Dectate Documentation

Release 0.14.dev0

Martijn Faassen

December 23, 2016

Contents

Dectate is a Python library that lets you construct a decorator-based configuration system for frameworks. Configuration is associated with class objects. It supports configuration inheritance and overrides as well as conflict detection.

CHAPTER 1

Using Dectate

1.1 Introduction

Dectate is a configuration system that can help you construct Python frameworks. A framework needs to record some information about the functions and classes that the user supplies. We call this process *configuration*.

Imagine for instance a framework that supports a certain kind of plugins. The user registers each plugin with a decorator:

```
from framework import plugin

@plugin(name="foo")
def foo_plugin(...):
    ...
```

Here the framework registers as a plugin the function `foo_plugin` under the name `foo`.

You can implement the `plugin` decorator as follows:

```
plugins = {}

class plugin(name):
    def __init__(self, name):
        self.name = name

    def __call__(self, f):
        plugins[self.name] = f
```

In the user application the user makes sure to import all modules that use the `plugin` decorator. As a result, the `plugins` dict contains the names as keys and the functions as values. Your framework can then use this information to do whatever you need to do.

There are a lot of examples of code configuration in frameworks. In a web framework for instance the user can declare routes and assemble middleware.

You may be okay constructing a framework with the simple decorator technique described above. But advanced frameworks need a lot more that the basic decorator system described above cannot offer. You may for instance want to allow the user to reuse configuration, override it, do more advanced error checking, and execute configuration in a particular order.

Dectate supports such advanced use cases. It was extracted from the Morepath web framework.

1.2 Features

Here are some features of Dectate:

- Decorator-based configuration – users declare things by using Python decorators on functions and classes: we call these decorators *directives*, which issue configuration *actions*.
- Dectate detects conflicts between configuration actions in user code and reports what pieces of code are in conflict.
- Users can easily reuse and extend configuration: it's just Python class inheritance.
- Users can easily override configurations in subclasses.
- You can compose configuration actions from other, simpler ones.
- You can control the order in which configuration actions are executed. This is unrelated to where the user uses the directives in code. You do this by declaring *dependencies* between types of configuration actions, and by *grouping* configuration actions together.
- You can declare exactly what objects are used by a type of configuration action to register the configuration – different types of actions can use different registries.
- Unlike normal decorators, configuration actions aren't performed immediately when a module is imported. Instead configuration actions are executed only when the user explicitly *commits* the configuration. This way, all configuration actions are known when they are performed.
- Dectate-based decorators always return the function or class object that is decorated unchanged, which makes the code more predictable for a Python programmer – the user can use the decorated function or class directly in their Python code, just like any other.
- Dectate-based configuration systems are themselves easily extensible with new directives and registries.
- Dectate-based configuration systems can be queried. Dectate also provides the infrastructure to easily construct command-line tools for querying configuration.

1.3 Actions

In Dectate, the simple *plugins* example above looks like this:

```
import dectate

class PluginAction(dectate.Action):
    config = {
       'plugins': dict
    }
    def __init__(self, name):
        self.name = name

    def identifier(self, plugins):
        return self.name

    def perform(self, obj, plugins):
        plugins[self.name] = obj
```

We have formulated a configuration action that affects a `plugins` dict.

1.4 App classes

Configuration in Dectate is associated with special *classes* which derive from *dectate.App*. We also associate the action with it as a directive:

```
class PluginApp(dectate.App):
    plugin = dectate.directive(PluginAction)
```

Let's use it now:

```
@PluginApp.plugin('a')
def f():
    pass # do something interesting

@PluginApp.plugin('b')
def g():
    pass # something else interesting
```

We have registered the function `f` on `PluginApp`. The `name` argument is `'a'`. We've registered `g` under `'b'`.

We can now commit the configuration for `PluginApp`:

```
dectate.commit(PluginApp)
```

Once the commit has successfully completed, we can take a look at the configuration:

```
>>> sorted(PluginApp.config.plugins.items())
[('a', <function f at ...>), ('b', <function g at ...>)]
```

What are the changes between this and the simple plugins example?

The main difference is that the `plugin` decorator is associated with a class and so is the resulting configuration, which gets stored as the `plugins` attribute of *dectate.App.config*. The other difference is that we provide an `identifier` method in the action definition. These differences support configuration *reuse*, *conflicts*, *extension*, *overrides* and *isolation*.

1.4.1 Reuse

You can reuse configuration by simply subclassing `PluginApp`:

```
class SubApp(PluginApp):
   pass
```

We commit both classes:

```
dectate.commit(PluginApp, SubApp)
```

`SubClass` now contains all the configuration declared for `PluginApp`:

```
>>> sorted(SubApp.config.plugins.items())
[('a', <function f at ...>), ('b', <function g at ...>)]
```

So class inheritance lets us reuse configuration, which allows *extension* and *overrides*, which we discuss below.

1.4.2 Conflicts

Consider this example:

```
class ConflictingApp(PluginApp):
    pass

@ConflictingApp.plugin('foo')
def f():
    pass

@ConflictingApp.plugin('foo')
def g():
    pass
```

Which function should be registered for `foo`, `f` or `g`? We should refuse to guess and instead raise an error that the configuration is in conflict. This is exactly what Dectate does:

```
>>> dectate.commit(ConflictingApp)
Traceback (most recent call last):
  ...
ConflictError: Conflict between:
 File "...", line 4
   @ConflictingApp.plugin('foo')
 File "...", line 8
   @ConflictingApp.plugin('foo')
```

As you can see, Dectate reports the lines in which the conflicting configurations occurs.

How does Dectate know that these configurations are in conflict? This is what the `identifier` method in our action definition did:

```
def identifier(self, plugins):
    return self.name
```

We say here that the configuration is uniquely identified by its `name` attribute. If two configurations exist with the same name, the configuration is considered to be in conflict.

