should have `easy_install` (from setuptools or something equivalent) installed on your system.

To install the package, just type:

```
$ easy_install yaak.inject
```

You can also install the package from a source tarball. Decompress the source archive and type:

```
$ python setup.py install
```

CHAPTER

TWO

SUPPORT

This project is hosted on bitbucket.org. Please report issues via the bug tracker.

The package documentation can be found here.

Automated tests are run over the mercurial repository regularly. Build results can be found here.

CHAPTER

THREE

GETTING STARTED

The `yaak.inject` module implements dependency injection. Here is a tutorial that explains how to use this module.

First, import the `yaak.inject` module so that you can use the injection functionality in your application:

```
>>> from yaak import inject
```

Create a class whose instances have to be injected a *feature* identified by the string `IService` (but could be any hashable type, such as a class):

```
>>> class Client(object):
...   service = inject.Attr('IService')  # inject a feature as an attribute
...   def use_service(self):
...     self.service.do_something()  # use the injected feature
...
```

Also, create a class (or any callable) that implements the *feature*:

```
>>> class Service(object):
...   def do_something(self):
...     print "Service: I'm working hard"
...
```

Then, when you configure your application, you need to wire an implementation for each *feature*. In this case, we provide an implementation for the `IService` feature:

```
>>> inject.provide('IService', Service)
```

Note that we provide a factory (class) for the feature and not the instance itself. You'll see later why.

Now, a Client instance can use the service:

```
>>> client = Client()
>>> client.use_service()
Service: I'm working hard
```

When you use the default `provide()` behavior, all instances of the Client class will be injected the same Service instance:

```
>>> another_client = Client()
>>> client.service is another_client.service
True
```

In fact, the default behavior when you `provide()` a feature is to create a thread-local singleton that is injected in all instances that request the feature. That's what we call the *scope*: it defines the lifespan of the feature instance.

You may want a different `IService` instance for each Client. You can do that by changing the default scope to `Scope.Transient` when you provide the feature:

```
>>> inject.provide('IService', Service, scope=inject.Scope.Transient)
```

Then, a different Service instance is injected in each new Client instance:

```
>>> client = Client()
>>> another_client = Client()
>>> client.service is another_client.service
False
```

You can also declare injected features as function/method parameters instead of attributes:

```
>>> class Client(object):
...   @inject.Param(service='IService')
...   def __init__(self, text, service):
...     self.text = text
...     self.service = service
...   def use_service(self):
...     print self.text
...     self.service.do_something()
...
```

Then you could use the Client class and get the parameters injected automatically if you don't provide a value for them:

```
>>> client = Client('This is a text')
>>> client.use_service()
This is a text
Service: I'm working hard
```

That's the easiest way to declare injected parameters. But if you want to keep your class decoupled from the injection framework, you can also define the injection afterwards:

```
>>> class Client(object):
...   def __init__(self, text, service):
...     self.text = text
...     self.service = service
...   def use_service(self):
...     print self.text
...     self.service.do_something()
...
>>> inject_service = inject.Param(service='IService')
>>> InjectedClient = inject_service(Client)
>>> client = InjectedClient('This is a text')
>>> client.use_service()
This is a text
Service: I'm working hard
```

CHAPTER

FOUR

DOCUMENTATION

4.1 API Documentation for yaak.inject 0.2.1

4.1.1 Defining the injected features

The Dependency Injection can be performed either by attribute injection or parameter injection:

class yaak.inject.**Attr**(*feature*, *provider=None*)

Descriptor that provides attribute-based dependency injection.

Inject a *feature* as an instance attribute. *feature* can be any hashable identifier. If a *provider* is specified, the feature instance will be retrieved from this provider. Otherwise, the default feature provider will be used.

Example:

```
>>> from yaak import inject
>>> class Client(object):
...   service = inject.Attr('IService')
```

class yaak.inject.**Param**(*provider=None*, ***injections*)

Decorator that provides parameter-based dependency injection.

