Stalker Documentation

Release 0.2.24

Erkan Ozgur Yilmaz

Jan 01, 2020

Contents

CHAPTER 1

About

Stalker is an Open Source Production Asset Management (ProdAM) Library designed specifically for Animation and VFX Studios but can be used for any kind of projects. Stalker is licensed under LGPL v3.

1.1 Features

Stalker has the following features:

- Designed for **Animation and VFX Studios**.
- Platform independent.
- Default installation handles nearly all the asset and project management needs of an animation and vfx studio.
- Customizable with configuration scripts.
- Customizable object model (**Stalker Object Model - SOM**).
- Uses **TaskJuggler** as the project planing and tracking backend.
- Mainly developed for **PostgreSQL** in mind but **SQLite3** is also supported.
- Can be connected to all the major 3D animation packages like **Maya, Houdini, Nuke, Fusion, Softimage, Blender** etc. and any application that has a Python API. And with applications like **Adobe Photoshop** which does not have a direct Python API but supports `win32com` or `comtypes`.
- Mainly developed for **Python 3.0+** and **Python 2.7** is fully supported.
- Developed with **TDD** practices.

Stalker is build over these other OpenSource projects:

- Python
- SQLAlchemy and Alembic
- Jinja2
- TaskJuggler

Stalker as a library has no graphical UI, it is a python library that gives you the ability to build your pipeline on top of it. There are other python packages like the Open Source Pyramid Web Application Stalker Pyramid and

the Open Source pipeline library Anima which has PyQt/PySide/PySide2 UIs for applications like Maya, Nuke, Houdini, Fusion, Photoshop etc.

1.2 Installation

Use:

```
pip install stalker
```

1.3 Examples

Let's play with **Stalker**.

Initialize the database and fill with some default data:

```
from stalker import db
db.setup()
db.init()
```

Create a `User`:

```
from stalker.db.session import DBSession
from stalker import User
me = User(
    name='Erkan Ozgur Yilmaz',
    login='erkanozgur',
    email='my_email@gmail.com',
    password='secretpass'
)

# Save the user to database
DBSession.save(me)
```

Create a `Repository` for project files to be saved under:

```
from stalker import Repository
repo = Repository(
    name='Commercial Projects Repository',
    windows_path='Z:/Projects',
    linux_path='/mnt/Z/Projects',
    osx_path='/Volumes/Z/Projects'
)
```

Create a `FilenameTemplate` (to be used as file naming convention):

```
from stalker import FilenameTemplate

task_template = FilenameTemplate(
    name='Standard Task Filename Template',
    target_entity_type='Task',  # This is for files saved for Tasks
    path='{{project.repository.path}}/{{project.code}}/'
         '{%- for parent_task in parent_tasks -%}'
         '{{parent_task.nice_name}}/'
         '{%- endfor -%}',  # This is Jinja2 template code
    filename='{{version.nice_name}}_v{{"%03d"|format(version.version_number)}}'
)
```

Create a `Structure` that uses this template:

```
from stalker import Structure
standard_folder_structure = Structure(
    name='Standard Project Folder Structure',
    templates=[task_template],
    custom_template='{{project.code}}/References'  # If you need extra folders
)
```

Now create a `Project` that uses this structure and will be placed under the repository:

```
from stalker import Project
new_project = Project(
    name='Test Project',
    code='TP',
    structure=standard_folder_structure,
    repositories=[repo],  # if you have more than one repository you can do it
)
```

Define the project resolution:

```
from stalker import ImageFormat
hd1080 = ImageFormat(
    name='1080p',
    width=1920,
    height=1080
)
```

Set the project resolution:

```
new_project.image_format = hd1080

# Save the project and all the other data it is connected to it
DBSession.save(new_project)
```

Create Assets, Shots and other Tasks:

```
from stalker import Task, Asset, Shot, Type

# define Character asset type
char_type = Type(name='Character', code='CHAR', target_entity_type='Asset')

character1 = Asset(
    name='Character 1',
    code='CHAR1',
    type=char_type,
    project=new_project
)

# Save the Asset
DBSession.save(character1)

model = Task(
    name='Model',
    parent=character1
)

rigging = Task(
    name='Rig',
    parent=character1,
    depends=[model],  # For project management, define that Rig can not start
                      # before Model ends.
)
```

(continues on next page)

(continued from previous page)

```
# Save the new tasks
DBSession.save([model, rigging])

# A shot and some tasks for it
shot = Shot(
    name='SH001',
    code='SH001',
    project=new_project
)

# Save the Shot
DBSession.save(shot)

animation = Task(
    name='Animation',
    parent=shot,
)

lighting = Task(
    name='Lighting',
    parent=shot,
    depends=[animation], # Lighting can not start before Animation ends,
    schedule_timing=1,
    schedule_unit='d',  # The task expected to take 1 day to complete
    resources=[me]
)
DBSession.save([animation, lighting])
```

Let's create versions for the Animation task.

```
from stalker import Version

new_version = Version(task=animation)
new_version.update_paths()  # to render the naming convention template
new_version.extension = '.ma'  # let's say that we have created under Maya
```

Let's check how the version path is rendered:

```
assert new_version.absolute_full_path == \
    "Z:/Projects/TP/SH001/Animation/SH001_Animation_Main_v001.ma"
assert new_version.version_number == 1
```

Create a new version and check that the version number increased automatically:

```
new_version2 = Version(task=animation)
new_version2.update_paths()  # to render the naming convention template
new_version2.extension = '.ma'  # let's say that we have created under Maya

assert new_version2.version_number == 2
```

See more detailed example in API Tutorial.

CHAPTER 2

Installation

2.1 How to Install Stalker

This document will help you install and run Stalker.

2.2 Install Python

Stalker is completely written with Python, so it requires Python. It currently works with Python version 2.6 and 2.7. So you first need to have Python installed in your system. On Linux and OSX there is a system wide Python already installed. For Windows, you need to download the Python installer suitable for your Windows operating system (32 or 64 bit) from Python.org

2.3 Install Stalker

The easiest way to install the latest version of Stalker along with all its dependencies is to use the *setuptools*. If your system doesn't have setuptools (particularly Windows) you need to install *setuptools* by using *ez_setup* bootstrap script.

2.3.1 Installing *setuptools* with *ez_setup*:

These steps are generally needed just for Windows. Linux and OSX users can skip this part.

1. download ez_setup.py
2. run the following command in the command prompt/shell/terminal:

   ```
   python ez_setup
   ```

   It will install or build the *setuptools* if there are no suitable installer for your operating system.

2.3.2 Installing Stalker (All OSes):

After installing the *setuptools* you can run the following command:

```
easy_install -U stalker
```

Now you have installed Stalker along with all its dependencies.

2.4 Checking the installation of Stalker

If everything went ok you should be able to import and check the version of Stalker by using the Python prompt like this:

```
>>> import stalker
>>> stalker.__version__
0.2.21
```

2.5 For developers

It is highly recommended to create a *VirtualEnv* specific for Stalker development. So to setup a virtualenv for Stalker:

```
virtualenv --no-site-packages stalker
```

Then clone the repository (you need git to do that):

```
cd stalker
git clone https://github.com/eoyilmaz/stalker.git stalker
```

And then to setup the virtual environment for development:

```
cd stalker
../bin/python setup.py develop
```

This command should install any dependent package to the virtual environment.

2.6 Installing a Database

Stalker uses a database to store all the data. The only database backend that doesn't require any extra installation is SQLite3. You can setup Stalker to run with an SQLite3 database. But it is much suitable to have a dedicated database server in your studio. And it is recommended to use the same kind of database backend both in development and production to reduce any compatibility problems and any migration headaches.

Although Stalker is mainly tested and developed on SQLite3, the developers of Stalker are using it in a studio environment where the main database is PosgreSQL, and it is the recommended database for any application based on Stalker. But, testing and using Stalker in any other database is encouraged.

See the SQLAlchemy documentation for supported databases.

CHAPTER 3

API Tutorial

3.1 Introduction

Using Stalker along with Python is all about interacting with a database by using the Stalker Object Model (SOM). Stalker uses the powerful SQLAlchemy ORM.

This tutorial section let you familiarise with the Stalker Python API and Stalker Object Model (SOM). If you used SQLAlchemy before you will feel at home and if you aren't you will see that it is fun dealing with databases with SOM.

3.2 Part I - Basics

Lets say that we just installed Stalker (as you are right now) and want to use Stalker in our first project.

The first thing we are going to learn about is how to connect to the database so we can enter information about our studio and the projects.

We are going to use a helper script to connect to the default database. Use the following command to connect to the database:

```
from stalker import db
db.setup({"sqlalchemy.url": "sqlite:///"})
```

This will create an in-memory SQLite3 database, which is useless other than testing purposes. To be able to get more out of Stalker we should give a proper database information. The most basic setup is to use a file based SQLite3 database:

```
db.setup({"sqlalchemy.url": "sqlite:///C:/studio.db"}) # assumed Windows
```

or:

```
db.setup({"sqlalchemy.url": "sqlite:////home/ozgur/studio.db"}) # under linux or␣
↪osx
```

Note: Although with Stalker v0.2.18 the SQLite3 support is dropped, Stalker can still work with an SQLite3 database. But the suggested database backend is PostgreSQL (preferably PostgreSQL 9.5).

Then if this is the first time you are connecting to the database, then you should initialize the database to create some default data:

```
db.init()
```

This will create some very important default data required for Stalker to work properly. Although it will not break anything to call `db.init()` multiple times it is needed only once (so you don't need to call it again when you close your python shell and open up a new and fresh one).

Lets continue by creating a `Studio` for our self:

```
from stalker import Studio
my_studio = Studio(
    name='My Great Studio'
)
```

For now don't care what a Studio is about. It will be explained later on this tutorial.

Lets continue by creating a **User** for ourselves in the database. The first thing we need to do is to import the `User` class in to the current namespace:

```
from stalker import User
```

then create the `User` object:

```
me = User(
    name="Erkan Ozgur Yilmaz",
    login="eoyilmaz",
    email="some_email_address@gmail.com",
    password="secret",
    description="This is me"
)
```

Now we have just created a user which represents us.

Lets create a new `Department` to define your department:

```
from stalker import Department
tds_department = Department(
    name="TDs",
    description="This is the TDs department"
)
```

Now add your user to the department:

```
tds_department.users.append(me)
```

or we can do it by using the User instance:

```
me.departments.append(tds_department)
```

Even if you didn't do the latter, when you run:

```
print(me.departments)
# you should get something like
# [<TDs (Department)>]
```

We have successfully created a `User` and a `Department` and we assigned the user as one of the member of the **TDs Department**.

Because we didn't tell Stalker to commit the changes, no data has been saved to the database yet. So lets send it the data to the database:

```
from stalker.db.session import DBSession
DBSession.add(my_studio)
DBSession.add(me)
DBSession.add(tds_department)
DBSession.commit()
```

As you see we have used the `DBSession` object to send (commit) the data to the database. These information are stored in the database right now.

