03 - using sql alchemy

Define the meta

We will use the sample without query made in the "create a plugin" chapter and make the **User** model persisted and interrogated with a database. In order to see how we are going to create tables and mapped them using SQL Alchemy you need to read the documentation. First we need to define a **MetaData** object that will be used for creating the table, the meta is defined in the module **sample_plugin.meta.__*init*__**.

add to python path:	**distribution/libraries/SQLAlchemy-0.7.1-py3.2.egg**

sample_plugin.meta.__*init*__

```
'''
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 package where all the meta modules will be found.
'''

from sqlalchemy.schema import MetaData

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

meta = MetaData()
# Provides the meta object for SQL alchemy.
```

Now we create a a **user** module in package **sample_plugin.meta** in here we will map the **User** model to a database table. The REST models mapping is pretty similar to what SQL alchemy presents except that the mapping between the table and the REST model is done through a different mapping method.

sample_plugin.meta.user

```
'''
Created on Mar 30, 2012

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

Mapping for the user model.
'''

from sqlalchemy.schema import Table, Column
from sqlalchemy.types import String, Integer
from sample_plugin.meta import meta

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

table = Table('sample_user', meta,
              Column('id', Integer, primary_key=True, key='Id'),
              Column('name', String(20), nullable=False, key='Name'))
```

We have defined here a SQL alchemy table called "sample_user" that has two columns, and id column and a name column. The columns that we require to be linked with the REST models need to have the key contain the name of the attribute in the model that is linked with, now we just

need to map this table to the REST model.

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

sample_plugin.meta.user

```
from ally.support.sqlalchemy.mapper import mapperModel
from sample_plugin.api.user import User
from sample_plugin.meta import meta
from sqlalchemy.schema import Table, Column
from sqlalchemy.types import String, Integer

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

# map User entity to defined table (above)
User = mapperModel(User, table)
```

Handling the implementation

We now have a mapped **User** model class that we can easily use in our implementations, keep in mind that you need to use the **User** class from the meta in order to be able to use it with SQL Alchemy. Now we can proceed to the implementation in order to adjust it to get the data from database.

sample_plugin.impl.user

```
from sample_plugin.api.user import IUserService
from ally.support.sqlalchemy.session import SessionSupport

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

class UserService(IUserService, SessionSupport):
    '''
    Implementation for @see: IUserService
    '''
```

First we need to add session support to the service implementation so in order to do this we extend the **SessionSupport**, now we have a session assigned on our service. If you implement the ***__init__*** method do not forget to also call the **SessionSupport.*__init__***. The session is automatically handeled by the ally framework SQL alchemy component so there is no need to begin or end a transaction. The transaction starts when the request is made and ends after the response has been delivered. Now we need to adjust the **getUsers** method to deliver the users from the database.

sample_plugin.impl.user

```
from sample_plugin.api.user import IUserService
from ally.support.sqlalchemy.session import SessionSupport
from sample_plugin.meta.user import User

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

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

    def getUsers(self):
        '''
        @see: IUserService.getUsers
        '''
        return self.session().query(User).all()
```

Notice that we are not using the **User** model from the **sample_plugin.api.user** module we are now using it from **sample_plugin.meta.user** module. As you see is very simple to get all the users from the database this is because the **User** model has been mapped and now SQL

alchemy now how to handle it. Now the problem is that there is no user in the database table, so if we want to add users we need to create an **insert** method. We need first to specify in the **sample_plugin.api.user** the new functionality we require so we need to add the method to the **IUserService** class:

sample_plugin.api.user

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

    ...