Inject feature instances into function parameters. First, specify the parameters that should be injected as keyword arguments (e.g. `param=<feature>` where `<feature>` is the feature identifier). Then, each time the function will be called, the parameters will receive feature instances. If a *provider* is specified, the feature instances will be retrieved from this provider. Otherwise, the default feature provider will be used.

Example:

```
>>> from yaak import inject
>>> class Client(object):
...   inject.Param(service='IService')
...   def func(self, service):
...     pass  # use service
```

4.1.2 Providing the features

class yaak.inject.**FeatureProvider**(*scope_manager=None*)

Provides the feature instances for injection when requested. It creates instances when necessary and uses the scope manager to obtain the scoped contexts where we store the feature instances.

Creates a feature provider. If *scope_manager* is specified, feature instances will be stored in the contexts of the *scope_manager*. Otherwise, the default scope manager will be used.

clear()
Unregister all features.

get(*feature*)
Retrieve a (scoped) feature instance. Either find the instance in the associated context or create a new instance using the factory method and store it in the context. Raises a `yaak.inject.MissingFeatureError` when no feature has been provided yet.

provide(*feature*, *factory*, *scope='Thread'*)
Provide a *factory* that build (scoped) instances of the *feature*. By default, the scope of the *feature* instance is `yaak.inject.Scope.Thread`, but you can change this by providing a *scope* parameter.

Note that you can change the factory for a feature by providing the same feature again, but the injected instances that already have a reference on the feature instance will not get a new instance.

4.1.3 Scopes

class yaak.inject.**Scope**
Enumeration of the different scope values. Not all scopes are available in every circumstance.

Application = 'Application'
One instance per application (subject to thread-safety issues)

Request = 'Request'
One instance per HTTP request

Session = 'Session'
One instance per HTTP session

Thread = 'Thread'
One instance per thread: this is the default

Transient = 'Transient'
A new instance is created each time the feature is requested

class yaak.inject.**ScopeManager**
Manages scope contexts where we store the instances to be used for the injection.

Creates a new scope manager.

clear_context(*scope*)
Clears the context for a *scope*, that is, remove all instances from the *scope* context.

enter_scope(*scope*, *context=None*, *context_lock=None*)
Called when we enter a *scope*. You can eventually provide the *context* to be used in this *scope*, that is, a dictionary of the instances to be injected for each feature. This is especially useful for implementing session scopes, when we want to reinstall a previous context. You can also pass a lock to acquire when modifying the context dictionary via the parameter *context_lock* if the scope is subject to thread concurrency issues. Raises a `yaak.inject.ScopeReenterError` when re-entering an already entered *scope*.

exit_scope(*scope*)
Called when we exit the *scope*. Remove the context for this *scope*. Raises a `yaak.inject.UndefinedScopeError` if the *scope* is not defined.

get_or_create(*scope*, *key*, *factory*)
Get the value for a *key* from the *scope* context, or create one using the *factory* provided if there's no value for this *key*. Raises a `yaak.inject.UndefinedScopeError` if the *scope* is not defined.

class yaak.inject.**ScopeContext**(*scope*, *context=None*, *context_lock=None*, *scope_manager=None*)
Context manager that defines the lifespan of a scope.

Creates a scope context for the specified *scope*. If *context* is passed a dictionary, the created instances will be stored in this dictionary. Otherwise, a new dictionary will be created for storing instance each time we enter the scope. So the *context* argument can be used to recall a previous context. If *context_lock* is specified, the lock will be acquired/released when the context dictionary is updated, in order to avoid thread concurrency issues. If *scope_manager* is specified, contexts will be stored in this *scope_manager*. Otherwise, the default scope manager will be used.

class `yaak.inject.`**`WSGIRequestScope`**(*app*, *scope_manager=None*)

WSGI middleware that installs the `yaak.inject.Scope.Request` contexts for the wrapped application.