Lets try to get something back from the database by querying all the departments, then getting the second one (the first department is always the "admins" which is created by default) and getting its first members name:

```
all_departments = Department.query.all()
print(all_departments)
# This should print something like
# [<admins (Department)>, <TDs (Department)>]
# "admins" department is created by default

admins = all_departments[0]
tds = all_departments[1]

all_users = tds.users  # Department.users is a synonym for Department.members
                       # they are essentially the same attribute
print(all_users[0])
# this should print
# <Erkan Ozgur Yilmaz ('eoyilmaz') (User)>
```

3.3 Part II/A - Creating Simple Data

Lets say that we have this new commercial project coming and you want to start using Stalker with it. So we need to create a `Project` object to hold data about it.

A project instance needs to have a suitable `StatusList` (see status_and_status_lists_toplevel) and a `Repository` instance:

```
# we will reuse the Statuses created by default (in db.init())
from stalker import Status

status_new = Status.query.filter_by(code='NEW').first()
status_wip = Status.query.filter_by(code='WIP').first()
status_cmpl = Status.query.filter_by(code='CMPL').first()
```

Note: When the Stalker database is first initialized (with `db.init()`) a set of `Statuses` for `Tasks`, `Assets`, `Shots`, `Sequences` and `Tickets` are created along with a `StatusList` for each of the data types. Up to this point in the tutorial we have used those Statuses (new, wip and cmpl) that are created by default.

For now we have just created generic statuses. These `Status` instances can be used with any kind of **statusable** objects. The idea behind is to define the statuses only once, and use them in mixtures suitable for different type of objects. So you can define all the possible Statuses for your entities, then you can create a list of them for specific type of objects.

Lets create a `StatusList` suitable for `Project` instances:

```
# a status list which is suitable for Project instances
from stalker import StatusList, Project
```

(continues on next page)

(continued from previous page)

```
project_statuses = StatusList(
    name="Project Status List",
    statuses=[
        status_new,
        status_wip,
        status_cmpl
    ],
    target_entity_type='Project'  # you can also use Project which is the
                                  # class itself
)
```

So we defined a status list which is suitable for `Project` instances. As you see we didn't used all the generic Statuses in our `project_statuses` because for a Project object we thought that these statuses are enough.

And finally, the `Repository`. The Repository (or Repo if you like) is a path in our file server, where we place files and which is visible to all the workstations/render farmers:

```
from stalker import Repository

# and the repository itself
commercial_repo = Repository(
    name="Commercial Repository",
    code="CR"
)
```

New in version 0.2.24: Starting with Stalker version 0.2.24 `Repository` instances have `stalker.models.repository.Repository.code` attribute to help generating universal paths (both across OSes and different installations of Stalker).

`Repository` class will be explained in detail in upcoming sections.

So:

```
new_project = Project(
    name="Fancy Commercial",
    code='FC',
    status_list=project_statuses,
    repositories=[commercial_repo],
)
```

So we have created our project now.

Lets enter more information about this new project:

```
import tzlocal
import datetime
from stalker import ImageFormat

new_project.description = \
"""The commercial is about this fancy product. The
client want us to have a shiny look with their
product bla bla bla..."""

new_project.image_format = ImageFormat(
    name="HD 1080",
    width=1920,
    height=1080
)

new_project.fps = 25
local_tz = tzlocal.get_localzone()
```

(continues on next page)

(continued from previous page)

```
new_project.end = datetime.datetime(2014, 5, 15, tzinfo=local_tz)
new_project.users.append(me)
```

Lets save all the new data to the database:

```
DBSession.add(new_project)
DBSession.commit()
```

As you see, even though we have created multiple objects (new_project, statuses, status lists etc.) we've just added the `new_project` object to the database, but don't worry all the related objects will be added to the database.

Note: Starting with Stalker v0.2.18 all the datetime information needs to have timezone information (we've used the local timezone in the example).

A Project generally is group of `Tasks` that needs to be completed. A `Task` in Stalker is a type of entity where we define the total amount of effort need to be done (or the duration or the length of the task, see `Task` class documentation) to consider that Task as completed. All of the tasks (leaf tasks in fact, coming next) has `resources` which defines the `Users` who need to work on that task and complete it. These are all explained in `Task` class documentation.

For now you just need to now that `Assets`, `Shots` and `Sequences` in Stalker are derived from `Task` and they are in fact other type of Tasks or a specialized version of Tasks.

So lets create a `Sequence`:

```
from stalker import Sequence

seq1 = Sequence(
    name="Sequence 1",
    code="SEQ1",
    project=new_project,
)
```

And a Sequence generally has `Shots`:

```
from stalker import Shot

sh001 = Shot(
    name='SH001',
    code='SH001',
    project=new_project,
    sequences=[seq1]
)
sh002 = Shot(
    code='SH002',
    project=new_project,
    sequences=[seq1]
)
sh003 = Shot(
    code='SH003',
    project=new_project,
    sequences=[seq1]
)
```

send them to the database:

```
DBsession.add_all([sh001, sh002, sh003])
DBsession.commit()
```

Note: Even though, in this tutorial we have created `Shots` with one `Sequence` instance, it is not needed. You can create `Shots` without any `Sequence` instance needed.

For small projects like commercials, you may skip creating a Sequence at all.

For bigger projects, like feature films, it is a very good idea to use Sequences and then group the Shots under them.

But again, a Shot can be connected to multiple sequences, which is useful if your shot, let say, is a kind of flashback and you will use this shot again without changing it at all, then this feature becomes handy.

3.4 Part II/B - Querying, Updating and Deleting Data

So far we just created some simple data. What about updating them. Let say that we created a new shot with wrong info:

```
sh004 = Shot(
    code='SH004',
    project=new_project,
    sequences=[seq1]
)
DBSession.add(sh004)
DBSession.commit()
```

and you figured out that you have created and committed a wrong info and you want to correct it:

```
sh004.code = "SH005"
DBsession.commit()
```

later on lets say you wanted to get the shot back from database:

```
# first find the data
wrong_shot = Shot.query.filter_by(code="SH005").first()

# now update it
wrong_shot.code = "SH004"

# commit the changes to the database
DBsession.commit()
```

and let say that you decided to delete the data:

```
DBsession.delete(wrong_shot)
DBsession.commit()
```

If you don't close your python session, your variable are still going to contain the data but they do not exist in the database anymore:

```
wrong_shot = Shot.query.filter_by(code="SH005").first()
print(wrong_shot)
# should print None
```

for more info about update and delete options (like cascades) in SQLAlchemy please see the SQLAlchemy documentation.

3.5 Part III - Pipeline

Up until now, we skipped a lot of stuff here to take little steps every time. Even tough we have created users, departments, projects, sequences and shots, Stalker still doesn't know much about our studio. For example, it doesn't have any information about the **pipeline** that we are following and what steps we do to complete those shots, thus to complete the project.

In Stalker, pipeline is managed by `Tasks`. So you create Tasks for Shots and then you can create dependencies between tasks.

So lets create a couple of tasks for one of the shots we have created before:

```
from stalker import Task

previs = Task(
    name="Previs",
    parent=sh001
)

matchmove = Task(
    name="Matchmove",
    parent=sh001
)

anim = Task(
    name="Animation",
    parent=sh001
)

lighting = Task(
    name="Lighting",
    parent=sh001
)

comp = Task(
    name="comp",
    parent=sh001
)
```

Now create the dependencies between them:

```
comp.depends = [lighting]
lighting.depends = [anim]
anim.depends = [previs, matchmove]
```

Stalker uses this dependency relation in scheduling these tasks. That is by appending "lighting" task as one of the dependencies of comp, Stalker now know that lighting should be completed to let the resource of the comp task start working. The "Task Scheduling" will be explained in detail later on in this tutorial.

3.6 Part IV - Task & Resource Management

Now we have a couple of Shots with couple of tasks inside it but we didn't assign the tasks to anybody to let them complete this job.

Lets assign all this stuff to our self (for now :)):

```
previs.resources = [me]
previs.schedule_timing = 10
previs.schedule_unit = 'd'
```

(continues on next page)

(continued from previous page)

```
matchmove.resources = [me]
matchmove.schedule_timing = 2
matchmove.schedule_unit = 'd'

anim.resources = [me]
anim.schedule_timing = 5
anim.schedule_unit = 'd'

lighting.resources = [me]
lighting.schedule_timing = 3
lighting.schedule_unit = 'd'

comp.resources = [me]
comp.schedule_timing = 6
comp.schedule_unit = 'h'
```

Now Stalker knows the hierarchy of the tasks and how much effort is needed to complete this tasks. Stalker will use this information to solve the Scheduling problem, and will tell you when to start and complete this tasks.

Lets commit the changes again:

```
DBsession.commit()
```

If you noticed, this time we didn't add anything to the session, cause we have added the `sh001` in a previous commit, and because all the objects are attached to this shot object in some way, all the changes has been tracked and added to the database.

3.7 Part V - Scheduling

In previous sections of this tutorial we have created a `Shot` and then created a couple of `Tasks` to this shot and then assigned our self as the resource of these tasks.

Stalker knows enough about our little project now, but we don't know where to start the project from. That is which task should we start from.

In Stalker, defining the start and end dates of a Task (also of an Asset, Shot and Sequence) is called "Scheduling". Stalker, with the help of TaskJuggler, can solve this problem and define when the resource should work on a specific task.

Warning: You should have TaskJuggler installed in your system, and you should have configured your Stalker installation to be able to find the `tj3` executable.

On a linux system this should be fairly straight forward, just install TaskJuggler and stalker will be able to use it.

But for other OSes, like OSX and Windows, you should create an environment variable called `STALKER_PATH` and then place a file called `config.py` inside the folder that this path is pointing at. And then add the following to this `config.py`:

```
tj_command = 'C:\\Path\\to\\tj3.exe'
```

The default value for `tj_command` config variable is `/usr/local/bin/tj3`, so if on a Linux or OSX system when you run:

```
which tj3
```

is returning this value (`/usr/local/bin/tj3`) you don't need to setup anything.

So, lets schedule our project by using the `Studio` instance that we have created at the beginning of this tutorial:

```
from stalker import TaskJugglerScheduler

my_studio.scheduler = TaskJugglerScheduler()
my_studio.duration = datetime.timedelta(days=365)   # we are setting the
my_studio.schedule(scheduled_by=me)                 # duration to 1 year just
                                                    # to be sure that TJ3
                                                    # will not complain
                                                    # about the project is not
                                                    # fitting in to the time
                                                    # frame.
DBsession.commit()  # to reflect the change
```

This should take a little while depending to your projects size (around 1-2 seconds for this tutorial, but around ~15 min for a project with 15000+ tasks).