1.4.3 Extension

When you subclass configuration, you can also *extend* `SubApp` with additional configuration actions:

```
@SubApp.plugin('c')
def h():
    pass # do something interesting

dectate.commit(PluginApp, SubApp)
```

`SubApp` now has the additional plugin `c`:

```
>>> sorted(SubApp.config.plugins.items())
[('a', <function f at ...>), ('b', <function g at ...>), ('c', <function h at ...>)]
```

But `PluginApp` is unaffected:

```
>>> sorted(PluginApp.config.plugins.items())
[('a', <function f at ...>), ('b', <function g at ...>)]
```

1.4.4 Overrides

What if you wanted to override a piece of configuration? You can do this in `SubApp` by simply reusing the same `name`:

```
@SubApp.plugin('a')
def x():
    pass

dectate.commit(PluginApp, SubApp)
```

In `SubApp` we now have changed the configuration for `a` to register the function `x` instead of `f`. If we had done this for `MyApp` this would have been a conflict, but doing so in a subclass lets you override configuration instead:

```
>>> sorted(SubApp.config.plugins.items())
[('a', <function x at ...>), ('b', <function g at ...>), ('c', <function h at ...>)]
```

But `PluginApp` still uses `f`:

```
>>> sorted(PluginApp.config.plugins.items())
[('a', <function f at ...>), ('b', <function g at ...>)]
```

1.4.5 Isolation

We have already seen in the inheritance and override examples that `PluginApp` is isolated from configuration extension and overrides done for `SubApp`. We can in fact entirely isolate configuration from each other.

We first set up a new base class with a directive, independently from everything before:

```
class PluginAction2(dectate.Action):
    config = {
       'plugins': dict
    }
    def __init__(self, name):
        self.name = name

    def identifier(self, plugins):
        return self.name

    def perform(self, obj, plugins):
        plugins[self.name] = obj

class BaseApp(dectate.App):
    plugin = dectate.directive(PluginAction2)
```

We don't set up any configuration for `BaseApp`; it's intended to be part of our framework. Now we create two subclasses:

```
class OneApp(BaseApp):
    pass

class TwoApp(BaseApp):
    pass
```

As you can see `OneApp` and `TwoApp` are completely isolated from each other; the only thing they share is a common `BaseApp`.

We register a plugin for `OneApp`:

```
@OneApp.plugin('a')
def f():
    pass
```

This won't affect `TwoApp` in any way:

```
dectate.commit(OneApp, TwoApp)
```

```
>>> sorted(OneApp.config.plugins.items())
[('a', <function f at ...>)]
>>> sorted(TwoApp.config.plugins.items())
[]
```

`OneApp` and `TwoApp` are isolated, so configurations are independent, and cannot conflict or override.

1.5 The Anatomy of an Action

Let's consider the plugin action in detail:

```
class PluginAction(dectate.Action):
    config = {
       'plugins': dict
    }
    def __init__(self, name):
        self.name = name

    def identifier(self, plugins):
        return self.name

    def perform(self, obj, plugins):
        plugins[self.name] = obj
```

What is going on here?

- We implement a custom class called `PluginAction` that inherits from *`dectate.Action`*.
- `config` (*`dectate.Action.config`*) specifies that this directive has a configuration effect on `plugins`. We declare that `plugins` is created using the `dict` factory, so our registry is a plain dictionary. You provide any factory function you like here.
- `__init__` specifies the parameters the directive should take and how to store them on the action object. You can use default parameters and such, but otherwise `__init__` should be very simple and not do any registration or validation. That logic should be in `perform`.
- `identifier` (*`dectate.Action.identifier()`*) takes the configuration objects specified by `config` as keyword arguments. It returns an immutable that is unique for this action. This is used to detect conflicts and determine how configurations override each other.
- `perform` (*`dectate.Action.perform()`*) takes `obj`, which is the function or class that the decorator is used on, and the arguments specified in `config`. It should use `obj` and the information on `self` to configure the configuration objects. In this case we store `obj` under the key `self.name` in the `plugins` dict.

We then associate the action with a class as a directive:

```
class PluginApp(dectate.App):
    plugin = dectate.directive(PluginAction)
```

Once we have declared the directive for our framework we can tell programmers to use it.

Directives have absolutely no effect until *commit* is called, which we do with `dectate.commit`. This performs the actions and we can then find the result `PluginApp.config` (*dectate.App.config*).

The results are in `PluginApp.config.plugins` as we set this up with `config` in our `PluginAction`.

1.6 Depends

In some cases you want to make sure that one type of directive has been executed before the other – the configuration of the second type of directive depends on the former. You can make sure this happens by using the `depends` (*dectate.Action.depends*) class attribute.

First we set up a `FooAction` that registers into a `foos` dict:

```
class FooAction(dectate.Action):
    config = {
        'foos': dict
    }
    def __init__(self, name):
        self.name = name

    def identifier(self, foos):
        return self.name

    def perform(self, obj, foos):
        foos[self.name] = obj
```

Now we create a `BarAction` directive that depends on `FooAction` and uses information in the `foos` dict:

```
class BarAction(dectate.Action):
    depends = [FooAction]

    config = {
        'foos': dict,  # also use the foos dict
        'bars': list
    }
    def __init__(self, name):
        self.name = name

    def identifier(self, foos, bars):
        return self.name

    def perform(self, obj, foos, bars):
        in_foo = self.name in foos
        bars.append((self.name, obj, in_foo))
```

In order to use them we need to hook up the actions as directives onto an app class:

```
class DependsApp(dectate.App):
    foo = dectate.directive(FooAction)
    bar = dectate.directive(BarAction)
```

Using `depends` we have ensured that `BarAction` actions are performed after `FooAction` action, no matter what order we use them:

```
@DependsApp.bar('a')
def f():
    pass
```

```
@DependsApp.bar('b')
def g():
    pass

@DependsApp.foo('a')
def x():
    pass

dectate.commit(DependsApp)
```

We expect `in_foo` to be `True` for `a` but to be `False` for `b`:

```
>>> DependsApp.config.bars
[('a', <function f at ...>, True), ('b', <function g at ...>, False)]
```

1.7 config dependencies

In the example above, the items in `bars` depend on the items in `foos` and we've implemented this dependency in the `perform` of `BarAction`.

We can instead make the configuration object for the `BarAction` depend on `foos`. This way `BarAction` does not need to know about `foos`. You can declare a dependency between config objects with the `factory_arguments` attribute of the config factory. Any config object that is created in earlier dependencies of this action, or in the action itself, can be listed in `factory_arguments`. The key and value in `factory_arguments` have to match the key and value in `config` of that earlier action.