Installs a `yaak.inject.Scope.Request` context for the application *app*. That is, a new context will be used in each HTTP request for storing the request scoped features. You can eventually pass the *scope_manager* that will handle the scope contexts. Otherwise, the default scope manager will be used.

4.1.4 Using the default feature provider

`yaak.inject.`**`provide`**(*self*, *feature*, *factory*, *scope='Thread'*)

Provides a *factory* for a *feature* to the default feature provider. See `yaak.inject.FeatureProvider.provide()` for more information.

`yaak.inject.`**`get`**(*self*, *feature*)

Gets a *feature* from the default feature provider. See `yaak.inject.FeatureProvider.get()` for more information.

`yaak.inject.`**`clear`**(*self*)

Clears the features from the default feature provider. See `yaak.inject.FeatureProvider.clear()` for more information.

4.1.5 Helper tools

`yaak.inject.`**`bind`**(*func*, ***frozen_args*)

This function is similar to the `functools.partial()` function: it implements partial application. That is, it's a way to transform a function to another function with less arguments, because some of the arguments of the original function will get some fixed values: these arguments are called frozen arguments. But unlike the `functools.partial()` function, the frozen parameters can be anywhere in the signature of the transformed function, they are not required to be the first or last ones. Also, you can pass a `yaak.inject.late_binding()` function as the value of a parameter to get the value from a call to this function when the bound function is called (this implements late binding). Raises a `yaak.inject.BindNotSupportedError` when passing an unsupported function to bind, for example a function with variable arguments.

Say you have a function `add()` defined like this:

```
>>> def add(a, b):
...   return a + b
```

You can bind the parameter *b* to the value 1:

```
>>> add_one = bind(add, b=1)
```

Now, `add_one()` has only one parameters *a* since *b* will always get the value 1. So:

```
>>> add_one(1)
2
>>> add_one(2)
3
```

Now, an example of late binding:

```
>>> import itertools
>>> count = itertools.count(0)
>>> def more_and_more():
...   return count.next()
...
>>> add_more_and_more = bind(add, b=late_binding(more_and_more))
>>> add_more_and_more(1)
1
>>> add_more_and_more(1)
2
>>> add_more_and_more(1)
3
```

yaak.inject.**late_binding**(*func*)
Create a late binding by providing a factory function to be called when the bound function is called

4.1.6 Exceptions

exception yaak.inject.**MissingFeatureError**
Exception raised when no implementation has been provided for a feature.

exception yaak.inject.**ScopeError**
Base class for all scope related errors

exception yaak.inject.**UndefinedScopeError**
Exception raised when using a scope that has not been entered yet.

exception yaak.inject.**ScopeReenterError**
Exception raised when re-entering a scope that has already been entered.

exception yaak.inject.**BindNotSupportedError**
Exception raised when a function could not be used in the yaak.inject.bind() method.

CHAPTER

FIVE

CHANGELOG

5.1 0.2.1 (11-March-2012)

- The setup.py file does not import code anymore in order to retrieve the version information, since it may cause some installation problems
- Fixed bad years in the changelog, and reordered the items so that the most recent changes appear first
- Changed the aliases for releasing new versions
- Fixed line endings (unix style)
- Removed the extensions of the text files since it's a convention in the Python world.

5.2 0.2.0 (24-Oct-2011)

- Fixed the broken lock acquire/release implementation when updating the application context dictionary.
- The locking mechanism is now available for all scopes.
- The context manager is now responsible for updating the context dictionaries.
- Fixed duplicate factory calls when providing a factory returning None
- ScopeManager.enter_scope now raise a ScopeReenterError when re-entering a scope
- ScopeManager.exit_scope now raise a UndefinedScopeError when exiting an undeclared scope
- Fixed the API documentation

5.3 0.1.0 (23-Oct-2011)

- Initial release

CHAPTER

SIX

MIT LICENSE

CHAPTER

SEVEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

y