When it is finished all of your tasks now have their `computed_start` and `computed_end` values filled with proper data. Now check the start and end values:

```
print(previs.computed_start)      # 2014-04-02 16:00:00
print(previs.computed_end)        # 2014-04-15 15:00:00

print(matchmove.computed_start)   # 2014-04-15 15:00:00
print(matchmove.computed_end)     # 2014-04-17 13:00:00

print(anim.computed_start)        # 2014-04-17 13:00:00
print(anim.computed_end)          # 2014-04-23 17:00:00

print(lighting.computed_start)    # 2014-04-23 17:00:00
print(lighting.computed_end)      # 2014-04-24 11:00:00

print(comp.computed_start)        # 2014-04-24 11:00:00
print(comp.computed_end)          # 2014-04-24 17:00:00
```

The dates are probably going to be different in your computer. But as you see Stalker has computed the start and end date values for each of the tasks. They are simply following one other, this is because we have entered only one resource for each of the task.

You should know that "Scheduling" is a huge concept and it is greatly explained in TaskJuggler documentation.

For a last thing you can check the `to_tjp` values of each data we have created for now, so try running:

```
print(my_studio.to_tjp)
print(me.to_tjp)
print(comp.to_tjp)
print(new_project.to_tjp)
```

If you are familiar with TaskJuggler, you should recognize the output of each `to_tjp` variable. So essentially Stalker is mapping all of its data to a TaskJuggler compatible string. A very small part of TaskJuggler directives are currently supported. But it is enough to schedule very complex projects with complex dependency relation and Task hierarchies. And with every new version of Stalker the supported TaskJuggler directives are expanded.

3.8 Part VI - Asset Management

Now we have created a lot of things but other then storing all the data in the database, we didn't do much. Stalker still doesn't have information about a lot of things. For example, it doesn't know how to handle your asset versions (`Version`) namely it doesn't know how to store your data that you are going to create while completing these tasks.

So what we need to define is a place in our file structure. It doesn't need to be a network shared directory but if you are not working alone than it means that everyone needs to reach your data and the simplest way to do this is

to place your files in a network share, there are other alternatives like storing your files locally and sharing your revisions with a Software Configuration Management (SCM) system, Stalker doesn't support the latter right now.

We are going to see the first alternative, which uses a network share in our fileserver, and this network share is called a `Repository` in Stalker.

A repository is a file path, preferably a path which is mapped or mounted to the same path on every computer in your studio (also you can use `autofs` with an OpenLDAP server in which you can synchronize all off the mount points on all of your workstations and render slaves at once).

In Stalker, you can have several repositories, let say one for Commercials and another one for each big Movie projects.

You can define repositories and assign projects to those repositories.

We have already created a repository while creating our first project. But the repository has missing information. A Repository object shows the path that we create our projects into. Lets enter the paths for all the major operating systems:

```
commercial_repo.linux_path   = "/mnt/M/commercials"
commercial_repo.osx_path     = "/Volumes/M/commercials"
commercial_repo.windows_path = "M:/commercials"  # you can use reverse
                                                 # slashes (\\) if you want
```

And if you ask for the path to a repository object it will always give the correct answer according to your operating system:

```
print(commercial_repo.path)
# under Windows outputs:
# M:/commercials
#
# in Linux and variants:
# /mnt/M/commercials
#
# and in OSX:
# /Volumes/M/commercials
```

Note: Stalker always uses forward slashes no matter what operating system you are using. It is like that even if you define your paths with reverse slashes (\).

Assigning this repository to our project is not enough, Stalker still doesn't know about the directory structure of this project. To explain the project structure to Stalker we use a `Structure` instance:

```
from stalker import Structure

commercial_project_structure = Structure(
    name="Commercial Projects Structure"
)

# now assign this structure to our project
new_project.structure = commercial_project_structure
```

New in version 0.2.13: Starting with Stalker version 0.2.13 `Project` instances can have **multiple** `Repository` instances attached. So you can create complex templates where you can for example store published versions on a different server/network share or you can setup so the outputs of a version (like the rendered files) are stored on a different server, and etc.

The following examples are updated in a simple way and examples showing the advantage of having multiple repositories will be added on later versions.

Now we have created a very simple structure instance, but we still need to create `FilenameTemplate` instances for Tasks which then will be used by the `Version` instances to generate a consistent and meaningful path and filename:

```
from stalker import FilenameTemplate

task_template = FilenameTemplate(
    name='Task Template for Commercials',
    target_entity_type='Task',
    path='$REPO{{project.repository.id}}/{{project.code}}/{%- for p in parent_
↪tasks -%}{{p.nice_name}}/{%- endfor -%}',
    filename='{{version.nice_name}}_v{{"%03d"|format(version.version_number)}}'
)

# and append it to our project structure
commercial_project_structure.templates.append(task_template)

# commit to database
DBsession.commit()  # no need to add anything, project is already on db
```

By defining a `FilenameTemplate` instance we have essentially told Stalker how to store `Version` instances created for `Task` entities in our `Repository`.

The data entered both to the `path` and `filename` arguments are Jinja2 directives. The `Version` class knows how to render these templates while calculating its `path` and `filename` attributes.

Also, if you noticed we have used an environment variable "$REPO" along with the id of the first repository in the project "{{project.repository.id}}" (attention! `project.repository` always shows the first repository in the project), this is a new feature introduced with Stalker version 0.2.13. Stalker creates environment variables on runtime for each of the repository whenever a repository is created and inserted in to the DB or it will create environment variables for already existing repositories upon a successful database connection.

Lets create a `Version` instance for one of our tasks:

```
from stalker import Version

vers1 = Version(
    task=comp
)

# we need to update the paths
vers1.update_paths()

# check the path and filename
print(vers1.path)                # '$REPO33/FC/SH001/comp'
print(vers1.filename)            # 'SH001_comp_Main_v001'
print(vers1.full_path)           # '$REPO33/FC/SH001/comp/SH001_comp_Main_v001'

# now the absolute values, values with repository root
# because I'm running this code in a Linux laptop, my results are using the
# linux path of the repository
print(vers1.absolute_path)       # '/mnt/M/commercials/FC/SH001/comp'
print(vers1.absolute_full_path)  # '/mnt/M/commercials/FC/SH001/comp/SH001_comp_
↪Main_v001'

# check the version_number
print(vers1.version_number)      # 1

# commit to database
DBsession.commit()
```

As you see, the `Version` instance magically knows where to place itself and what to use as the filename. Thanks to Stalker it is now easy to create version files where you don't have weird file names (ex: 'Shot1_comp_Final', 'Shot1_comp_Final_revised', 'Shot1_comp_Final_revised_Final', 'Shot1_comp_Final_revised_Final_real_final' and the list goes on, we all know those filenames don't we :)).

With Stalker the filename and path always follows strict rules.

Also by using the `Version.is_published` attribute you can define which of the versions are usable and which are versions that you are still working on:

```
vers1.is_published = False  # I still work on this version, this is not a
                            # usable one
```

Lets create another version for the same task and see what happens:

```
# be sure that you've committed the previous version to the database
# to let Stalker now what number to give for the next version
vers2 = Version(task=comp)
vers2.update_paths()  # this call probably will disappear in next version of
                      # Stalker, so Stalker will automatically update the
                      # paths on Version.__init__()

print(vers2.version_number)  # 2
print(vers2.filename)        # 'SH001_comp_Main_v002'

# before creating a new version commit this one to db
DBsession.commit()

# now create a new version
vers3 = Version(task=comp)
vers3.update_paths()

print(vers3.version_number)  # 3
print(vers3.filename)        # 'SH001_comp_Main_v002'
```

Isn't that nice, Stalker increments the version number automatically.

Also you can query all the versions of a specific task by:

```
# using pure Python
vers_from_python = comp.versions  # [<FC_SH001_comp_Main_v001 (Version)>,
                                  #  <FC_SH001_comp_Main_v002 (Version)>,
                                  #  <FC_SH001_comp_Main_v003 (Version)>]

# or using a query
vers_from_query = Version.query.filter_by(task=comp).all()

# again returns
# [<FC_SH001_comp_Main_v001 (Version)>,
#  <FC_SH001_comp_Main_v002 (Version)>,
#  <FC_SH001_comp_Main_v003 (Version)>]

assert vers_from_python == vers_from_query
```

Note: Stalker stores `Version.path` and `Version.filename` attributes in the database, so the values does not contain any OS specific path. It will only show the OS specific path on `Version.absolute_path` and on `Version.absolute_full_path` attributes by joining the `Repository.path` with the path values from database momentarily.

You can also setup your project structure to have default directories:

```
commercial_project_structure.custom_template = """
Temp
References
References/Movies
References/Images
"""
```

When the above template is executed each line will refer to a directory.

3.9 Part VII - Collaboration (not completed)

We came a lot from the start, but what is the use of an Production Asset Management System if we can not communicate with our colleagues.

In Stalker you can communicate with others in the system, by:

- Leaving a `Note` to anything created in Stalker (except you can not create a `Note` to another `Note` and to a `Tag`).
- Sending a `Message` directly to them or to a group of users. (Not implemented yet).
- Anyone can create a `Ticket` for a `Project`.
- You can create wiki `Pages` per `Project`.

3.10 Part VIII - Extending SOM (coming)

This part will be covered soon

3.11 Conclusion

In this tutorial, you have nearly learned a quarter of what Stalker supplies as a Python library.

Stalker is a very flexible and powerful Production Asset Management system. As of writing this tutorial it has been developed for the last 5 years (4 years with the only developer being yours truly and for another 1 year where his wife is also attended to the project) and it is currently been used in production of a feature movie.

But it is only a Python library so it doesn't supply any graphical user interface.

There are other projects, namely Stalker Pyramid and Anima that is using Stalker in their back ends. Stalker Pyramid is an Pyramid based Web application and Anima is a pipeline library.

You can clone their repositories to see how PyQt4 and PySide UIs are created with Stalker (in Anima) and how it is used as the database model for a Web application in Stalker Pyramid.

CHAPTER 4

Design

The design of Stalker is mentioned in the following sections.

4.1 Introduction

Stalker is an Open Source Production Asset Management Library. Although it is designed VFX and Animation studios in mind, its flexible Project Management muscles will allow it to be used in a wide variety of fields.

An Asset Management Systems' duty is to hold the data which are created by the users of the system in an organised manner, and let them quickly reach and find their files. A Production Asset Management Systems' duty is, in addition to the asset management systems', also handle the production steps or tasks and allow the users of the system to collaborate. If more information about this subject is needed, there are great books about Digital Asset Management (DAM) Systems.

The usage of an asset management system in an animation/vfx studio is a must for the sake of the studio itself. Even the benefits of the system becomes silly to be mentioned when compared to the lack of even a simple system to organise stuff.

Every studio outside establishes and develops their own asset management system. Stalker will try to be the framework that these proprietary asset management systems will be build over. Thus reducing the work repeated on every big projects' start.

4.2 Concepts

There are a couple of design concepts those needs to be clarified before any further explanation of Stalker.

Stalker on itself basically is the **Model** in an **MTV** system (where the Stalker Pyramid is the *Template* and *View*). So it defines the data and the interaction of the data with itself.