First we create a `FooAction` that sets up a `foos` config item as before:

```
class FooAction(dectate.Action):
    config = {
       'foos': dict
    }
    def __init__(self, name):
        self.name = name

    def identifier(self, foos):
        return self.name

    def perform(self, obj, foos):
        foos[self.name] = obj
```

Now we create a `Bar` class that also depends on the `foos` dict by listing it in `factory_arguments`:

```
class Bar(object):
    factory_arguments = {
        'foos': dict
    }

    def __init__(self, foos):
        self.foos = foos
        self.l = []

    def add(self, name, obj):
        in_foo = name in self.foos
        self.l.append((name, obj, in_foo))
```

We create a `BarAction` that depends on the `FooAction` (so that `foos` is created first) and that uses the `Bar` factory:

```
class BarAction(dectate.Action):
   depends = [FooAction]

   config = {
      'bar': Bar
   }

   def __init__(self, name):
       self.name = name

   def identifier(self, bar):
       return self.name

   def perform(self, obj, bar):
       bar.add(self.name, obj)
```

And we set them up as directives:

```
class ConfigDependsApp(dectate.App):
    foo = dectate.directive(FooAction)
    bar = dectate.directive(BarAction)
```

When we use our directives:

```
@ConfigDependsApp.bar('a')
def f():
    pass

@ConfigDependsApp.bar('b')
def g():
    pass

@ConfigDependsApp.foo('a')
def x():
    pass

dectate.commit(ConfigDependsApp)
```

we get the same result as before:

```
>>> ConfigDependsApp.config.bar.l
[('a', <function f at ...>, True), ('b', <function g at ...>, False)]
```

1.8 app_class_arg

In some cases what you want to configure is not on in the config object (`app_class.config`), but is associated with the app class in another way. You can get the app class passed in as an argument to *`dectate.Action.perform()`*, *`dectate.Action.identifier()`*, and so on by setting the special `app_class_arg` class attribute:

```
class PluginAction(dectate.Action):
    config = {
       'plugins': dict
    }
```

```
    app_class_arg = True

    def __init__(self, name):
        self.name = name

    def identifier(self, plugins, app_class):
        return self.name

    def perform(self, obj, plugins, app_class):
        plugins[self.name] = obj
        app_class.touched = True

class MyApp(dectate.App):
   plugin_with_app_class = dectate.directive(PluginAction)
```

When we now perform this directive:

```
@MyApp.plugin_with_app_class('a')
def f():
    pass # do something interesting

dectate.commit(MyApp)
```

We can see the app class was indeed affected:

```
>>> MyApp.touched
True
```

You can also use `app_class_arg` on a factory so that Dectate passes in the `app_class` factory argument.

1.9 before and after

It can be useful to do some additional setup just before all actions of a certain type are performed, or just afterwards. You can do this using `before` (*dectate.Action.before()*) and `after` (*dectate.Action.after()*) static methods on the Action class:

```
class FooAction(dectate.Action):
    config = {
       'foos': list
    }
    def __init__(self, name):
        self.name = name

    @staticmethod
    def before(foos):
        print("before:", foos)

    @staticmethod
    def after(foos):
        print("after:", foos)

    def identifier(self, foos):
        return self.name

    def perform(self, obj, foos):
        foos.append((self.name, obj))
```

```
class BeforeAfterApp(dectate.App):
    foo = dectate.directive(FooAction)

@BeforeAfterApp.foo('a')
def f():
    pass

@BeforeAfterApp.foo('b')
def g():
    pass
```

This executes `before` just before `a` and `b` are configured, and then executes `after`:

```
>>> dectate.commit(BeforeAfterApp)
before: []
after: [('a', <function f at ...>), ('b', <function g at ...>)]
```

1.10 grouping actions

Different actions normally don't conflict with each other. It can be useful to group different actions together in a group so that they do affect each other. You can do this with the `group_class` (*dectate.Action.group_class*) class attribute. Grouped classes share their `config` and their `before` and `after` methods.

```
class FooAction(dectate.Action):
    config = {
       'foos': list
    }
    def __init__(self, name):
        self.name = name

    def identifier(self, foos):
        return self.name

    def perform(self, obj, foos):
        foos.append((self.name, obj))
```

We now create a `BarAction` that groups with `FooAction`:

```
class BarAction(dectate.Action):
    group_class = FooAction

    def __init__(self, name):
        self.name = name

    def identifier(self, foos):
        return self.name

    def perform(self, obj, foos):
        foos.append((self.name, obj))

class GroupApp(dectate.App):
    foo = dectate.directive(FooAction)
    bar = dectate.directive(BarAction)
```

It reuses the `config` from `FooAction`. This means that `foo` and `bar` can be in conflict:

```
@GroupApp.foo('a')
def f():
    pass

@GroupApp.bar('a')
def g():
    pass
```

```
>>> dectate.commit(GroupApp)
Traceback (most recent call last):
  ...
ConflictError: Conflict between:
  File "...", line 8
    @GroupApp.bar('a')
```

1.11 Additional discriminators

In some cases an action should conflict with *multiple* other actions all at once. You can take care of this with the `discriminators` (*dectate.Action.discriminators()*) method on your action:

```
class FooAction(dectate.Action):
    config = {
       'foos': dict
    }
    def __init__(self, name, extras):
        self.name = name
        self.extras = extras

    def identifier(self, foos):
        return self.name

    def discriminators(self, foos):
        return self.extras

    def perform(self, obj, foos):
        foos[self.name] = obj


class DiscriminatorsApp(dectate.App):
    foo = dectate.directive(FooAction)
```

An action now conflicts with an action of the same name *and* with any action that is in the `extra` list:

```
# example
@DiscriminatorsApp.foo('a', ['b', 'c'])
def f():
    pass

@DiscriminatorsApp.foo('b', [])
def g():
    pass
```

And then:

```
>>> dectate.commit(DiscriminatorsApp)
Traceback (most recent call last):
```

```
  ...
ConflictError: Conflict between:
  File "...", line 2:
    @DiscriminatorsApp.foo('a', ['b', 'c'])
  File "...", line 6
    @DiscriminatorsApp.foo('b', [])
```

1.12 Composite actions

When you can define an action entirely in terms of other actions, you can subclass *dectate.Composite*.

