Your First Application

Mehdi Sadeqi

Contents

1 Adding a new application

Writing a new application using Deltapy application server framework is really easy. First create a python package for new application:

```
mkdir sample_application
touch sample_application/__init__.py
```

Each deltapy application has a *settings* package inside it, so first we should add the basic needed configuration to application package. Read 2 section to understand how to do that for the sample application.

In order to recognize the package as an application by Deltapy application loader, we should add a class to `__init__.py` module in the main package that inherits `deltapy.application.Application` class:

```
from deltapy.application.base import Application

class SampleApplication(Application):
    '''
    Sample application for demonstration.
    '''
```

The application is ready. Although, it has no commands and services, so check the 3 to write a sample *add* command. To run the new application, in a module or python interpreter enter:

```
>>> from sample_application import SampleApplication
>>> sample_app = SampleApplication()
>>> sample_app.run()
>> Initializing application[sample_application] ...
>>> sample_app.run()
>> Loading application[sample_application].
>> Disabled packages:
Package [deltapy.config] loaded.
Package [deltapy.logging] loaded.
Package [deltapy.database] loaded.
Package [deltapy.transaction] loaded.
Package [deltapy.commander] loaded.
Package [deltapy.event_system] loaded.

>> deltapy.event_system:[1] packages loaded.
>> Total loaded packages: 6
>> Disabled packages: 0

>> Disabled packages:
Package [deltapy.scheduling] loaded.
Package [deltapy.caching.database] loaded.
Package [deltapy.caching.file] loaded.
Package [deltapy.caching.remote] loaded.
Package [deltapy.caching] loaded.
Package [deltapy.unique_id.generators.uuid_generator] loaded.
Package [deltapy.unique_id.generators] loaded.
Package [deltapy.unique_id] loaded.
Package [deltapy.system] loaded.
Package [deltapy.security.database] loaded.
Package [deltapy.security.authentication] loaded.
Package [deltapy.security.session] loaded.
Package [deltapy.security.null] loaded.
Package [deltapy.security.authorization] loaded.
Package [deltapy.security] loaded.
Package [deltapy.request_processor.multiprocess] loaded.
Package [deltapy.request_processor.multithread] loaded.
Package [deltapy.request_processor.complex_multiprocess] loaded.
Package [deltapy.request_processor] loaded.
Package [deltapy.communication.xmlrpc] loaded.
Package [deltapy.communication.ice] loaded.
Package [deltapy.communication.pyro] loaded.
Package [deltapy.communication.pymp] loaded.
Package [deltapy.communication] loaded.
Package [deltapy.integeration] loaded.
Package [deltapy.batch] loaded.
Package [deltapy.scripting] loaded.
Package [deltapy.storm] loaded.

>> deltapy:[34] packages loaded.
>> Total loaded packages: 34
>> Disabled packages: 0

>> Disabled packages:

>> sample_application:[0] packages loaded.
>> Total loaded packages: 34
>> Disabled packages: 0

>> Application[sample_application] loaded.
>> Security services activated.
```

Congratulations! The application is up and running. Connect to it with any of different protocol types you defined previously using DeltaConsole tool[see it's documentation]:

```
mehdi@FREEDOM:~$ deltaconsole -n 127.0.0.1 -p 19083 -t xmlrpc
or
mehdi@FREEDOM:~$ deltaconsole -n 127.0.0.1 -p 19082 -t pyro
```

```
or
mehdi@FREEDOM:~$ deltaconsole -n 127.0.0.1 -p 19081 -t ice
You will see your application console and you can run framework specific or your applica
mehdi@FREEDOM:~$ deltaconsole -n 127.0.0.1 -p 19081 -t ice -uroot
    ___     _ _             ___                       _
   /   \___| | |_  __ _    / __\ ___  _ __  ___  ___ | | ___
  / /\ / _ \ | __|/ _` |  / /   / _ \| '_ \/ __|/ _ \| |/ _ \
 / /_//  __/ | |_| (_| | / /___| (_) | | | \__ \ (_) | |  __/
/___,' \___|_|\__|\__,_| \____/ \___/|_| |_|___/\___/|_|\___|
    version : 3.1

 Type 'help' for more information

Password:
sample_application/>
```

Run the newlly added *add* command:

```
sample_application/> sample_app.add(2 + 3)
44
Command takes [0.0460250377655] seconds to execute.
sample_application/>
```

Read 3([how to add commands and services]) document, to learn how to add more commands.