Because the idea behind Stalker was to build an open source library that any studio using it can build their own pipeline on top of it, it is designed to stay simple and solid at the same time. So the UI and other stuff is ripped off from the original Stalker package and moved to another Pyramid web application called Stalker Pyramid.

4.2.1 Stalker Object Model (SOM)

Stalker has a very robust object model, which is called **Stalker Object Model** or **SOM**. The idea behind SOM is to create a class hierarchy which is both usable right out of the box and also expandable by the studios' developers. SOM is actually a little bit more complex than a basic possible model, it is designed in this way just to be able to create a simple pipeline to be able to build the system over it.

Lets look at how a simple studio works and try to create our asset management concepts around it.

An animation/vfx studios duty is to complete a `Project`. A project, generally is about to create a `Sequence` of `Shots` which are a series of images those at the end converts to a movie. So a sequence in general contains Shots. And Shots can use `Assets`. So basically to complete a project the studio should complete the shots and assets needed by those shots.

Furthermore all the Projects, Sequences, Shots or Assets are divided in to different `Tasks` those need to be done sequentially or in parallel to complete that project.

A Task relates to a work, a work is a quantity of time spent or going to be spend for that specific task. The time spent on the course of completion of a Task can be recorded with `TimeLogs`. TimeLogs show the total time spent by an artist for a certain Task. So it holds information about how much **effort** has been spent to complete a Task.

During the completion of the Task or at the end of the work a **User** creates `Versions` for that particular Task. Versions are the different incarnations or the progress of the resultant product, and it is connected to files in the fileserver or in Stalkers term the `Repository`.

All the names those shown in bold fonts are a class in SOM. and there are a series of other classes to accommodate the needs of a `Studio`.

The inheritance diagram of the classes in the SOM is shown below:

4.3 Inheritance Diagram

Stalker is a configurable and expandable and most importantly it is an open source system. All of these features allows the system to have a flexible structure.

There are two levels of expansion, the first level is the simplest one, by just adding different statuses, different types or these kind of things in which Stalker's current design is ready to. This is explained in *How To Customize Stalker*.

The second level of expansion is achieved by expanding the SOM. Expanding the SOM includes creating new classes and database tables, and updating the old ones which are already coming with Stalker. These expansion schemes are further explained in *How To Extend SOM*.

4.3.1 Features

1. Developed purely in Python (2.6 and over) using TDD (Test Driven Development) practices
2. SQLAlchemy for the database back-end and ORM
3. Uses Jinja2 as the template system for the file and folder naming convention, it is possible to use templates like:

   {repository.path}/{project.code}/Assets/{asset.type.name}/{asset.code}/ {asset.name}_{asset.type.name}_v{version.version_number}.{version.extension}
4. File and folders and file sequences can be uploaded to the server as assets, and the server decides where to place the folder or file by using the template system.
5. The event system gives full control for every CRUDL (create/insert, read, update, delete and list) by giving step like before insert, after insert call-backs.
6. The messaging system allows the users collaborate efficiently.
7. Has an embedded Ticket system.

8. Uses TaskJuggler as the task management backend and supports basic Task attributes.
9. Has a predefined workflow for task statuses called Task Status Workflow which manages the statuses of a Task during the project completion.

For usage examples see *API Tutorial*.

4.4 How To Customize Stalker

This part explains the customization of Stalker.

4.5 How To Extend SOM

This part explains how to extend Stalker Object Model or SOM.

4.6 Creating Data

There are some examples here, to create simple data.

4.6.1 Creating a Project

To create a Project, we need:

1. A Repository
2. A Structure object to define the file structure of the Project:
3. FilenameTemplates for Task, Asset, Shot, Sequence types, to define the placement of the Versions created for them.
4. An ImageFormat to define the output size of the project.
5. A StatusList with enough Statuses that will define the desired Project Statuses. Stalker doesn't have a Project Status Workflow, yet! so define yours.
6. If desired we can also add a Type for the Project to distinguish commercials from Feature Film projects.
7. We need to create a user as the lead for the project.

Here is the code:

```
from stalker import (db, Repository, Structure, FilenameTemplate, StatusList,
                     Status, Task, User)

# first setup the database connection (assuming that you have a config.py
# defined, so we do not need to supply a database address)
db.setup()

# initialize the database just for the first time
db.init()  # run this only for the first time, subsequent runs will not
           # create any errors, but it is unnecessary

# re-use Statuses NEW, WIP and CMPL from default statuses
status_new = Status.query.filter_by(code='NEW').first()
status_wip = Status.query.filter_by(code='WIP').first()
status_cmpl = Status.query.filter_by(code='CMPL').first()

# and create a new one
```

(continues on next page)

(continued from previous page)

```
status_on_air = Status(name='On Air', code='OA')

# status list for project
project_status_list = StatusList(
    name='Project Statuses',
    target_entity_type='Project',
    statuses=[
        status_new,
        status_wip,
        status_cmpl,
        status_on_air
    ],
)

image_format_hd = ImageFormat(
    name="HD",
    width=1920,
    height=1080,
)

commercial_type = Type(
  name='Commercial',
  code='COMM',
  target_entity_type='Project'
)

repo = Repository(
    name='Commercials Repo',
    linux_path='/mnt/T/Commercials/',
    windows_path='T:/Commercials/',
    osx_path='/Volumes/T/Commercials/'
)

commercial_structure = Structure(
    name='Commercial Project Structure',
    code=''
)

lead = User(
    name='Erkan Ozgur Yilmaz',
    login='eoyilmaz',
    email='eoyilmaz@stalker.com',
    password='secret'
)

# lets create the Project
proj1 = Project(
    name='Test Project',
    code='TP',
    description="This is the first project",
    lead=lead,
    image_format=image_format_hd,
    fps=25,
    type=commercial_type,
    structure=commercial_structure,
    repository=repo,
    status_list=project_status_list,
    status=status_new
)

# just add the project to the database
```

(continues on next page)

(continued from previous page)

```
from stalker.db.session import DBSession
DBSession.add(proj1)

# and commit the data to database
DBSession.commit()
```

It may seem too much for just creating a Project, but it is for the first time only. For a second project, we can use the previous Repository, Structure, Lead, StatusList etc.

4.6.2 Create a Task

Because we have a project now lets create a task for this project:

```
# connect to the database if you have not done yet
db.setup()

# create a new user as the resource for the task
resource1 = User(
    name='User1',
    login='user1',
    email='user@users.com',
    password='secret'
)

# now create the task
task1 = Task(
    name='Task1',
    description="This is our first Task, and it is about, creating "
                "something fancy",
    resources=[resource1],
    schedule_timing=1,
    schedule_unit='d',
    schedule_model='effort',
    project=proj1
)
# we do not need to supply a StatusList for the Task, statuses for tasks are
# created by default when we called db.init() in previous example

# add it to the database
DBSession.add(task1)

# and commit
DBSession.commit()
```

Now we have created a simple Task and assigned it to the resource1. Lets check the status of the Task:

```
print(task1.status)
# this should print something like <Ready To Start (RTS) (Status)>
# stating that our task is ready to start working on.
```

CHAPTER 5

Configuring Stalker

To configure Stalker and make it fit to your Studios need you should use the `config.py` file as mentioned in next sections.

5.1 config.py File

Stalker uses the `config.py` to let one to customize the system config.

The `config.py` file is searched in a couple of places through the system:

- under "~/.strc/" directory (not yet)
- under "$STALKER_PATH"

The first path is a folder in the users home dir. The second one is a path defined by the `STALKER_PATH` environment variable.

Defining the `config.py` by using the environment variable gives the most customizable and consistent setup through the studio. You can set `STALKER_PATH` to a shared folder in your fileserver where all the users can access.

Because, `config.py` is a regular Python code which is executed by Stalker, you can do anything you were doing in a normal Python script. This is very handy (also dangerous!) if you have another source of information which is reachable by a Python script.

If there is no `STALKER_PATH` variable in your current environment or it is not showing an existing path or there is no `config.py` file the system will use the system defaults.

5.2 Config Variables

Variables which can be set in `config.py` are as follows:

`actions`
Actions for authorization system. These are used to create ACLs. Stalker uses CRUDL system. Default value is:

```
actions = ['Create', 'Read', 'Update', 'Delete', 'List'] #CRUDL
```

auto_create_admin
Tells Stalker to create an admin by default. Default value is:

```
auto_create_admin = True
```

admin_name
The default admin user name. Default value is:

```
admin_name = 'admin'
```

admin_login
The default admin login. Default value is:

```
admin_login = 'admin'
```

admin_password
The default admin password. Default value is:

```
admin_password = 'admin'
```

admin_email
The default email for admin user. Default value is:

```
admin_email = 'admin@admin.com'
```

admin_department_name
The default department name for admin. Default value is:

```
admin_department_name = 'admins'
```

admin_group_name
The default admin permission group name. Default value is:

```
admin_group_name = 'admins'
```

database_engine_settings
A dictionary of config values. The default value is:

```
database_engine_settings = {
    "sqlalchemy.url": "sqlite:///:memory:",
    "sqlalchemy.echo": False,
}
```

database_session_settings
This value is not used.

local_storage_path
The local storage path for Stalker.

local_storage_path = os.path.expanduser('~/.strc')

local_session_data_file_name
The per user or local session file name. It is used for storing logged in user info. The default value is:

```
local_session_data_file_name = 'local_session_data'
```

server_side_storage_path
Storage for uploaded files. This used by Stalker Pyramid and shows the server side storage path. Will be moved to Stalker Pyramid in later versions. Not used by Stalker by default. Default value is:

```
server_side_storage_path = os.path.expanduser('~/Stalker_Storage')
```

key
The default keyword which is going to be used in password scrambling. Default value is:

```
key = "stalker_default_key"
```

version_take_name
The default take name for `Version` instances. Default value is:

```
version_take_name = "Main"
```

status_bg_color
Default background color for `Status` instances. Default value is:

```
status_bg_color = 0xffffff
```

status_fg_color
Default foreground color for `Status` instances. Default value is:

```
status_fg_color = 0x000000
```

ticket_label
Default ticket label. Used by `Ticket` when generating a ticket name. Default value is:

```
ticket_label = "Ticket"
```

ticket_status_order
Defines the ticket statuses and the order of them. Default value is:

```
ticket_status_order = [
    'new', 'accepted', 'assigned', 'reopened', 'closed'
]
```

ticket_resolutions
Defines the default ticket resolutions. Default value is:

```
ticket_resolutions = [
    'fixed', 'invalid', 'wontfix', 'duplicate', 'worksforme', 'cantfix'
]
```

ticket_workflow
Defines the default ticket workflow. It is a dictionary of actions. Shows the new status per action. Default value is:

```
ticket_workflow = {
    'resolve' : {
        'new': {
            'new_status': 'closed',
            'action': 'set_resolution'
        },
        'accepted': {
            'new_status': 'closed',
            'action': 'set_resolution'
        },
        'assigned': {
            'new_status': 'closed',
            'action': 'set_resolution'
        },
        'reopened': {
            'new_status': 'closed',
            'action': 'set_resolution'
        },
```