First we define a normal SubAction to use in the composite action later:

```
class SubAction(dectate.Action):
    config = {
        'my': list
    }

    def __init__(self, name):
        self.name = name

    def identifier(self, my):
        return self.name

    def perform(self, obj, my):
        my.append((self.name, obj))
```

Now we can define a special *dectate.Composite* subclass that uses SubAction in an actions (*dectate.Composite.actions()*) method:

```
class CompositeAction(dectate.Composite):
    def __init__(self, names):
        self.names = names

    def actions(self, obj):
        return [(SubAction(name), obj) for name in self.names]

class CompositeApp(dectate.App):
    _sub = dectate.directive(SubAction)
    composite = dectate.directive(CompositeAction)
```

Note that even though _sub is not intended to be a public part of the API we still need to include it in our *dectate.App* subclass, as Dectate does need to know it exists.

We can now use it:

```
@CompositeApp.composite(['a', 'b', 'c'])
def f():
    pass

dectate.commit(CompositeApp)
```

And SubAction is performed three times as a result:

```
>>> CompositeApp.config.my
[('a', <function f at ...>), ('b', <function f at ...>), ('c', <function f at ...>)]
```

1.13 `with` statement

Sometimes you want to issue a lot of similar actions at once. You can use the `with` statement to do so with less repetition:

```
class FooAction(dectate.Action):
    config = {
        'my': list
    }

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def identifier(self, my):
        return (self.a, self.b)

    def perform(self, obj, my):
        my.append((self.a, self.b, obj))


class WithApp(dectate.App):
    foo = dectate.directive(FooAction)
```

Instead of this:

```
class VerboseWithApp(WithApp):
    pass

@VerboseWithApp.foo('a', 'x')
def f():
   pass

@VerboseWithApp.foo('a', 'y')
def g():
   pass

@VerboseWithApp.foo('a', 'z')
def h():
   pass
```

You can instead write:

```
class SuccinctWithApp(WithApp):
    pass

with SuccinctWithApp.foo('a') as foo:
    @foo('x')
    def f():
        pass

    @foo('y')
    def g():
        pass

    @foo('z')
    def h():
        pass
```

And this has the same configuration effect:

```
>>> dectate.commit(VerboseWithApp, SuccinctWithApp)
>>> VerboseWithApp.config.my
[('a', 'x', <function f at ...>), ('a', 'y', <function g at ...>), ('a', 'z', <function h at ...>)]
>>> SuccinctWithApp.config.my
[('a', 'x', <function f at ...>), ('a', 'y', <function g at ...>), ('a', 'z', <function h at ...>)]
```

1.14 importing recursively

When you use dectate-based decorators across a package, it can be useful to just import *all* modules in it at once. This way the user cannot forget to import a module with decorators in it.

Dectate itself does not offer this facility, but you can use the importscan library to do this recursive import. Simply do something like:

```
import my_package

importscan.scan(my_package, ignore=['.tests'])
```

This imports every module in `my_package`, except for the `tests` sub package.

1.15 logging

Dectate logs information about the performed actions as debug log messages. By default this goes to the `dectate.directive.<directive_name>` log. You can use the standard Python `logging` module function to make this information go to a log file.

If you want to override the name of the log you can set `logger_name` (*`dectate.App.logger_name`*) on the app class:

```
class MorepathApp(dectate.App):
   logger_name = 'morepath.directive'
```

1.16 querying

Dectate keeps a database of committed actions that can be queried by using *`dectate.Query`*.

Here is an example of a query for all the plugin actions on `PluginApp`:

```
q = dectate.Query('plugin')
```

We can now run the query:

```
>>> list(q(PluginApp))
[(<PluginAction ...>, <function f ...>),
 (<PluginAction ...>, <function g ...>)]
```

We can also filter the query for attributes of the action:

```
>>> list(q.filter(name='a')(PluginApp))
[(<PluginAction object ...>, <function f ...>)]
```

Sometimes the attribute on the action is not the same as the name you may want to use in the filter. You can use *dectate.Action.filter_name* to create a mapping to the correct attribute.

By default the filter does an equality comparison. You can define your own comparison function for an attribute using *dectate.Action.filter_compare*.

If you want to allow a query on a `Composite` action you need to give it some help by defining xs:attr:*dectate.Composite.query_classes*.

1.17 query tool

Dectate also includes a command-line tool that lets you issue queries. You need to configure it for your application. For instance, in the module `main.py` of your project:

```
import dectate

def query_tool():
    # make sure to scan or import everything needed at this point
    dectate.query_tool(SomeApp.commit())
```

In this function you should commit any *dectate.App* subclasses your application normally uses, and then provide an iterable of them to *dectate.query_tool()*. These are the applications that are queried by default if you don't specify `--app`. We do it all in one here as we can get the app class that were committed from the result of `App.commit()`.

Then in `setup.py` of your project:

```
entry_points={
    'console_scripts': [
        'decq = query.main:query_tool',
    ]
},
```

When you re-install this project you have a command-line tool called `decq` that lets you issues queries. For instance, this query returns all uses of directive `foo` in the apps you provided to `query_tool`:

```
$ decq foo
App: <class 'query.a.App'>
  File ".../query/b.py", line 4
  @App.foo(name='alpha')

  File ".../query/b.py", line 9
  @App.foo(name='beta')

  File ".../query/b.py", line 14
  @App.foo(name='gamma')

  File ".../query/c.py", line 4
  @App.foo(name='lah')

App: <class 'query.a.Other'>
  File ".../query/b.py", line 19
  @Other.foo(name='alpha')
```

And this query filters by `name`:

```
$ decq foo name=alpha
App: <class 'query.a.App'>
```

```
  File ".../query/b.py", line 4
  @App.foo(name='alpha')

App: <class 'query.a.Other'>
  File ".../query/b.py", line 19
  @Other.foo(name='alpha')
```

You can also explicit provide the app classes to query with the `--app` option; the default list of app classes is ignored in this case:

```
$ bin/decq --app query.a.App foo name=alpha
App: <class 'query.a.App'>
  File ".../query/b.py", line 4
  @App.foo(name='alpha')
```

You need to give `--app` a dotted name of the *dectate.App* subclass to query. You can repeat the `--app` option to query multiple apps.

Not all things you would wish to query on are string attributes. You can provide a conversion function that takes the string input and converts it to the underlying object you want to compare to using *dectate.Action.filter_convert*.

A working example is in `scenarios/query` of the Dectate project.

1.18 Sphinx Extension

If you use Sphinx to document your project and you use the `sphinx.ext.autodoc` extension to document your API, you need to install a Sphinx extension so that directives are documented properly. In your Sphinx `conf.py` add `'dectate.sphinxext'` to the `extensions` list.