2 Deltapy application configuration (for sample application)

Deltapy has its own configuration system. Most of Deltapy features use configuration files. Each Deltapy application should have a *settings* folder in it's package with a few config files. So we should create the settings package in the application folder:

```
mkdir sample_application/settings
touch sample_application/settings/__init__.py
```

Each Deltapy application should have the following configuration files in it's settings folder:

- app.config
- communication.config
- request_processor.config
- logging.config

- database.config

So add above files to settings folder. There are some other config files that are not mandatory when using version.

2.1 Adding logging configuration

An application can have any number of loggers introduces in logging configuration file. Each logger can write it's output to a seperate log file or to a common log file. First we add the logging config file:

```
touch sample_application/settings/logging.config
```

Then we add the basic root logger to it:

```
[loggers]
keys=root

[handlers]
keys=console, root_file, root_syslog

[formatters]
keys=base

[logger_root]
level=DEBUG
handlers=root_file

[handler_root_file]
class=handlers.TimedRotatingFileHandler
formatter=base
args=('/var/log/corebanking/root.log','D' , 1, 7)

[handler_console]
class=StreamHandler
formatter=base
args=(sys.stdout,)

[formatter_base]
format=[%(levelname)s]-[%(process)d]-[%(asctime)s]-[%(name)s]: %(message)s

[handler_root_syslog]
class=handlers.SysLogHandler
formatter=base
args=(('localhost', handlers.SYSLOG_UDP_PORT), handlers.SysLogHandler.LOG_USER)
```

You can add any number of loggers and handlers.

2.2 Adding database configuration

Like logging configuration, you can have any number of database sections in database configuration file. First we create the file:

```
touch sample_application/settings/database.config
```

The we add the sections for our database. Currently supported databases are: *sqlite*, *mysql*, *postgresq*l

```
[global]
default:local

[local]
connection:sqlite:./sample_app.db
min:0
max:0
growup:0
```

Again we can add any number of database sections, each for one database. We should introduce default section to be used in `[global]` section.

`min` and `max` are the minimum and maximum number of connections to this database that the application will keep in it's connection pool. The growup is the number of connections that will be added temporarily to connection pool once needed.

2.3 Adding communication configuration

Deltapy supports multiple communication channles. You can define differet protocols, IPs and ports to connect to the new appliction. Currenty *XMLRPC*, *Pyro* and *Ice* are supported. To use each of these communication channles, related modules should be installed on your system. Add the following code to `communication.config` file:

```
[pyro]
type=pyro
host=0.0.0.0
port=19082

[ice]
type=ice
host=192.168.22.83
port=19081

[xmlrpc]
type=xmlrpc
host=0.0.0.0
port=19083
```

2.4 Adding request processor configuration

This configuration defines how the application will handle the requests, multithreaded or multiprocess or mix. The name defined in global section is the default behaviour.

Add the following code to `request_process.config` file:

```
[global]
default = multithread

[multithread]
max_threads = 64

[multiprocess]
max_processes = 8

[complex_multiprocess]
max_threades = 8
max_processes = 8
```

These are the very basic configuration needed to run a Deltapy application.

3 DeltaPy Application Architecture

Each Deltapy application should respect the following well defined application architechture. There are four different and important layers [and also concepts] that deltapy applications are built upon them: Package, Command, Service and Component.

```
+ Application
  + Package [any number of packages are allowed]
    + Commands
      - Command Modules
      - __init__.py [Contains only functions]
    + Services
      - Service Modules
      - __init__.py [Contains only functinos]
    + Components
      - Component Modules
      - __init__.py [Contains SampleApplicationFirstComponent class]
    - __init__.py [Contains SampleApplicationFirstPackage class]
  - __init__.py [Contains SampleApplication class]
```

Each well designed Deltapy application should have it's three different layer in three different packages: *command*, *service* and *component* package. These are encapsulated in Packages. So we add the first Package to application:

```
mkdir sample_application/first_package
touch sample_application/first_package/__init__.py
```

Each application can have any number of Packages in it. Package is a Deltapy class that can contain command, service and component. For a python package to be recognized by Deltapy application loader as a Deltapy Package it should inherit from Deltapy Package class in it's `__init__.py` module [Contents of `sample_application/first_package/__init__.py` module]:

```
from deltapy.packaging.package import Package

# This is the string which will be used to register the
    first app component and also to find it in runtime,
    and should be unique.
SAMPLE_APP_FIRST_COMPONENT_ID =
    'sample_app.first_package'

class SampleApplicationFirstPackage(Package):
    '''
    A package is a container for commands, service and
        component.
    '''
```

The Deltapy application loader will walk through the application tree to find Packages and load their commands and components.