(continues on next page)

(continued from previous page)

```
    },
    'accept' : {
        'new': {
            'new_status': 'accepted',
            'action': 'set_owner'
        },
        'accepted': {
            'new_status': 'accepted',
            'action': 'set_owner'
        },
        'assigned': {
            'new_status': 'accepted',
            'action': 'set_owner'
        },
        'reopened': {
            'new_status': 'accepted',
            'action': 'set_owner'
        },
    },
    'reassign': {
        'new': {
            'new_status': 'assigned',
            'action': 'set_owner'
        },
        'accepted': {
            'new_status': 'assigned',
            'action': 'set_owner'
        },
        'assigned': {
            'new_status': 'assigned',
            'action': 'set_owner'
        },
        'reopened': {
            'new_status': 'assigned',
            'action': 'set_owner'
        },
    },
    'reopen': {
        'closed': {
            'new_status': 'reopened',
            'action': 'del_resolution'
        }
    }
}
```

timing_resolution

Defines the default timing resolution for classes which are mixed with `DateRangeMixin`. Stalker uses the TaskJuggler default timing resolution which is 1 hour:

```
timing_resolution = datetime.timedelta(hours=1)
```

task_duration

Defines the default task duration. If only a start or end value is entered for a `Task` then Stalker calculates the other value by adding or subtracting the default task duration value from it. Default value is 1 hour:

```
task_duration = datetime.timedelta(hours=1)
```

task_priority

Defines the default task priority. This is used by TaskJuggler to prioritize tasks. Should be a number between 0 and 1000. Default value is 500:

```
task_priority = 500
```

working_hours
Defines the default weekly working hours per week day. Stalker uses the TaskJuggler default value of 9am to 6pm. The values entered are minutes from midnight, and it is a list of lists of two integers. Each list of two integers shows a working hour interval. Default value is:

```
working_hours = {
  'mon': [[540, 1080]], # 9:00 - 18:00
  'tue': [[540, 1080]], # 9:00 - 18:00
  'wed': [[540, 1080]], # 9:00 - 18:00
  'thu': [[540, 1080]], # 9:00 - 18:00
  'fri': [[540, 1080]], # 9:00 - 18:00
  'sat': [], # saturday off
  'sun': [], # sunday off
}
```

daily_working_hours
Defines the default daily working hour. This is strongly related with the `working_hours`, `weekly_working_hours`, `weekly_working_days` and `yearly_working_days` settings and shows a mean value of daily working hour. Default value is 9:

```
daily_working_hours = 9
```

weekly_working_hours
Defines the default weekly working hour. This is strongly related with the `working_hours`, `daily_working_hours`, `weekly_working_days` and `yearly_working_days` settings. Default value is 45:

```
weekly_working_hours = 45
```

weekly_working_days
Defines the default weekly working days. This is strongly related with the `working_hours`, `daily_working_hours`, `weekly_working_hours` and `yearly_working_days` settings. Default value is 5:

```
weekly_working_days = 5
```

yearly_working_days
Defines the default yearly working days. This is strongly related with the `working_hours`, `daily_working_hours`, `weekly_working_hours` and `weekly_working_days` settings. Default value is 260.714 which equals `weekly_working_days` * 52.1428:

```
yearly_working_days = 260.714
```

day_order
Defines the order of the week days. Default value uses European system:

```
day_order = ['mon', 'tue', 'wed', 'thu', 'fri', 'sat', 'sun']
```

datetime_units
Defines the date and time units. The order should match the `datetime_unit_names` setting. Default value is:

```
datetime_units = ['min', 'h', 'd', 'w', 'm', 'y']
```

datetime_unit_names
Defines the names of date and time units. The order should match the `datetime_units` setting. Default value is:

```
datetime_unit_names = ['minute', 'hour', 'day', 'week', 'month', 'year']
```

datetime_units_to_timedelta_kwargs

Defines the conversion ratios of each date and time unit. Default value is:

```
datetime_units_to_timedelta_kwargs = {
    'min': {'name': 'minutes', 'multiplier': 1},
    'h'  : {'name': 'hours'  , 'multiplier': 1},
    'd'  : {'name': 'days'   , 'multiplier': 1},
    'w'  : {'name': 'weeks'  , 'multiplier': 1},
    'm'  : {'name': 'days'   , 'multiplier': 30},
    'y'  : {'name': 'days'   , 'multiplier': 365}
}
```

task_schedule_models

Defines the default schedule models. These are highly related with TaskJuggler, so anything entered here should exist in TaskJuggler. Default value is:

```
task_schedule_models = ['effort', 'length', 'duration']
```

task_schedule_constraints

Defines the default schedule constraints. The order also defines a binary number corresponding to each value (00: none, 01: start, 10:end, 11:both) and used in defining which side of a Task is constrained to a date. Also used by TaskJuggler to constrain the start or end or both dates of a task to a certain date. Also a Task with schedule_constraint is set to 2 (both) is considered a **duration** task even if its schedule_model is set to **effort** or **length**. Default value is:

```
task_schedule_constraints = ['none', 'start', 'end', 'both']
```

tjp_working_hours_template

Defines a Jinja2 template for converting `WorkingHours` instances to a TaskJuggler compatible string. By default Stalker converts a WorkingHours instance to a `workinghours` statement in TaskJuggler. Default value is:

```
tjp_working_hours_template = """{% macro wh(wh, day) -%}
{%- if wh[day]|length %}    workinghours {{day}} {% for part in wh[day] -%}
        {%- if loop.index != 1%}, {% endif -%}
        {{"%02d"|format(part[0]//60)}}:{{"%02d"|format(part[0]%60)}} - {{"%02d
↪"|format(part[1]//60)}}:{{"%02d"|format(part[1]%60)}}
        {%- endfor -%}
{%- else %}    workinghours {{day}} off
{%- endif -%}
{%- endmacro -%}
{{wh(workinghours, 'mon')}}
{{wh(workinghours, 'tue')}}
{{wh(workinghours, 'wed')}}
{{wh(workinghours, 'thu')}}
{{wh(workinghours, 'fri')}}
{{wh(workinghours, 'sat')}}
{{wh(workinghours, 'sun')}}"""
```

tjp_studio_template

Defines a Jinja2 template for converting a `Studio` instance to a TaskJuggler compatible string. By default Stalker converts a Studio instance to a `project` statement in TaskJuggler. Default value is:

```
tjp_studio_template = """project {{ studio.tjp_id }} "{{ studio.name }}" {{␣
↪studio.start.date() }} - {{ studio.end.date() }} {
    timingresolution {{ '%i'|format((studio.timing_resolution.days * 86400 +␣
↪studio.timing_resolution.seconds)//60|int) }}min
    now {{ studio.now.strftime('%Y-%m-%d-%H:%M') }}
    dailyworkinghours {{ studio.daily_working_hours }}
```

(continues on next page)

(continued from previous page)

```
    weekstartsmonday
{{ studio.working_hours.to_tjp }}
    timeformat "%Y-%m-%d"
    scenario plan "Plan"
    trackingscenario plan
}
"""
```

tjp_project_template

Defines a Jinja2 template for converting a `Project` instance to a TaskJuggler compatible string. By default Stalker converts a Project instance to a `task` statement in TaskJuggler. Default value is:

```
tjp_project_template = """task {{project.tjp_id}} "{{project.name}}" {
    {% for task in project.root_tasks %}
        {{task.to_tjp}}
    {% endfor %}
}
"""
```

tjp_task_template

Defines a Jinja2 template for converting a `Task` instance to a TaskJuggler compatible string. By default Stalker converts a Task to a `task` statement in TaskJuggler. Default value is:

```
tjp_task_template = """task {{task.tjp_id}} "{{task.name}}" {
{% if task.priority != 500 -%}priority {{task.priority}}{%- endif %}
{%- if task.depends %}
    depends {% for depends in task.depends %}
    {%- if loop.index != 1 %}, {% endif %}{{depends.tjp_abs_id}}
{%- endfor -%}
{%- endif -%}
{%- if task.is_container -%}
    {%- for child_task in task.children %}
        {{ child_task.to_tjp }}
    {%- endfor %}
{%- else %}
    {% if task.resources|length -%}
    {% if task.schedule_constraint %}
        {%- if task.schedule_constraint == 1 or task.schedule_constraint == 3 -
↪%}
            start {{ task.start.strftime('%Y-%m-%d-%H:%M') }}
        {%- endif %}
        {%- if task.schedule_constraint == 2 or task.schedule_constraint == 3
↪%}
            end {{ task.end.strftime('%Y-%m-%d-%H:%M') }}
        {%- endif -%}
    {% endif %}
    {{task.schedule_model}} {{task.schedule_timing}}{{task.schedule_unit}}
    allocate {% for resource in task.resources -%}
        {%-if loop.index != 1 %}, {% endif %}{{resource.tjp_id}}{% endfor %}
    {%- endif -%}
    {% for time_log in task.time_logs %}
    booking {{time_log.resource.tjp_id}} {{time_log.start.strftime('%Y-%m-%d-
↪%H:%M:%S')}} +{{'%i'|format(time_log.duration.days*24 + time_log.duration.
↪seconds/3600)}}h { overtime 2 }
    {%- endfor -%}
{% endif %}
}
"""
```

tjp_department_template

Defines a Jinja2 template for converting a `Department` instance to a TaskJuggler compatible string. By

default Stalker converts a Department to a `resource` statement in TaskJuggler. Default value is:

```
tjp_department_template = '''resource {{department.tjp_id}} "{{department.name}
↪}" {
{%- for resource in department.users %}
    {{resource.to_tjp}}
{%- endfor %}
}'''
```

tjp_vacation_template

Defines a Jinja2 template for converting a `Vacation` instance to a TaskJuggler compatible string. By default Stalker converts a Vacation instance to a `vacation` statement in TaskJuggler. Default value is:

```
tjp_vacation_template = '''vacation {{ vacation.start.strftime('%Y-%m-%d-%H:%M
↪') }}, {{ vacation.end.strftime('%Y-%m-%d-%H:%M') }}'''
```

tjp_user_template

Defines a Jinja2 template for converting a `User` instance to a TaskJuggler `resource` statement. Default value is:

```
tjp_user_template = '''resource {{user.tjp_id}} "{{user.name}}"{% if user.
↪vacations %} {
    {% for vacation in user.vacations -%}
        {{vacation.to_tjp}}
    {% endfor -%}
}{% endif %}'''
```

tjp_main_template

Defines a Jinja2 template for converting all the information coming from Stalker to a TaskJuggler compatible `tjp` file. Default value is:

```
tjp_main_template = """# Generated By Stalker v{{stalker.__version__}}
{{studio.to_tjp}}

# resources
resource resources "Resources" {
{%- for user in studio.users %}
    {{user.to_tjp}}
{%- endfor %}
}

# tasks
{% for project in studio.active_projects %}
    {{project.to_tjp}}
{% endfor %}

# reports
taskreport breakdown "{{csv_file_full_path}}"{
    formats csv
    timeformat "%Y-%m-%d-%H:%M"
    columns id, start, end
}
"""
```

tj_command

Defines the TaskJuggler command. Stalker uses this configuration value to call TaskJugglers `tj3` command.