1.19 `__main__` and conflicts

Import-time side effects are evil

This scenario is based on the one described in Application programmers don't control the module-scope codepath in the Pyramid design defense document. If you're curious, look under `scenarios/main_module` in the Dectate project for a Dectate version.
Dectate makes a different compromise than Venusian – it reports an error if a directive is executed because of a double import, so it won't get you into trouble. But since Dectate's directives cause registrations to happen immediately (but defer configuration), you can dynamically generate them inside Python function, which won't work with with Venusian.

In certain scenarios where you run your code like this:

```
$ python app.py
```

and you use `__name__ == '__main__'` to determine whether the module should run:

```
if __name__ == '__main__':
    import another_module
    dectate.commit(App)
```

you might get a `ConflictError` from Dectate that looks somewhat like this:

```
Traceback (most recent call last):
 ...
dectate.error.ConflictError: Conflict between:
  File "/path/to/app.py", line 6
    @App.foo(name='a')
  File "app.py", line 6
    @App.foo(name='a')
```

The same line shows up on *both* sides of the configuration conflict, but the path is absolute on one side and relative on the other.

This happens because in some scenarios involving `__main__`, Python imports a module *twice* (more about this). Dectate refuses to operate in this case until you change your imports so that this doesn't happen anymore.

How to avoid this scenario? If you use setuptools automatic script creation this problem is avoided entirely.

Fooling Dectate after all

It *is* possible to fool Dectate into accepting a double import without conflicts, but you'd need to work hard. You need to use a global variable that gets modified during import time and then use it as a directive argument. If you want to dynamically generate directives then don't do that in module-scope – do it in a function.

If you want to use the `if __name__ == '__main__'` system, keep your main module tiny and just import the main function you want to run from elsewhere.

So, Dectate warns you if you do it wrong, so don't worry about it.

CHAPTER 2

API

dectate.**commit**(**apps*)
Commit one or more app classes

A commit causes the configuration actions to be performed. The resulting configuration information is stored under the `.config` class attribute of each *App* subclass supplied.

This function may safely be invoked multiple times – each time the known configuration is recommitted.

Parameters ***apps** – one or more *App* subclasses to perform configuration actions on.

dectate.**topological_sort**(*l*, *get_depends*)
Topological sort

Given a list of items that depend on each other, sort so that dependencies come before the dependent items. Dependency graph must be a DAG.

Parameters

- **l** – a list of items to sort
- **get_depends** – a function that given an item gives other items that this item depends on. This item will be sorted after the items it depends on.

Returns the list sorted topologically.

class dectate.**App**
A configurable application object.

Subclass this in your framework and add directives using the `App.directive()` decorator.

Set the `logger_name` class attribute to the logging prefix that Dectate should log to. By default it is `"dectate.directive"`.

classmethod **clean**()
A method that sets or restores the state of the class.

Normally Dectate only sets up configuration into the `config` attribute, but in some cases you may touch other aspects of the class during configuration time. You can override this classmethod to set up the state of the class in its pristine condition.

classmethod **commit**()
Commit this class and any depending on it.

This is intended to be overridden by subclasses if committing the class also commits other classes automatically, such as in the case in Morepath when one app is mounted into another. In such case it should return an iterable of all committed classes.

Returns an iterable of committed classes

classmethod **is_committed**()
True if this app class was ever committed.

Returns bool that is `True` when the app was committed before.

config = <dectate.app.Config object>
Config object that contains the configuration after commit.

This is installed when the class object is initialized, so during import-time when you use the `class` statement and subclass *dectate.App*, but is only filled after you commit the configuration.

This keeps the final configuration result after commit. It is a very dumb object that has no methods and is just a container for attributes that contain the real configuration.

dectate = <dectate.config.Configurable object>
A dectate Configurable instance is installed here.

This is installed when the class object is initialized, so during import-time when you use the `class` statement and subclass *dectate.App*.

This keeps tracks of the registrations done by using directives as long as committed configurations.

logger_name = 'dectate.directive'
The prefix to use for directive debug logging.

class dectate.**Action**
A configuration action.

Base class of configuration actions.

A configuration action is performed for an object (typically a function or a class object) and affects one or more configuration objects.

Actions can conflict with each other based on their identifier and discriminators. Actions can override each other based on their identifier. Actions can only be in conflict with actions of the same action class or actions with the same `action_group`.

static **after**(***kw*)
Do setup just after actions in a group are performed.

Can be implemented as a static method by the *Action* subclass.

Parameters ****kw** – a dictionary of configuration objects as specified by the `config` class attribute.

static **before**(***kw*)
Do setup just before actions in a group are performed.

Can be implemented as a static method by the *Action* subclass.

Parameters ****kw** – a dictionary of configuration objects as specified by the `config` class attribute.

discriminators(***kw*)
Returns an iterable of immutables to detect conflicts.

Can be implemented by the *Action* subclass.

Used for additional configuration conflict detection.

Parameters ****kw** – a dictionary of configuration objects as specified by the `config` class attribute.

Returns an iterable of immutable values.

filter_get_value(*name*)
A function to get the filter value.

Takes two arguments, action and name. Should return the value on the filter.

This function is called if the name cannot be determined by looking for the attribute directly using *Action.filter_name*.

The function should return *NOT_FOUND* if no value with that name can be found.

For example if the filter values are stored on `key_dict`:

```
def filter_get_value(self, name):
    return self.key_dict.get(name, dectate.NOT_FOUND)
```

Parameters **name** – the name of the filter.

Returns the value to filter on.

get_value_for_filter(*name*)
Get value. Takes into account `filter_name`, `filter_get_value`

Used by the query system. You can override it if your action has a different way storing values altogether.

Parameters **name** – the filter name to get the value for.

Returns the value to filter on.

identifier(***kw*)
Returns an immutable that uniquely identifies this config.

Needs to be implemented by the *Action* subclass.

Used for overrides and conflict detection.

If two actions in the same group have the same identifier in the same configurable, those two actions are in conflict and a *ConflictError* is raised during *commit()*.

If an action in an extending configurable has the same identifier as the configurable being extended, that action overrides the original one in the extending configurable.

Parameters ****kw** – a dictionary of configuration objects as specified by the `config` class attribute.

