02 - create a plugin

Create the plugin structure

First we need to create the structure of the plugin as described in "organize a plugin" chapter. We will end up having the packages and modules:

- **plugin #** the setup package
  - *_init_* # the init module of the package
  - **sample_plugin** # the unique configuration package
    - *__init__*
- **sample_plugin** # the main package
  - *__init__*
  - **api** # the API package
    - *__init__*
  - **meta** # the meta package even though we do not use it for this example
    - *__init__*
  - **impl** # the implementation package
    - *__init__*
- build-plugin.ant # the ant build file for the plugin

We have now a clean empty structure for our plugin to be implemented on.

Describing the API

The example will be based on a simple user description. First we need to create the module **user** in package **sample_plugin.api** that will contain:

sample_plugin.api.user

```
'''
Created on Mar 29, 2012

@package: simple plugin sample
@copyright: 2011 Sourcefabric o.p.s.
@license: http://www.gnu.org/licenses/gpl-3.0.txt
@author: Gabriel Nistor

The API descriptions for user sample.
'''

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

class User:
    '''
    The user model.
    '''
    Id = int
    Name = str
```

We have now a simple model class that contains attributes having as a value the intended type for the instance attribute. This class is not yet recognized by the ally framework. First of all if use an IDE you need to add to your python path the "ally-api.1.0.egg" found in GIT repository in folder "distribution/components" this will provide you with type hinting since the IDE knows the resource you require. So in order to make the model class usable we need to decorated like this:

add to python path:	**distribution/components/ally-api.1.0.egg**

sample_plugin.api.user

```
from ally.api.config import model

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

@model(id='Id')
class User:
    '''
    The user model.
    '''
    Id = int
    Name = str
```

All models require to have a single unique id regardless of the primitive type, the id is specified as a keyed argument to the model decorator. We need to also a specific domain in order to avoid confusion of models that have the same name but with different purposes as an example if we have the model **User** with no domain it will be accessible on http://localhost/resources/User in this manner there is no reference to what the user model might be useful for but if we set the domain **Sample** the model is available on http://localhost/resources/Sample/User which is much more suggestive. You can specify domains to individual models like:

sample_plugin.api.user

```
from ally.api.config import model

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

@model(id='Id', domain='Sample')
class User:
    '''
    The user model.
    '''
    Id = int
    Name = str
```

Since other plugins or models might use the same domain in order to avoid repetition we can simply define a domained model decorator in the **sample_plugin.api.*__init__*** module like this:

sample_plugin.api.*__init__*

```
from functools import partial
from ally.api.config import model

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

modelSample = partial(model, domain='Sample')
```

So now the decorated user model will look like this:

sample_plugin.api.user

```
from sample_plugin.api import modelSample

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

@modelSample(id='Id')
class User:
    '''
    The user model.
    '''
    Id = int
    Name = str
```

For more details on the **model** and decorator please see the "API configurations" chapter.
Now that we have a model we require a service that will deliver instances of this model, this instances will be rendered as a REST response. First of all we need to define the service API(interface).

sample_plugin.api.user

```
from ally.api.config import service, call
from ally.api.type import Iter
from sample_plugin.api import modelSample

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

@modelSample(id='Id')
class User:
    '''
    The user model.
    '''
    Id = int
    Name = str

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

@service
class IUserService:
    '''
    The user service.
    '''

    @call
    def getUsers(self) -> Iter(User):
        '''
        Provides all the users.
        '''
```

The service interfaces will have the name starting with "**I**" capital letter than the service name and end with "**Service**". Is important to respect this conventions since the ally IoC setup also offers AOP (aspect orientated programing) features, so respecting the convention will lead to less work in the configuration modules. The service interface needs to be decorated with the **service** decorator in order to make it available for the ally framework, also each method definition that needs to be considered as exposing response models needs to be decorated with **call**. We need to annotate the method with the return type of the method in this case is an iterable collection that contains **User** models. The ally framework uses the annotated return and input types in order to associate a path that will invoke the corresponding service method, in our example the mapped path is http://localhost/resources/Sample/User this is because the return type is a collection of **User** models. All methods that belong to a service need to have the input and return type annotated event if a method is not exposed by using the **call** decorator. The function name is not used by the ally framework in the path construction so any refactoring that changes the method name will not affect the REST client. For more details on the **service** and **call** decorators please see the "API configurations" chapter.