    @call
    def insert(self, user:User) -> User.Id:
        '''
        Persist the user model.
        '''
```

The method will handle the insert of the user model. We need to annotate the insert method with the types of the input an output, in this case the input is a **User** model and the output is the **User.Id** based on this annotations the ally framework knows how to handle the method. The models classes and models classes attributes are used as types for defining service methods as you seen also with the **getUser** method. Now lets see how we will implement the insert method.

sample_plugin.impl.user

```
...
from sqlalchemy.exc import SQLAlchemyError
import logging

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

log = logging.getLogger(__name__)

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

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

    ...

    def insert(self, user):
        '''
        @see: IUserService.insert
        '''
        mapped = User()
        if User.Name in user: mapped.Name = user.Name
        try:
            self.session().add(mapped)
            self.session().flush((mapped,))
        except SQLAlchemyError:
            log.exception('Could not insert %s' % user)
        return mapped.Id
```

The first problem is that the user attribute will contain a User model that is from **sample_plugin.api.user** but SQL Alchemy only knows how to handle the **User** model from **sample_plugin.meta.user** basically the mapped model so we need to make this conversion this is why we have:

sample_plugin.impl.user

```
mapped = User()
        if User.Name in user: mapped.Name = user.Name
```

In the second line we actually check is the **User.Name** attribute is specified for the user instance, if specified we will also set it on our mapped object. After we have the mapped **User** object we need to persist it, in order to do this we are adding it to the session, the flush is necessary in order to get an **Id** on our mapped **User** object in order to be able to return it. We also added a log in order to report any problem that might appear while we are inserting the **User** object into the database.

The configurations

We have now the means of adding users in the database and also to fetch them but there is still the problem of configuring the database we need to use. First we need to define the database setup module, so we create the **db_sample** module in the ***__plugin__*.sample_plugin** package.

__*plugin*__.sample_plugin.db_sample

```
'''
Created on Mar 30, 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 database setup for the samples.
'''

from ally.container import ioc
from sample_plugin.meta import meta
from sqlalchemy.engine import create_engine
from sqlalchemy.engine.base import Engine
from sqlalchemy.orm.session import sessionmaker

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

@ioc.config
def database_url():
    '''The database URL for the samples'''
    return 'sqlite:///sample.db'

@ioc.entity
def alchemyEngine() -> Engine:
    engine = create_engine(database_url())
    return engine

@ioc.entity
def alchemySessionCreator():
    return sessionmaker(bind=alchemyEngine())

@ioc.start
def createTables():
    meta.create_all(alchemyEngine())
```

We will take each function step by step in order to see the role of each one:

__*plugin*__.sample_plugin.db_sample

```
@ioc.config
def database_url():
    '''The database URL for the samples'''
    return 'sqlite:///sample.db'
```

This function provides the database URL that will be used by SQL Alchemy to connect to a database. In this case we have provided a SQLite database that will be in the "sample.db" file, when the application will start this file will be created in the distribution folder if it doesn't exist already. You will be able to change this with other databases like MySQL for instance by changing the entry associated with this configuration in the "plugins.properties" file.

__plugin__.sample_plugin.db_sample

```
@ioc.entity
def alchemyEngine() -> Engine:
    engine = create_engine(database_url())
    return engine
```

This entity setup functions create the SQL Alchemy engine this is more of a SQL Alchemy feature, the only thing I want to point out is that in order to get the URL for the database engine we are calling the configuration function **database_url** that returns the URL. This is done because the actual returned string that contains the URL might be changed by configurations in the "plugins.properties" file so is very important to actually place all configurations in configuration setup function in order to allow the IoC container to override the configurations if it might be required.

__plugin__.sample_plugin.db_sample

```
@ioc.entity
def alchemySessionCreator():
    return sessionmaker(bind=alchemyEngine())
```

This will provide the session creator that will be used for creating the sessions whenever a service method is invoked, also the session creator is something from SQL Alchemy that we just take advantage of.

__plugin__.sample_plugin.db_sample

```
@ioc.start
def createTables():
    meta.create_all(alchemyEngine())
```

This start event setup function is used to create the tables in the database. When the application starts all the tables in the meta we used will be created if they do not exist already, so this will ensure us that we will have a **user_sample** table in the database when the application starts. Now we have the database configuration made and we ensured also that the tables are created, now we need to adapt the service to automatically handle the sessions. A session needs to be created using the session creator whenever a method (that belongs to the service API) of the service is invoked, after the method has been invoked the session is closed with a commit if no exception has occur or with a rollback if there was an exception. This is the general view for the session handling but there are some exceptions, if for instance while a service method is processed and another service method is used while processing and that service uses the same session creator (same database) no new session or transaction will be created but instead the same one will be used. So in order to have this session handling we need to wrap the service implementation with a proxy that does that, so lets go back to the **__plugin__.sample_plugin.service** configuration module.

__plugin__.sample_plugin.service