Returns an immutable value uniquely identifying this action.

perform(*obj*, ***kw*)
Do whatever configuration is needed for `obj`.

Needs to be implemented by the *Action* subclass.

Raise a *DirectiveError* to indicate that the action cannot be performed due to incorrect configuration.

Parameters

- **obj** – the object that the action should be performed for. Typically a function or a class object.
- ****kw** – a dictionary of configuration objects as specified by the `config` class attribute.

app_class_arg = False
Pass in app class as argument.

In addition to the arguments defined in *Action.config*, pass in the app class itself as an argument into *Action.identifier()*, *Action.discriminators()*, *Action.perform()*, and *Action.before()* and *Action.after()*.

code_info
Info about where in the source code the action was invoked.

Is an instance of *CodeInfo*.

Can be `None` if action does not have an associated directive but was created manually.

config = {}
Describe configuration.

A dict mapping configuration names to factory functions. The resulting configuration objects are passed into *Action.identifier()*, *Action.discriminators()*, *Action.perform()*, and *Action.before()* and *Action.after()*.

After commit completes, the configured objects are found as attributes on *App.config*.

depends = []
List of other action classes to be executed before this one.

The `depends` class attribute contains a list of other action classes that need to be executed before this one is. Actions which depend on another will be executed after those actions are executed.

Omit if you don't care about the order.

filter_compare = {}
Map of names used in query filter to comparison functions.

If for instance you want to be able check whether the value of `model` on the action is a subclass of the value provided in the filter, you can provide it here:

```
filter_compare = {
    'model': issubclass
}
```

The default filter compare is an equality comparison.

filter_convert = {}
Map of names to convert functions.

The query tool that can be generated for a Dectate-based application uses this information to parse filter input into actual objects. If omitted it defaults to passing through the string unchanged.

A conversion function takes a string as input and outputs a Python object. The conversion function may raise `ValueError` if the conversion failed.

A useful conversion function is provided that can be used to refer to an object in a module using a dotted name: *convert_dotted_name()*.

filter_name = {}
Map of names used in query filter to attribute names.

If for instance you want to be able to filter the attribute `_foo` using `foo` in the query, you can map `foo` to `_foo`:

```
filter_name = {
   'foo': '_foo'
}
```

If a filter name is omitted the filter name is assumed to be the same as the attribute name.

group_class = None
Action class to group with.

This class attribute can be supplied with the class of another action that this action should be grouped with. Only actions in the same group can be in conflict. Actions in the same group share the `config` and `before` and `after` of the action class indicated by `group_class`.

By default an action only groups with others of its same class.

class `dectate.`**`Composite`**

A composite configuration action.

Base class of composite actions.

Composite actions are very simple: implement the `action` method and return a iterable of actions in there.

`actions`(*obj*)

Specify a iterable of actions to perform for `obj`.

The iteratable should yield `action, obj` tuples, where `action` is an instance of class *`Action`* or *`Composite`* and `obj` is the object to perform the action with.

Needs to be implemented by the *`Composite`* subclass.

Parameters **`obj`** – the obj that the composite action was performed on.

Returns iterable of `action, obj` tuples.

`code_info`

Info about where in the source code the action was invoked.

Is an instance of *`CodeInfo`*.

Can be `None` if action does not have an associated directive but was created manually.

`filter_convert` = {}

Map of names to convert functions.

The query tool that can be generated for a Dectate-based application uses this information to parse filter input into actual objects. If omitted it defaults to passing through the string unchanged.

A conversion function takes a string as input and outputs a Python object. The conversion function may raise `ValueError` if the conversion failed.

A useful conversion function is provided that can be used to refer to an object in a module using a dotted name: *`convert_dotted_name()`*.

`query_classes` = []

A list of actual action classes that this composite can generate.

This is to allow the querying of composites. If the list if empty (the default) the query system refuses to query the composite. Note that if actions of the same action class can also be generated in another way they are in the same query result.

class `dectate.`**`Query`**(**action_classes*)

An object representing a query.

A query can be chained with *`Query.filter()`*, *`Query.attrs()`*, *`Query.obj()`*.

Param `*action_classes`: one or more action classes to query for. Can be instances of *`Action`* or *`Composite`*. Can also be strings indicating directive names, in which case they are looked up on the app class before execution.

`attrs`(**names*)

Extract attributes from resulting actions.

The list of attribute names indicates which keys to include in the dictionary. Obeys *`Action.filter_name`* and *`Action.filter_get_value`*.

Param `*names`: list of names to extract.

Returns iterable of dictionaries.

filter(***kw*)

Filter this query by keyword arguments.

The keyword arguments are matched with attributes on the action. *Action.filter_name* is used to map keyword name to attribute name, by default they are the same. *Action.filter_get_value()* can also be implemented for more complicated attribute access as a fallback.

By default the keyword argument values are matched by equality, but you can override this using *Action.filter_compare*.

Can be chained again with a new `filter`.

Parameters ****kw** – keyword arguments to match against.

Returns iterable of `(action, obj)`.

obj()

Get objects from results.

Throws away actions in the results and return an iterable of objects.

Returns iterable of decorated objects.

dectate.**directive**(*action_factory*)

Create a classmethod to hook action to application class.

You pass in a *dectate.Action* or a *dectate.Composite* subclass and can attach the result as a class method to an *dectate.App* subclass:

```
class FooAction(dectate.Action):
    ...

class MyApp(dectate.App):
    my_directive = dectate.directive(MyAction)
```

Alternatively you can also define the direction inline using this as a decorator:

```
class MyApp(dectate.App):
    @directive
    class my_directive(dectate.Action):
        ...
```

Parameters **action_factory** – an action class to use as the directive.

Returns a class method that represents the directive.

dectate.**query_tool**(*app_classes*)

Command-line query tool for dectate.

Uses command-line arguments to do the query and prints the results.

usage: decq [-h] [–app APP] directive <filter>

Query all directives named `foo` in given app classes:

```
$ decq foo
```

Query directives `foo` with `name` attribute set to `alpha`:

```
$ decq foo name=alpha
```

Query directives `foo` specifically in given app:

```
$ decq --app=myproject.App foo
```

Parameters **app_classes** – a list of *App* subclasses to query by default.

dectate.**query_app**(*app_class*, *directive*, ***filters*)
Query a single app with raw filters.

This function is especially useful for writing unit tests that test the conversion behavior.