Commands are the external interface of each application that are being executed from any communication channel(using tools like *DeltaConsole* or any other client which supports Deltapy communication channles). Actually, commands are functions defined in a module that has been decorated with deltapy `command` decorator, we will add a sample command soon. Each command exposes a single atomic feature of your application. Deltapy application loader will walk through all over your application packages to find command packages and modules in it and exposing them to outside world. So we add the command package and module:

```
mkdir sample_application/first_package/commands
touch sample_application/first_package/commands/__init__.py
```

Then we edit the `sample_application/first_package/commands/__init__.py` module to add a new command:

```
from deltapy.commander.decorators import command
import sample_application.first_package.services as
    first_package_services

@command('sample_app.first_package.add')
def add(a, b):
    '''
    Returns a + b [No implementation here.]

    @param a: int
    @param b: int

    @return: int
    '''
    return first_package_services.add(a, b)
```

Note the text within command decorator, it will be the name of the exposing command, and the only name of it.

Remember that commands never have any implementation, the only call the appropriate service and return the result. And, commands only are being executed via communication channle and it is not possible to run a command inside the application, we will use services to do it.

Service is the layer between commands and components[the real implementation]. In other words the services are the internal interface of a deltapy application. Services are the standard way for a Deltapy application to expose services to other parts of the application. Services are functions too, that are defined within service package.

Services are not directly depended upon components, but they use the application context to find the component dynamically and call it's appropriate method. Again, services never have any implemenation, they are only intented to find and call the appropriate component method and then return it's result. We add the service package the same way as before:

```
mkdir sample_application/first_package/services
touch sample_application/first_package/services/__init__.py
```

Then we edit the `sample_application/first_package/services/__init__.py` module to add a new service for previously added command:

```
from deltapy.application.services import get_component
from sample_application.first_package import
    SAMPLE_APP_FIRST_COMPONENT_ID

def add(a, b):
    '''
    Returns a + b

    @param a: int
    @param b: int

    @return: int
    '''
    return
        get_component(SAMPLE_APP_FIRST_COMPONENT_ID).add(a,
        b)
```

Note the dynamic nature of a service and how the component is detached from the service. This is a key feature which allows us to use different implementations without touching the service and command layers.

Components are the a bit different, they are not only consits of functions or method. Each component is a class that is inherited from Deltapy Component class. Component classes should be decorated with deltapy 'register' decorator. When application is loading, the Deltapy application loader will walk through out the application tree to find commands and components. First, we create the components package:

```
mkdir sample_application/first_package/components
touch sample_application/first_package/components/__init__.py
```

and we add our class to it[content of `sample_application/first_package/components/__init__.py` module]:

```
from deltapy.application.decorators import register
from deltapy.core import DeltaObject

from sample_application.first_package import
    SAMPLE_APP_FIRST_COMPONENT_ID

@register(SAMPLE_APP_FIRST_COMPONENT_ID)
class
    SampleApplicationFirstPackageComponent(SampleApplicationFirstPackageRealIm
     '''
     Components are the real implementaion you can also
         inherit them from another class for better and
         cleaner code.
     '''


class
    SampleApplicationFirstPackageRealImplementation(DeltaObject):
     '''
     This class contains the implementaion.
     '''
     def add(self, a, b):
         '''
         Returns a + b

         @param a: int
         @param b: int

         @return: int
         '''
         return a + b
```

Note the ‘register’ decorater on top of component class. The Deltapy application loader will add this component to it’s applications component collection. The DeltaObject class is the base class for all classes in Deltapy framework. And finally, see how we detached the real implementation from component itself.

Now our SampleApplicationFirstPackage is completed and the ‘add’ command is exposed and service is working. We can run the command after running and connecting to SampleApplicatin instance.