tj_command = '/usr/local/bin/tj3',

path_template

Defines a default value for path template for `FilenameTemplate` instances to be used by `Version` instances. This value is not used yet. Default value is:

```
path_template = '{{project.code}}/{%- for parent_task in parent_tasks -%}{
↪{parent_task.nice_name}}/{%- endfor -%}'
```

filename_template

Defines a default value for filename template for `FilenameTemplate` instances to be used by `Version` instances. This value is not used yet. Default value is:

```
filename_template = '{{task.entity_type}}_{{task.id}}_{{version.take_name}}_v{{
↪"%03d"|format(version.version_number)}}'
```

sequence_format

Defines the default file sequence format to be used with PySeq. This value is not used yet. Default value is:

```
sequence_format = "%h%p%t %R"
```

Fore details about the format see the PySeq documentation.

file_size_format

Defines the default file size format to be used in UI. Default value is:

```
file_size_format = "%.2f MB"
```

date_time_format

Defines the default datetime format to be used in UI and string representations of datetime.datetime instances. Default value is:

```
date_time_format = '%Y.%m.%d %H:%M'
```

resolution_presets

Defines default resolution presets. This value is not used yet. Default value is:

```
resolution_presets = {
    "PC Video": [640, 480, 1.0],
    "NTSC": [720, 486, 0.91],
    "NTSC 16:9": [720, 486, 1.21],
    "PAL": [720, 576, 1.067],
    "PAL 16:9": [720, 576, 1.46],
    "HD 720": [1280, 720, 1.0],
    "HD 1080": [1920, 1080, 1.0],
    "1K Super 35": [1024, 778, 1.0],
    "2K Super 35": [2048, 1556, 1.0],
    "4K Super 35": [4096, 3112, 1.0],
    "A4 Portrait": [2480, 3508, 1.0],
    "A4 Landscape": [3508, 2480, 1.0],
    "A3 Portrait": [3508, 4960, 1.0],
    "A3 Landscape": [4960, 3508, 1.0],
    "A2 Portrait": [4960, 7016, 1.0],
    "A2 Landscape": [7016, 4960, 1.0],
    "50x70cm Poster Portrait": [5905, 8268, 1.0],
    "50x70cm Poster Landscape": [8268, 5905, 1.0],
    "70x100cm Poster Portrait": [8268, 11810, 1.0],
    "70x100cm Poster Landscape": [11810, 8268, 1.0],
    "1k Square": [1024, 1024, 1.0],
    "2k Square": [2048, 2048, 1.0],
    "3k Square": [3072, 3072, 1.0],
    "4k Square": [4096, 4096, 1.0],
}
```

default_resolution_preset

Defines the default resolution preset fro new Projects. This value is not used yet. Default value is:

```
default_resolution_preset = "HD 1080"
```

project_structure

Defines the default project structure. This value is not used by Stalker. Default value is:

```
project_structure = """{% for shot in project.shots %}
        Shots/{{shot.code}}
        Shots/{{shot.code}}/Plate
        Shots/{{shot.code}}/Reference
        Shots/{{shot.code}}/Texture
    {% endfor %}
{% for asset in project.assets%}
    {% set asset_path = project.full_path + '/Assets/' + asset.type.name + '/'␣
↪+ asset.code %}
    {{asset_path}}/Texture
    {{asset_path}}/Reference
{% endfor %}
"""
```

thumbnail_format

Defines the default thumbnail format. This value is not used by Stalker. Default value is:

```
thumbnail_format = "jpg"
```

thumbnail_quality

Defines the default thumbnail quality. This value is not used by Stalker. Default value is:

```
thumbnail_quality = 70
```

thumbnail_size

Defines the defaul thumbnail size. This value is not used by Stalker. Default value is:

```
thumbnail_size = [320, 180]
```

CHAPTER 6

Upgrading Database

6.1 Introduction

From time to time, with new releases of Stalker, your Stalker database may need to be upgraded. This is done with the Alembic library, which is a database migration library for SQLAlchemy.

6.2 Instructions

The upgrade is easy, just run the following command on the root of the stalker installation directory:

```
# for Windows
..\Scripts\alembic.exe upgrade head

# for Linux or OSX
../bin/alembic upgrade head

# this should output something like that:
#
# INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
# INFO  [alembic.runtime.migration] Will assume transactional DDL.
# INFO  [alembic.runtime.migration] Running upgrade 745b210e6907 -> f2005d1fbadc,␣
↪added ProjectClients
```

That's it, your database is now migrated to the latest version.

CHAPTER 7

How To Contribute

Stalker started as an Open Source project with the expectation of contributions. The soul of the open source is to share the knowledge and contribute.

These are the areas that you can contribute to:

- Documentation
- Testing the code
- Writing the code
- Creating user interface elements (graphics, icons etc.)

7.1 Development Style

Stalker is developed strictly by following TDD practices. So every participant should follow TDD methodology. Skipping this steps is highly prohibited. Every added code to the trunk should have a corresponding test and the tests should be written before implementing a single line of code.

DRY is also another methodology that a participant should follow. So nothing should be repeated. If something needs to be repeated, then it is a good sign that this part needs to be in a special module, class or function.

7.2 Testing

As stated above all the code written should have a corresponding test.

Adding new features should start with design sketches. These sketches could be plain text files or mind maps or anything that can express the thing in you mind. While writing down these sketches, it should be kept in mind that these files also could be used to generate the documentation of the system. So writing down the sketches as rest files inside the docs is something very meaningful.

The design should be followed by the tests. And the test should be followed by the implementation, and the implementation should be followed by tests again, until you are confident about your code and it is rock solid. Then the refactoring phase can start, and because you have enough tests that will keep your code doing a certain thing, you can freely change your code, because you know that you code will do the same thing if it passes all the tests.

The first tests written should always fail by having:

```
self.fail("the test is not implemented yet")
```

failures. This is something good to have. This will inform us that the test is not written yet. After blocking all the tests and you are confident about the tests are covering all the aspects of your design sketches, you can start writing the tests.

Another very important note about the tests are the docstrings of the test methods. You should explain what is this test method testing, and what you expect as a result of the test. It

After finishing implementing the tests you can start adding the code that will pass the tests.

The test framework of Stalker is unitTest and nose to help testing.

These python modules should be installed to test Stalker properly:

- Nose
- Coverage

The coverage of the tests should be kept as close as possible to %100.

There is a helper script in the root of the project, called *doTests*. This is a shell script for linux, which runs all the necessary tests and prints the tests results and the coverage table.

Note: From version 0.1.1 the use of Mocker library is discontinued. The tests are done using real objects. It is done in this way cause the design of the objects were changing too quickly, and it started to be a guess work to see which of the tests are effected by this changes. So the Mocker is removed and it will not be used in future releases.

7.3 Coding Style

For the general coding style every participant should strictly follow PEP 8 rules, and there are some extra rules as listed below:

- Class names should start with an upper-case letter, function and method names should start with lower-case letter:

```
class MyClass(object):
    """the doc string of the class
    """

    def __init__(self):
        pass

    def my_method(self):
        pass
```

- There should be 1 spaces before and after functions and class methods:

```
class StatusBase(object):
    """The StatusBase class
    """

    def __init__(self, name, abbreviation, thumbnail=None):
        self._name = self._checkName(name)

    def _checkName(self, name):
        """checks the name attribute
        """

        if name == "" or not isinstance(name, str):
```

(continues on next page)

(continued from previous page)

```
            raise(ValueError("the name shouldn't be empty and it should \
                be a str"))

            return name.title()
```

- And also there should be 1 spaces before and after a class body:

```
#-*- coding: utf-8 -*-

class A(object):
    pass

class B(object):
    pass

pass
```

- Any lines that may contain a code or comment can not be longer than 79 characters, all the longer lines should be cancelled with "\" character and should continue properly from the line below:

```
def _checkName(self, name):
    """checks the name attribute
    """

    if name == "" or not isinstance(name, str):
        raise(ValueError("the name shouldn't be empty and it should be a \
        str"))

    return name.title()
```

  This rule is not followed for the first line of the docstrings and in long function or method names (particularly in tests).

- If anything is going to be checked against being None you should do it in this way:

```
if a is None:
    pass
```

- Do not add docstrings to __init__ rather use the classes' own docstring.
- The first line in the docstring should be a brief summary separated from the rest by a blank line.

If you are going to add a new python file (*.py), use the following line in the first line:

```
#-*- coding: utf-8 -*-
```

7.4 SCM - Git

The choice of SCM is Git. Every developer should be familiar with it. It is a good start to go the Git Web Site and do the tutorial if you don't feel familiar enough with hg.

7.5 Adding Changes

Stalker is hosted in GitHub.

If you want to do changes in Stalker, the basic pipeline is as follows:

- Fork Stalker from GitHub project page.

- Clone your own Stalker repository to your own computer.
- Do your addition, run your tests, and be sure that your part doesn't have any errors or failures.
- Commit your changes.
- Before creating a pull request check if your repository is in sync with the upstream GitHub repository (the repository that you've forked Stalker from) by using the tools supplied in your GitHub project page.
- In case there are new changes in upstream, merge them with yours.
- Do the tests again. If there are problems in your part of the code, solve the errors/failures.
- Commit your changes again.
- And push them to your own GitHub repository.
- And in the original GitHub page create a Pull Request.

CHAPTER 8

Stalker Development Roadmap

This section describes the direction Stalker is going.

8.1 Roadmap Based on Versions

Below you can find the roadmap based on the version

8.1.1 0.1.0:

- A complete working set of models in SOM which are using SQLAlchemy.ext.declarative.

8.1.2 0.2.0:

- Web interface
- Complete ProdAM capabilities.

8.1.3 0.3.0:

- Complete working Event system

CHAPTER 9

Stalker Changes

9.1 0.2.24

- **New:** `Repository` instances now have a `code` attribute which is used for generating the environment variables where in previous versions the `id` attribute has been used which caused difficulties in transferring the data to a different installation of Stalker. Also to make the system backwards compatible, Stalker will still set the old `id` based environment variables. But when asked for an environment variable it will return the `code` based one. The `code` argument as usual has to be initialized on `Repository` instance creation. That's why this version is slightly backwards incompatible and needs the database to be updated with Alembic (with the command `alembic update head`).
- **Fix:** `Repository` methods `is_in_repo` and `find_repo` are now case insensitive for Windows paths.
- **Update:** Updated `Project` class documentation and included information about what is going to be deleted or how the delete operation will be cascaded when a `Project` instance is deleted.

9.2 0.2.23

- **Update:** Updated the `setup.py` to require `psycopg2-binary` instead of `psycopg2`. Also updated the configuration files for Docker and Travis. This changes the requirement of psycopg2 to psycopg2-binary, which will make it easier to get the installation to complete on e.g. CentOS 7 without requiring pg_config.