Parameters

- **app_class** – a *App* subclass to query.
- **directive** – name of directive to query.
- ****filters** – raw (unconverted) filter values.

Returns iterable of `action, obj` tuples.

dectate.**convert_dotted_name**(*s*)
Convert input string to an object in a module.

Takes a dotted name: `pkg.module.attr` gets `attr` from module `module` which is in package `pkg`.

To refer to builtin objects such as `int` or `object`, in Python 2 prefix with `__builtin__.`, so `__builtin__.int` or `__builtin__.None`. In Python 3 use `builtins.` as the prefix, so `builtins.int` and `builtins.None`.

Raises `ValueError` if it cannot be imported.

dectate.**convert_bool**(*s*)
Convert input string to boolean.

Input string must either be `True` or `False`.

dectate.**NOT_FOUND = <NOT_FOUND>**
Sentinel value returned if filter value cannot be found on action.

class dectate.**CodeInfo**(*path*, *lineno*, *sourceline*)
Information about where code was invoked.

The `path` attribute gives the path to the Python module that the code was invoked in.

The `lineno` attribute gives the linenumber in that file.

The `sourceline` attribute contains the actual source line that did the invocation.

exception dectate.**ConfigError**
Raised when configuration is bad.

exception dectate.**ConflictError**(*actions*)
Bases: `dectate.error.ConfigError`

Raised when there is a conflict in configuration.

Describes where in the code directives are in conflict.

exception dectate.**DirectiveError**
Bases: `dectate.error.ConfigError`

Can be raised by user when there directive cannot be performed.

Raise it in *Action.perform()* with a message describing what the problem is:

```
raise DirectiveError("name should be a string, not None")
```

This is automatically converted by Dectate to a *DirectiveReportError*.

exception dectate.**DirectiveReportError**(*message*, *code_info*)
Bases: dectate.error.ConfigError

Raised when there's a problem with a directive.

Describes where in the code the problem occurred.

exception dectate.**TopologicalSortError**
Bases: exceptions.ValueError

Raised if dependencies cannot be sorted topologically.

This is due to circular dependencies.

CHAPTER 3

Developing Dectate

3.1 Install Dectate for development

Clone Dectate from github:

```
$ git clone git@github.com:morepath/dectate.git
```

If this doesn't work and you get an error 'Permission denied (publickey)', you need to upload your ssh public key to github.

Then go to the dectate directory:

```
$ cd dectate
```

Make sure you have virtualenv installed.

Create a new virtualenv for Python 3 inside the dectate directory:

```
$ virtualenv -p python3 env/py3
```

Activate the virtualenv:

```
$ source env/py3/bin/activate
```

Make sure you have recent setuptools and pip installed:

```
$ pip install -U setuptools pip
```

Install the various dependencies and development tools from develop_requirements.txt:

```
$ pip install -Ur develop_requirements.txt
```

For upgrading the requirements just run the command again.

If you want to test Dectate with Python 2.7 as well you can create a second virtualenv for it:

```
$ virtualenv -p python2.7 env/py27
```

You can then activate it:

```
$ source env/py27/bin/activate
```

Then uprade setuptools and pip and install the develop requirements as described above.

Note: The following commands work only if you have the virtualenv activated.

3.2 Running the tests

You can run the tests using py.test:

```
$ py.test
```

To generate test coverage information as HTML do:

```
$ py.test --cov --cov-report html
```

You can then point your web browser to the `htmlcov/index.html` file in the project directory and click on modules to see detailed coverage information.

3.3 Running the documentation tests

The documentation contains code. To check these code snippets, you can run this code using this command:

```
(py3) $ sphinx-build -b doctest doc doc/build/doctest
```

Or alternatively if you have `Make` installed:

```
(py3) $ cd doc
(py3) $ make doctest
```

Or from the Dectate project directory:

```
(py3) $ make -C doc doctest
```

Since the sample code in the documentation is maintained in Python 3 syntax, we do not support running the doctests with Python 2.7.

3.4 Building the HTML documentation

To build the HTML documentation (output in `doc/build/html`), run:

```
$ sphinx-build doc doc/build/html
```

Or alternatively if you have `Make` installed:

```
$ cd doc
$ make html
```

Or from the Dectate project directory:

```
$ make -C doc html
```

3.5 Various checking tools

flake8 is a tool that can do various checks for common Python mistakes using pyflakes, check for PEP8 style compliance and can do cyclomatic complexity checking. To do pyflakes and pep8 checking do:

```
$ flake8 dectate
```

To also show cyclomatic complexity, use this command:

```
$ flake8 --max-complexity=10 dectate
```

3.6 Tox

With tox you can test Morepath under different Python environments.

We have Travis continuous integration installed on Morepath's github repository and it runs the same tox tests after each checkin.

First you should install all Python versions which you want to test. The versions which are not installed will be skipped. You should at least install Python 3.5 which is required by flake8, coverage and doctests and Python 2.7 for testing Morepath with Python 2.

One tool you can use to install multiple versions of Python is pyenv.

To find out which test environments are defined for Morepath in tox.ini run:

```
$ tox -l
```

You can run all tox tests with:

```
$ tox
```

You can also specify a test environment to run e.g.:

```
$ tox -e py35
$ tox -e pep8
$ tox -e docs
```

CHAPTER 4

History of Dectate

Dectate was extracted from Morepath and then extensively refactored and cleaned up. It is authored by me, Martijn Faassen.

In the beginning (around 2001) there was zope.configuration, part of the Zope 3 project. It features declarative XML configuration with conflict detection and overrides to assemble pieces of Python code.

In 2006, I helped create the Grok project. This did away with the XML based configuration and instead used Python code. This in turn then drove *zope.configuration*. Grok did not use Python decorators but instead used specially annotated Python classes, which were recursively scanned from modules. Grok's configuration system was spun off as the Martian library.

Chris McDonough was then inspired by Martian to create Venusian, a deferred decorator execution system. It is like Martian in that it imports Python modules recursively in order to find configuration.

I created the Morepath web framework, which uses decorators for configuration throughout and used Venusian. Morepath grew a configuration subsystem where configuration is associated with classes, and uses class inheritance to power configuration reuse and overrides. This configuration subsystem started to get a bit messy as requirements grew.

