04 - using AOP

Why use aspect orientated programing (AOP)

The drawback when using dependency injection is that we need to write a lot of configuration code so we need AOP (you can read more here) in order to reduce the amount of code we write for the configurations of the plugins.

How to use it

Lets take as an example the plugin from "using sql alchemy" chapter, we needed to write a function in order to create a service instance for **UserService**, when we are going to have tens and maybe hundreds of such services it might become annoying so the solution is to adopt conventions and use AOP for configuring the plugin.

***__plugin__*.sample_plugin.service**

```
from __plugin__.plugin.registry import addService
from __plugin__.sample_plugin.db_sample import alchemySessionCreator
from ally.container import support
from ally.support.sqlalchemy.session import bindSession

# ----------------------------------------------------------------------

API, IMPL = 'sample_plugin.api.**.I*Service', 'sample_plugin.impl.**.*'

support.createEntitySetup(API, IMPL)

def bindSampleSession(proxy): bindSession(proxy, alchemySessionCreator())
support.listenToEntities(IMPL, listeners=addService(bindSampleSession,))

support.loadAllEntities(API)
```

It might look a little confusing but is actually pretty simple. First we have the variables **API** and **IMPL** they contain the AOP signature for the services, the **API** contains the path where all the service API specification classes are found, for instance the **IUserService** class will be found in this path and also any other API service classes that respect the path, just to put it in words the path can be translated like get all classes that start with **I** and end with **Service** regardless of the module name that are contain in the top package **sample_plugin.api**. The **IMPL** contains the aop path that contains the implementations, also to put this in words is get all classes regardless of the module name that are in the top package **sample_plugin.impl**. So we have the two variables that will point out the classes that are API services and classes that are contained in the implementation package, the next step is to create entity setup functions, for this we have the **createEntitySetup** support function. The function will create entity setup functions that if defined by the user will look like:

```
@ioc.entity
def IUserService() -> IUserService:# The API service interface
    return UserService()# The service implementation that inherits the API service
```

The used implementations are only those that extend at least one API service interface, that is why the aop path **IMPL** for the implementations is so general since the create function will only consider those that inherit an **API**. So now we have a entity setup function automatically generated for any pair of service classes that have an **API** class and also an **IMPL** class that inherits the API, now we need to add the instances provided by this setup functions as services. In order to do that we have the support function **listenToEntities** which listens for instances created that are of the provided classes. The listening is done at the moment of the entity creation so the listen is not dependent of the declared entity return type. The function **addService** from the plugin component actually returns a callable that will be used as the listener, this callable will create a proxy for the event instance and then call on that proxy the **bindSampleSession** function which actually just binds the session creator to the proxy. The final support function invoked is the **loadAllEntities**, this is necessary because the IoC container doesnt invokes an entity setup functions only when is required, in our case we just added listeners to the creation but nobody really tries to invoke the generated entity setup function, so in this case we need to force the entity setup function calls.

You can find the packaged egg here and the sample with the query here.