9.3 0.2.22

- **Fix:** Fixed `TaskJugglerScheduler.schedule()` method to correctly decode byte data from `sys.stderr` to string for Python 3.x.
- **Fix:** Fixed a couple of tests for TaskJuggler.
- **Update:** Updated Classifiers information in `setup.py`, removed Python versions 2.6, 3.0, 3.1 and 3.2 from supported Python versions.
- **Fix:** Removed Python 3.3 from TravisCI build which is not supported by `pytest` apparently.
- **Update:** Updated TravisCI config and removed Python 2.6 and added Python 3.6.
- **Update:** Added a test case for an edge usage of FilenameTemplate.

- **Update:** Updated .gitignore file to ignore PyTest cache folder.
- **Update:** Updated the License file to correctly reflect the project license of LGPLv3.
- **Update:** Update copyright information.
- **New:** Created `make_html.bat` for Windows.
- **New:** Added support for Python wheel.

9.4 0.2.21

- **New:** Switched from `nose + unittest` to `pytest` as the main testing framework (with `pytest-xdist` tests complete 4x faster).
- **New:** Added `DBSession.save()` shortcut method for convenience which does an `add` or `add_all` (depending to the input) followed by a `commit` at once.
- **Update:** Updated the about page for a more appealing introduction to the library.
- **New:** Stalker now creates default `StatusList` for `Project` instances on database initialization.
- **Update:** SQLite3 support is back. In fact it was newer gone. For simplicity of first time users the default database is again SQLite3. It was dropped for the sake of adding more PostgreSQL oriented features. But then it is recognized that the system can handle both. Though a two new Variant had to be created for JSON and Datetime columns.
- **Update:** With the reintroduction of SQLite3, the new JSON type column in `WorkingHours` class has been upgraded to support SQLite3. So with SQLite3 the column stores the data as TEXT but seamlessly convert them to JSON when ORM loads or commits the data.
- **New:** Added `ConfigBase` as a base class for `Config` to let it be used in other config classes.
- **Fix:** Fixed `testing.create_db()` and `testing.drop_db()` to fallback to `subprocess.check_call` method for Python 2.6.
- **Fix:** Fixed `stalker.models.auth.User._validate_password()` method to work with Python 2.6.
- **Update:** Updated all of the tests to use `pytest` style assertions to support Python 2.6 along with 2.7 and 3.0+.
- **Fix:** Fixed `stalker.db.check_alembic_version()` function to invalidate the connection, so it is not possible to continue with the current session, preventing users to ignore the raised `ValueError` when the `alembic_version` of the database is not matching the `alembic_version` of Stalker's current version.

9.5 0.2.20

- **New:** Added `goods` attribute to the `Client` class. To allow special priced `Goods` to be created for individual clients.
- **Fix:** The `WorkingHours` class is now derived from `Entity` thus it is not stored in a `PickleType` column in `Studio` anymore. (issue: #44)
- **Update:** Updated `appveyor.yml` to match `travis.yml`.

9.6 0.2.19

- **Update:** Updated the `stalker.config.Config.database_engine_settings` to point the test database.

- **Fix:** Fixed a bug in `stalker.testing.UnitTestDBBase.setUp()` where it was not considering the existence of the `STALKER_PATH` environment variable while doing the tests.
- **Update:** Removed debug message from `db.setup()` which was revealing the database password.
- **Update:** Updated the `UnitTestDBBase`, it now creates its own test database, which allows all the tests to run in an individual database. Thus, the tests can now be run in `multiprocess` mode which speeds things a lot.
- **Fix:** Removed any module level imports of `stalker.defaults` variable, which can be changed by a Studio (or by tests) and should always be refreshed.
- **Update:** Removed the module level import of the `stalker.db.session.DBSession` in `stalker.db`, so it is not possible to use `db.DBSession` anymore.
- **Update:** The import statements that imports `stalker.defaults` moved to local scopes to allow runtime changes to the `defaults` to be reflected correctly.
- **Update:** Added Python fall back mode to `stalker.shot.Shot._check_code_availability()` which runs when there is no database.
- **Update:** `stalker.models.task.TimeLog._validate_task()` is now getting the `Status` instances from the `StatusList` that is attached to the `Task` instance instead of doing a database query.
- **Update:** `stalker.models.task.TimeLog._validate_resource()` is now falling back to a Python implementation if there is no database connection.
- **Update:** `stalker.models.task.Task._total_logged_seconds_getter()` is now hundreds of times faster when there is a lot of `TimeLog` instances attached to the `Task`.
- **Update:** In `stalker.models.task.Task` class, methods those were doing a database query to get the required `Status` instances are now using the attached `StatusList` instance to get them.
- **Fix:** A possible `auto_flush` is prevented in `Ticket` class.
- **Update:** `Version.latest_version` property is now able to fall back to a pure Python implementation when there is no database connection.
- **Update:** The default log level has been increased from `DEBUG` to `INFO`.
- **Update:** In an attempt to speed up tests, a lot of tests that doesn't need an active Database has been updated to use the regular `unittest.TestCase` instead of `stalker.testing.TestBase` and as a result running all of the tests are now 2x faster.
- **Fix:** `TimeLogs` are now correctly reflected in UTC in a tj3 file.
- **Fix:** Fixed a lot of tests which were raising Warnings and surprisingly considered as Errors in TravisCI.
- **Fix:** `to_tjp` methods of SOM classes that is printing a Datetime object are now printing the dates in UTC.
- **Fix:** Fixed `stalker.models.auth.Permission` to be hashable for Python 3.
- **Fix:** Fixed `stalker.models.auth.AuthenticationLog` to be sortable for Python 3.
- **Fix:** Fixed `stalker.models.version.Version.latest_version` property for Python 3.
- **Fix:** Fixed tests of `Permission` class to check for correct exception messages in Python 3.
- **Update:** Replaced the `assertEquals` and `assertNotEquals` calls which are deprecated in Python 3 with `assertEqual` and `assertNotEquals` calls respectively.
- **Fix:** Fixed tests for `User` and `Version` classes to not to cause the `id column is None` warnings of SQLAlchemy to be emitted.

9.7 0.2.18

- **Update:** Support for DB backends other than Postgresql has been dropped. This is done to greatly benefit from a code that is highly optimized only for one DB backend. With This all of the tests should be inherited

from the `stalker.tests.UnitTestDBBase` class.

- **New:** All the DateTime fields in Stalker are now TimeZone aware and Stalker stores the DateTime values in UTC. Naive datetime values are not supported anymore. You should use a library like `pytz` to supply timezone information as shown below:

```
import datetime
import pytz
from stalker import db, SimpleEntity
new_simple_entity = SimpleEntity(
    name='New Simple Entity',
    date_created = datetime.datetime.now(tzinfo=pytz.utc)
)
```

- **Fix:** The default values for `date_created` and `date_updated` has now been properly set to a partial function that returns the current time.
- **Fix:** Previously it was possible to enter two TimeLogs for the same resource in the same datetime range by committing the data from two different sessions simultaneously. Thus the database was not aware that it should prevent that. Now with the new PostgreSQL only implementation and the `ExcludeConstraint` of PostgreSQL an `IntegrityError` is raised by the database backend when something like that happens.
- **Update:** All the tests those are checking the system against an Exception is being raised or not are now checking also the exception message.
- **Update:** In the `TimeLog` class, the raised `OverBookedException` message has now been made clear by adding the start and end date values of the clashing TimeLog instance.
- **Update:** Removed the unnecessary `computed_start` and `computed_end` columns from `Task` class, which are already defined in the `DateRangeMixin` which is a super for the Task class.

9.8 0.2.17.6

- **Fix:** Fixed a bug in `ProjectMixin` where a proper cascade was not defined and the `Delete` operations to the `Projects` table were not cascaded to the mixed-in classes properly.

9.9 0.2.17.5

- **Fix:** Fixed the `image_format` attribute implementation in `Shot` class. Now it will not copy the value of `Project.image_format` directly on `__init__` but instead will only store the value if the `image_format` argument in `__init__` or `Shot.image_format` attribute is set to something.

9.10 0.2.17.4

- **Update:** Updated the comment sections of all of the source files to correctly show that Stalker is LGPL v3 (not v2.1).

9.11 0.2.17.3

- **New:** Added `Shot.fps` attribute to hold the fps information per shot.
- **Update:** Added the necessary alembic revision to reflect the changes in the `Version_Inputs` table.

9.12 0.2.17.2

- **Fix:** Fixed `Version_Inputs` table to correctly take care of `DELETE``s on the ``Versions` table. So now it is possible to delete a `Version` instance without first cleaning the `Link` instances that is related to that `Version` instance.
- **Update:** Changed the `id` attribute name from `info_id` to `log_id` in `AuthenticationLog` class.
- **Update:** Started moving towards PostgreSQL only implementation. Merged the `DatabaseModelTester` class and `DatabaseModelsPostgreSQLTester` class.
- **Fix:** Fixed an autoflush issue in `stalker.models.review.Review.finalize_review_set()`.

9.13 0.2.17.1

- **Fix:** Fixed alembic revision

9.14 0.2.17

- **New:** Added `AuthenticationLog` class to hold user login/logout info.
- **New:** Added `stalker.testing` module to simplify testing setup.

9.15 0.2.16.4

- **Fix:** Fixed alembic revision.

9.16 0.2.16.3

- **New:** `ProjectUser` now also holds a new field called `rate`. The default value is equal to the `ProjectUser.user.rate`. It is a way to hold the rate of a user on a specific project.
- **New:** Added the `Invoice` class.
- **New:** Added the `Payment` class.
- **New:** Added two simple mixins `AmountMixin` and `UnitMixin`.
- **Update:** `Good` class is now mixed in with the new `UnitMixin` class.
- **Update:** `BudgetEntry` class is now mixed in with the new `AmountMixin` and `UnitMixin` classes.

9.17 0.2.16.2

- **New:** `Group` permissions can now be set on `__init__()` with the `permissions` argument.

9.18 0.2.16.1

- **Fix:** As usual after a new release that changes database schema, fixed the corresponding Alembic revision (92257ba439e1).

9.19 0.2.16

- **New:** `Budget` instances are now statusable.
- **Update:** Updated documentation to include database migration instructions with Alembic.

9.20 0.2.15.2

- **Fix:** Fixed a typo in the error message in `User._validate_email_format()` method.
- **Fix:** Fixed a query-invoked auto-flush problem in `Task.update_parent_statuses()` method.

9.21 0.2.15.1

- **Fix:** Fixed alembic revision (f2005d1fbadc), it will now drop any existing constraints before re-creating them. And the downgrade function will not remove the constraints.