Making a do nothing implementation for the API

Now that we have an API we can make the implementation based on this API. The implementation we will make is just as example and we will not use any database for it. First we need to create the module **user** in package **sample_plugin.impl** that will contain:

sample_plugin.impl.user

```
'''
Created on Mar 29, 2012

@package: simple plugin sample
@copyright: 2011 Sourcefabric o.p.s.
@license: http://www.gnu.org/licenses/gpl-3.0.txt
@author: Gabriel Nistor

Simple implementation for the user APIs.
'''

from sample_plugin.api.user import IUserService

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

class UserService(IUserService):
    '''
    Implementation for @see: IUserService
    '''

    def getUsers(self):
        '''
        @see: IUserService.getUsers
        '''
        return []
```

This implementation returns an empty list whenever a list of users is requested.

Creating the configuration

We have now the api and implementation we need a way now to specify to the ally framework that I wish my implementation to be exposed through HTTP REST. In order to do this we have the dependency injection container, you can find out more details on this in the "DI container" chapter but now will focus on what needs to be done to make this example work. First we need to create the module **service** in package ***__plugin__*.sample_plugin** that will contain:

add to python path:	**distribution/components/ally-utilities.1.0.egg**

__*plugin*__.sample_plugin.service

```
'''
Created on Mar 29, 2012

@package: simple plugin sample
@copyright: 2011 Sourcefabric o.p.s.
@license: http://www.gnu.org/licenses/gpl-3.0.txt
@author: Gabriel Nistor

Contains the services setups.
'''

from sample_plugin.api.user import IUserService
from sample_plugin.impl.user import UserService
from ally.container import ioc

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

@ioc.entity
def userService() -> IUserService:
    b = UserService()
    return b
```

We have defined the setup function that delivers the implementation instance of **UserService** for the **IUserService** api. Because this function is decorated with the **ioc.entity** decorator it will be used as a entity source by the DI container. Now we just need to provide this implementation instance to the exposed services, we need to add the code:

add to python path:	**distribution/components/ally-core-plugin.1.0.egg**

__*plugin*__.sample_plugin.service

```
from __plugin__.plugin.registry import registerService
from ally.container import ioc
from sample_plugin.api.user import IUserService
from sample_plugin.impl.user import UserService

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

@ioc.entity
def userService() -> IUserService:
    b = UserService()
    return b

@ioc.start
def register():
    registerService(userService())
```

The register method will register the user service implementation instance to be used exposed, please notice that the instance is obtained by invoking the DI entity function **userService**.

Deploying and packaging

The plugin is now fully functional but we need to included into the application distribution so it can be deployed. If you are using an IDE and you have imported the distribution as a project you just need to add to the python path the plugin and run the distribution application, anyway this is a development solution we need to package the plugin in order to be able to included in the distribution. In order to package the plugin we have the **build-plugin.ant**

build-plugin.ant

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="create-a-plugin" default="build" basedir=".">

 <property name="destination" value="." />
 <property name="egg" value="02 - plugin sample.1.0.egg" />

 <target name="clean">
  <delete file="${destination}/${egg}" />
 </target>

 <target name="build" depends="clean">
  <zip destfile="${destination}/${egg}" basedir=".">
   <exclude name="**/__pycache__/" />
   <exclude name="/.*" />
   <exclude name="/*.egg" />
  </zip>
 </target>

</project>
```

When running this ant build you will have in the source folder of the plugin the packaged egg file **02 - plugin sample.1.0.egg** you just need to copy this file into your distribution in the plugin folder (**distribution/plugin/**) and now just run the distribution application. We presume that the application is running on the localhost on port 80, by using a browse we will access the resources of the application distribution http://localhost/resources

http://localhost/resources