So in 2016 I extracted the configuration system from Morepath into its own library, Dectate. This allowed me to extensively refactor the code for clarity and features. Dectate does not use Venusian for configuration. Dectate still defers the execution of configuration actions to an explicit commit phase, so that conflict detection and overrides and such can take place.

CHAPTER 5

CHANGES

5.1 0.14 (unreleased)

- Nothing changed yet.

5.2 0.13 (2016-12-23)

- Add a Sentinel class, used for the `NOT_FOUND` object.
- Upload universal wheels to pypi during release.
- make `directive_name` property available on `Directive` instances.

5.3 0.12 (2016-10-04)

- **Breaking changes**: previously you defined new directives using the `App.directive` directive. This would lead to import confusion: you *have* to import the modules that define directives before you can actually use them, even though you've already imported your app class.

  In this version of Dectate we've changed the way you define directives. Instead of:

```
class MyApp(dectate.App):
    pass

@MyApp.directive('foo')
class FooAction(dectate.Action):
   ...
```

  You now write this:

```
class FooAction(directive.Action)
    ...

class MyApp(dectate.App):
    foo = directive(FooAction)
```

  So, you define the directives directly on the app class that needs them.

  Uses of `private_action_class` should be replaced by an underscored directive definition:

```
class MyApp(dectate.App):
    _my_private_thing = directive(PrivateAction)
```

- Use the same Git ignore file used in other Morepath projects.
- If you set the `app_class_arg` class attribute to `True` on an action, then an `app_class` is passed along to `perform`, `identifier`, etc. This way you can affect the app class directly instead of just its underlying configuration in the `config` attribute.
- Similarly if you set the `app_class_arg` attribute `True` on a factory class, it is passed in.
- Add a `clean` method to the `App` class. You can override this to introduce your own cleanup policy for aspects of the class that are not contained in the `config` attribute.
- We now use virtualenv and pip instead of buildout to set up the development environment. The development documentation has been updated accordingly.
- Include doctests in Tox and Travis.

5.4 0.11 (2016-07-18)

- **Removed**: `autocommit` was removed from the Dectate API. Rely on the `commit` class method of the `App` class instead for a more explicit alternative.
- **Removed**: `auto_query_tool` was removed from the Dectate API. Use `query_tool(App.commit())` instead.
- Fix `repr` of directives so that you can at least see their name.
- the execution order of filters is now reproducible, to ensure consistent test coverage reports.
- Use abstract base classes from the standard library for the `Action` and `Composite` classes.
- Use feature detection instead of version detection to ensure Python 2/3 compatibility.
- Increased test coverage.
- Set up Travis CI and Coverall as continuous integration services for quality assurance purposes.
- Add support for Python 3.3 and 3.5.
- Make Python 3.5 the default testing environment.

5.5 0.10.2 (2016-04-26)

- If nothing is found for an app in the query tool, don't mention it in the output so as to avoid cluttering the results.
- Fix a major bug in the query tool where if an app resulted in no results, any subsequent apps weren't even searched.

5.6 0.10.1 (2016-04-26)

- Create proper deprecation warnings instead of plain warnings for `autocommit` and `auto_query_tool`.

5.7 0.10 (2016-04-25)

- **Deprecated** The `autocommit` function is deprecated. Rely on the `commit` class method of the `App` class instead for a more explicit alternative.
- **Deprecated** The `auto_query_tool` function is deprecated. Rely on `dectate.query_tool(MyApp.commit())` instead. Since the commit method returns an iterable of `App` classes that are required to commit the app class it is invoked on, this returns the right information.
- `topological_sort` function is exposed as the public API.
- A `commit` class method on `App` classes.
- Report on inconsistent uses of factories between different directives' `config` settings as well as `factory_arguments` for registries. This prevents bugs where a new directive introduces the wrong factory for an existing directive.
- Expanded internals documentation.

5.8 0.9.1 (2016-04-19)

- Fix a subtle bug introduced in the last release. If `factory_arguments` were in use with a config name only created in that context, it was not properly cleaned up, which in some cases can make a commit of a subclass get the same config object as that of the base class.

5.9 0.9 (2016-04-19)

- Change the behavior of `query_tool` so that if it cannot find an action class for the directive name the query result is empty instead of making this an error. This makes `auto_query_tool` work better.
- Introduce `auto_query_tool` which uses the automatically found app classes as the default app classes to query.
- Fix tests that use `__builtin__` that were failing on Python 3.
- Dependencies only listed in `factory_arguments` are also created during config creation.

5.10 0.8 (2016-04-12)

- Document how to refer to builtins in Python 3.
- Expose `is_committed` method on `App` subclasses.

5.11 0.7 (2016-04-11)

- Fix a few documentation issues.
- Expose `convert_dotted_name` and document it.
- Implement new `convert_bool`.
- Allow use of directive name instead of Action subclass as argument to Query.

- A `query_app` function which is especially helpful when writing tests for the query tool – it takes unconverted filter arguments.
- Use newer version of `with_metaclass` from six.
- Expose `NOT_FOUND` and document it.
- Introduce a new `filter_get_value` method you can implement if the normal attribute getting and `filter_name` are not enough.

5.12 0.6 (2016-04-06)

- Introduce a query system for actions and a command-line tool that lets you query actions.

5.13 0.5 (2016-04-04)

- **Breaking change** The signature of `commit` has changed. Just pass in one or more arguments you want to commit instead of a list. See #8.

5.14 0.4 (2016-04-01)

- Expose `code_info` attribute for action. The `path` in particular can be useful in implementing a directive such as Morepath's `template_directory`. Expose it for composite too.
- Report a few more errors; you cannot use `config`, `before` or `after` after in an action class if `group_class` is set.
- Raise a DirectiveReportError if a DirectiveError is raised in a composite `actions` method.

5.15 0.3 (2016-03-30)

- Document `importscan` package that can be used in combination with this one.
- Introduced `factory_arguments` feature on `config` factories, which can be used to create dependency relationships between configuration.
- Fix a bug where config items were not always properly reused. Now only the first one in the action class dependency order is used, and it is not recreated.

5.16 0.2 (2016-03-29)

- Remove clear_autocommit as it was useless during testing anyway. In tests just use explicit commit.
- Add a `dectate.sphinxext` module that can be plugged into Sphinx so that directives are documented properly.
- Document how Dectate deals with double imports.

5.17 0.1 (2016-03-29)

- Initial public release.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

d