```
from __plugin__.plugin.registry import registerService
from __plugin__.sample_plugin.db_sample import alchemySessionCreator
from ally.container import ioc
from ally.container.proxy import createProxy, ProxyWrapper
from ally.support.sqlalchemy.session import bindSession
from sample_plugin.api.user import IUserService
from sample_plugin.impl.user import UserService

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

@ioc.entity
def userService() -> IUserService:
    b = UserService()
    proxy = createProxy(IUserService)(ProxyWrapper(b))
    bindSession(proxy, alchemySessionCreator())
    return proxy

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

So instead of returning the actual instance of **UserService** implementation we first create a proxy class for the API service interface **IuserService** this proxy class contains all the methods that are defined in the API, then we create an instance of this proxy class that will delegate all the calls to

the actual user service implementation ant this proxy will be the returned instance, but before we return this instance we are going to bind the session handling to all the proxy methods.
Now we have a service that uses a database, just added to the distribution and run the application. After the application has been started you should see the **sample.db** file in the **distribution** folder. If you access http://localhost/resources/Sample/User you will get an empty list as the response since there are no users in the **user_sample** database table. So in order to add a user you need to use a tool that will allow you to make POST requests, I use restclient-ui-2.3.3-jar-with-dependencies.jar but you can use any tool that you are conformable with.

method:	**POST**
Accept:	**xml**
Content-Type:	**xml**

http://localhost/resources/Sample/User

```
<User>
    <Name>Jhon Doe</Name>
</User>
```

After making this post you will receive as a response the id of the newly inserted user.

http://localhost/resources/Sample/User

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<User>
    <Id>1</Id>
</User>
```

You can find the packaged egg here.

Querying

We have seen how to do the simple implementation lets see how we will handle the querying, you just need to use the user API from the query example in the "create a plugin" chapter but keep also the **insert** service method. Because now the users are from the database we cannot know how many users we will have in the response, so in order to avoid huge responses we will introduce the **offset** and **limit** for the users list.

sample_plugin.impl.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

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

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

    @call
    def getUsers(self, offset:int=None, limit:int=10, q:QUser=None) -> Iter(User):
        '''
        Provides all the users.
        '''

    @call
    def insert(self, user:User) -> User.Id:
        '''
        Persist the user model.
        '''
```

We added an offset and limit attribute of type integer in the **getUsers** method. The ally framework knows how to handle free parameters as long as they have a default value and are of a primitive type. Now we need to adjust the implementation.

sample_plugin.impl.user

```
from ally.support.sqlalchemy.session import SessionSupport
from sample_plugin.api.user import IUserService, QUser
from sample_plugin.meta.user import User
from sqlalchemy.exc import SQLAlchemyError
from sqlalchemy.sql.expression import desc
from sqlalchemy.sql.operators import like_op
import logging

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

log = logging.getLogger(__name__)

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

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

    def getUsers(self, offset=None, limit=None, q=None):
        '''
        @see: IUserService.getUsers
        '''
        sql = self.session().query(User)
        if q:
            if QUser.name.like in q:
                sql = sql.filter(like_op(User.Name, q.name.like))
            if QUser.name.ascending in q:
                sql = sql.order_by(User.Name if q.name.ascending else desc(User.Name))
        if offset: sql = sql.offset(offset)
        if limit: sql = sql.limit(limit)
        return sql.all()

    def insert(self, user):
        '''
        @see: IUserService.insert
        '''
        mapped = User()
        if User.Name in user: mapped.Name = user.Name
        try:
            self.session().add(mapped)
            self.session().flush((mapped,))
        except SQLAlchemyError:
            log.exception('Could not insert %s' % user)
        return mapped.Id
```

You will notice that the **getUsers** implementation method has a default value for limit set to **None** instead of 10, the effect of this is that whenever the **getUsers** is called using external requests the API limit of 10 will be used, if is made internal (from a different plugin for example) the **None** limit will apply. Now in order to provide the limit and offset like this http://localhost/resources/Sample/User?offset=1&limit=1. You can find the egg here.