```
<Resources>
 <SampleUser href="http://localhost/resources/Sample/User/" />
 ...
</Resources>
```

If you see the **SampleUser** entry it means that your plugin deployment was successful. If we now access http://localhost/resources/Sample/User we get an empty user list response that is because we returned an empty list in our service implementation.

You can find the packaged egg here.

Making a dummy implementation for the API

So we managed to present something on the ally REST framework but what if we want to return some **User** entities event if they are generated, in this case we need to change the user implementation:

sample_plugin.impl.user

```
from sample_plugin.api.user import IUserService, User

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

class UserService(IUserService):
    '''
    Implementation for @see: IUserService
    '''

    def getUsers(self):
        '''
        @see: IUserService.getUsers
        '''
        users = []
        for k in range(1, 2):
            user = User()
            user.Id = k
            user.Name = 'User %s' % k
            users.append(user)
        return users
```

This implementation returns a list with one **User** model so now if we access the address http://localhost/resources/Sample/User we will get as a response:

http://localhost/resources/Sample/User

```
<UserList>
 <User>
  <Name>User 1</Name>
  <Id>1</Id>
 </User>
</UserList>
```

If we want more user models we can simply adjust the range. You can find the packaged egg here.

Querying

So now we have a list of users and we need a way of filtering it for this we have the query objects.

sample_plugin.api.user

```
from ally.api.config import service, call, query
from ally.api.criteria import AsLike
from ally.api.type import Iter
from sample_plugin.api import modelSample

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

@modelSample(id='Id')
class User:
    '''
    The user model.
    '''
    Id = int
    Name = str

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

@query
class QUser:
    '''
    The user model query object.
    '''
    name = AsLike

...
```

The query object is like a model but is designed to keep data used for filtering models, by convention a query for a model is named like the model but starts with a **Q**, also the query attributes start with a lower case in order to avoid confusions with the model, the query attributes have as values the criteria class that is applied for the attribute. The user query contains an attribute **name** attribute as a **AsLike** criteria, this criteria allows for filtering the name based on a specified like value and also allows for ordering, you can find out more by reading the documentation on the criteria classes. Now lets see how we can use this query.

sample_plugin.api.user

```
...

@service
class IUserService:
    '''
    The user service.
    '''

    @call
    def getUsers(self, q:QUser=None) -> Iter(User):
        '''
        Provides all the users.
        '''
```

We changed our service **getUsers** API method to take as a parameter a query object instance which is not mandatory because it has a **None** default value, keep in mind that all query objects when are used in a service method need to have a default value specified because queries should not be mandatory. The ally framework knows how to handle this query objects, we will see latter on how we will specify the sorting and like filtering, but first lets see how we are going to use it in the implementation.

sample_plugin.impl.user

```
from sample_plugin.api.user import IUserService, User, QUser
from ally.support.api.util_service import likeAsRegex

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

class UserService(IUserService):
    '''
    Implementation for @see: IUserService
    '''

    def getUsers(self, q=None):
        '''
        @see: IUserService.getUsers
        '''
        users = []
        for k in range(1, 10):
            user = User()
            user.Id = k
            user.Name = 'User %s' % k
            users.append(user)

        if q:
            assert isinstance(q, QUser)
            if QUser.name.like in q:
                nameRegex = likeAsRegex(q.name.like)
                users = [user for user in users if nameRegex.match(user.Name)]
            if QUser.name.ascending in q:
                users.sort(key=lambda user: user.Name, reverse=not q.name.ascending)

        return users
```

We first increased the range of users created to 10, after the users are generated we check to see if there is a query object specified. If there is a query object we check if it has specified in the name criteria the like value, if so we generate a regex based on a like pattern and then filter the users based on the generated regex. After that we check if the ascending flag is specified if so then sort the users list based on that.
Now we need to redeploy the application with the plugin changes, now if we access http://localhost/resources/Sample/User we will get as a response the entire list of 10 users, but if you access http://localhost/resources/Sample/User?name=%7 you will get in the response only the 7th user, in order to sort ascending the users list we call http://localhost/resources/Sample/User?asc=name for descending http://localhost/resources/Sample/User?desc=name. The **asc** and **desc** can take multiple names, if the user query has also a description for a sorting that is made first on the name and then on description it will look like http://localhost/resources/Sample/User?asc=name,description or http://localhost/resources/Sample/User?asc=name&asc=description. You can find this example here.