9.22 0.2.15

- **New:** `db.setup()` now checks for `alembic_version` before setting up a connection to the database and raises a `ValueError` if the database alembic version is not matching the current implementation of Stalker.
- **Fix:** `db.init()` sets the `created_by` and `updated_by` attributes to `admin` user if there is one while creating entity statuses.
- **New:** Created `create_sdist.cmd` and `upload_to_pypi.cmd` for Windows.
- **New:** `Project` to `Client` relation is now a many-to-many relation, thus it is possible to set multiple Clients for each project with each client having their own roles in a specific project.
- **Update:** `ScheduleMixin.schedule_timing` attribute is now Nullable.
- **Update:** `ScheduleMixin.schedule_unit` attribute is now Nullable.

9.23 0.2.14

- **Fix:** Fixed `Task.path` to always return a path with forward slashes.
- **New:** Introducing `EntityGroups` that lets one to group a bunch of

  ‘‘SimpleEntity‘‘s together, it can be used in grouping tasks even if they are in different places on the project task hierarchy or even in different projects.
- **Update:** `Task.percent_complete` is now correctly calculated for a `Duration` based task by using the `Task.start` and `Task.end` attribute values.
- **Fix:** Fixed `stalker.models.task.update_time_log_task_parents_for_end()` event to work with SQLAlchemy v1.0.
- **New:** Added an option called `__dag_cascade__` to the `DAGMixin` to control cascades on mixed in class. The default value is “all, delete”. Change it to “save-update, merge” if you don’t want the children also be deleted when the parent is deleted.
- **Fix:** Fixed a bug in `Version` class that occurs when a version instance that is a parent of other version instances is deleted, the child versions are also deleted (fixed through DAGMixin class).

9.24 0.2.13.3

- **Fix:** Fixed a bug in `Review.finalize_review_set()` for tasks that are sent to review and still have some extra time were not clamped to their total logged seconds when the review set is all approved.

9.25 0.2.13.2

- **New:** Removed `msrp`, `cost` and `unit` arguments from `BudgetEntry.__init__()` and added a new `good` argument to get all of the data from the related `Good` instance. But the `msrp`, `cost` and `unit` attributes of `BudgetEntry` class are still there to store the values that may not correlate with the related `Good` in future.

9.26 0.2.13.1

- **Fix:** Fixed a bug in `Review.finalize_review_set()` which causes Task instances to not to get any status update if the revised task is a second degree dependee to that particular task.

9.27 0.2.13

- **New:** `Project` instances can now have multiple repositories. Thus the `repository` attribute is renamed to `repositories`. And the order of the items in the `repositories` attribute is restored correctly.
- **New:** `stalker.db.init()` now automatically creates environment variables for each repository in the database.
- **New:** Added a new `after_insert` which listens `Repository` instance ‘‘insert‘‘s to automatically add environment variables for the newly inserted repositories.
- **Update:** `Repository.make_relative()` now handles paths with environment variables.
- **Fix:** Fixed `TaskJugglerScheduler` to correctly generate task absolute paths for PostgreSQL DB.
- **New:** `Repository.path` is now writable and sets the correct path (`linux_path`, `windows_path`, or `osx_path`) according to the current system.
- **New:** Setting either of the `Repository.path`, `Repository.linux_path`, `Repository.windows_path`, `Repository.osx_path` attributes will update the related environment variable if the system and attribute are matching to each other, setting the `linux_path` on Linux or setting the `windows_path` on Windows or setting the `osx_path` on OSX will update the environment variable.
- **New:** Added `Task.good` attribute to easily connect tasks to ‘‘Good‘‘s.
- **New:** Added new methods to `Repository` to help managing paths:
  - `Repository.find_repo()` to find a repo from a given path. This is a class method so it can be directly used with the Repository class.
  - `Repository.to_os_independent_path()` to convert the given path to a OS independent path which uses environment variables. Again this is a class method too so it can be directly used with the Repository class.
  - `Repository.env_var` a new property that returns the related environment variable name of a repo instance. This is an instance property:

    ```
    # with default settings
    ```

    repo = Repository(...) repo.env_var # should print something like “REPO131” which will be used

    # in paths as “$REPO131”

- **Fix:** Fixed `User.company_role` attribute which is a relationship to the `ClienUser` to cascade `all, delete-orphan` to prevent AssertionErrors when a Client instance is removed from the `User.companies` collection.

9.28 0.2.12.1

- **Update:** `Version` class is now mixed with the `DAGMixin`, so all the parent/child relation is coming from the DAGMixin.
- **Update:** `DAGMixin.walk_hierarchy()` is updated to walk the hierarchy in `Depth First` mode by default (method=0) instead of `Breadth First` mode (method=1).
- **Fix:** Fixed `alembic_revision` on database initialization.

9.29 0.2.12

- **Fix:** Fixed importing of `ProjectUser` directly from `stalker` namespace.
- **Fix:** Fixed importing of `ClientUser` directly from `stalker` namespace.
- **New:** Added two new columns to the `BudgetEntry` class to allow more detailed info to be hold.
- **New:** Added a new Mixin called `DAGMixin` to create parent/child relation between mixed in class.
- **Update:** The `Task` class is now mixed with the `DAGMixin`, so all the parent/child relation is coming from the DAGMixin.
- **New:** Added a new class called `Good` to hold details about the commercial items/services sold in the Studio.
- **New:** Added a new class called `PriceList` to create price lists from Goods.

9.30 0.2.11

- **New:** User instances now have a new attribute called `rate` to track their cost as a resource.
- **New:** Added two new classes called `Budget` and `BudgetEntry` to record Project budgets in a simple way.
- **New:** Added a new class called **Role** to manage user roles in different Departments, Clients and Projects.
- **New:** User and Department relation is updated to include the role of the user in that department in a more flexible way by using the newly introduced Role class and some association proxy tricks.
- **New:** Also updated the User to Project relation to include the role of the user in that Project by using an associated Role class.
- **Update:** Department.members attribute is renamed to **users** (and removed the *synonym* property).
- **Update:** Removed `Project.lead` attribute use `Role` instead.
- **Update:** Removed `Department.lead` attribute use `Role` instead.
- **Update:** Because the `Project.lead` attribute is removed, it is now possible to have tasks with no responsible.
- **Update:** Client to User relation is updated to use an association proxy which makes it possible to set a Role for each User for each Client it is assigned to.
- **Update:** Renamed User.company to User.companies as the relation is now able to handle more than one Client instances for the User company.

- **Update:** Task Status Workflow has been updated to convert the status of a DREV task to HREV instead of WIP when the dependent tasks has been set to CMPL. Also the timing of the task is expanded by the value of `stalker.defaults.timing_resolution` if it doesn't have any effort left (generally true for CMPL tasks) to allow the resource to review and decide if he/she needs more time to do any update on the task and also give a chance of setting the Task status to WIP by creating a time log.
- **New:** It is now possible to schedule only a desired set of projects by passing a **projects** argument to the TaskJugglerScheduler.
- **New:** Task.request_review() and Review.finalize() will not cap the timing of the task until it is approved and also Review.finalize() will extend the timing of the task if the total timing of the given revisions are not fitting in to the left timing.

9.31 0.2.10.5

- **Update:** TaskJuggler output is now written to debug output once per line.

9.32 0.2.10.4

- **New:** '@' character is now allowed in Entity nice name.

9.33 0.2.10.3

- **New:** '@' character is now allowed in Version take names.

9.34 0.2.10.2

- **Fix:** Fixed a bug in `stalker.models.schedulers.TaskJugglerScheduler._create_tjp_file_content()` caused by non-ascii task names.
- **Fix:** Removed the residual `RootFactory` class reference from documentation.
- **New:** Added to new functions called `utc_to_local` and `local_to_utc` for UTC to Local time and vice versa conversion.

9.35 0.2.10.1

- **Fix:** Fixed a bug where for a WIP Task with no time logs (apparently something went wrong) and no dependencies using `Task.update_status_with_dependent_statuses()` will convert the status to RTS.

9.36 0.2.10

- **New:** It is now possible to track the Edit information per Shot using the newly introduced `source_in`, `source_out` and `record_in` along with existent `cut_in` and `cut_out` attributes.

9.37 0.2.9.2

- **Fix:** Fixed MySQL initialization problem in `stalker.db.init()`.

9.38 0.2.9.1

- **New:** As usual, after a new release, fixed a bug in `stalker.db.create_entity_statuses()` caused by the behavioral change of the `map` built-in function in Python 3.

9.39 0.2.9

- **New:** Added a new class called `Daily` which will help managing `Version` outputs (Link instances including Versions itself) as a group.
- **New:** Added a new status list for `Daily` class which contains two statuses called "Open" and "Closed".
- **Update:** Setting the `Version.take_name` to a value other than a string will now raise a `TypeError`.

9.40 0.2.8.4

- **Fix:** Fixed `SimpleEntity._validate_name()` method for unicode strings.

9.41 0.2.8.3

- **Fix:** Fixed str/unicode errors due to the code written for Python3 compatibility.
- **Update:** Removed `Task.is_complete` attribute. Use the status "CMPL" instead of this attribute.

9.42 0.2.8.2

- **Fix:** Fixed `stalker.db.create_alembic_table()` again to prevent extra row insertion.

9.43 0.2.8.1.1

- **Fix:** Fixed `stalker.db.create_alembic_table()` function to handle the situation where the table is already created.

9.44 0.2.8.1

- **Fix:** Fixed `stalker.db.create_alembic_table()` function, it is not using the `alembic` library anymore to create the `alembic_version` table, which was the proper way of doing it but it created a lot of problems when Stalker is installed as a package.

9.45 0.2.8

- **Update:** Stalker is now Python3 compatible.
- **New:** Added a new class called `Client` which can be used to track down information about the clients of `Projects`. Also added `Project.client` and `User.company` attributes which are referencing a Client instance allowing to add clients as normal users.

- **New:** `db.init()` now creates `alembic_version` table and stamps the most recent version number to that table allowing newly initialized databases to be considered in head revision.
- **Fix:** Fixed `Version._format_take_name()` method. It is now possible to use multiple underscore characters in `Version.take_name` attribute.

9.46 0.2.7.6

- **Update:** Removed `TimeLog._expand_task_schedule_timing()` method which was automatically adjusting the `schedule_timing` and `schedule_unit` of a Task to total duration of the TimeLogs of that particular task, thus increasing the schedule info with the entered time logs.

  But it was setting the `schedule_timing` to 0 in some certain cases and it was unnecessary because the main purpose of this method was to prevent TaskJuggler to raise any errors related to the inconsistencies between the schedule values and the duration of TimeLogs and TaskJuggler has never given a real error about that situation.

9.47 0.2.7.5

- **Fix:** Fixed Task parent/child relationship, previously setting the parent of a task to None was cascading a delete operation due to the "all, delete-orphan" setting of the Task parent/child relationship, this is updated to be "all, delete" and it is now safe to set the parent to None without causing the task to be deleted.

9.48 0.2.7.4

- **Fix:** Fixed the following columns column type from String to Text:
  - Permissions.class_name
  - SimpleEntities.description
  - Links.full_path
  - Structures.custom_template
  - FilenameTemplates.path
  - FilenameTemplates.filename
  - Tickets.summary
  - Wiki.title
  - Wiki.content

  and specified a size for the following columns:
  - SimpleEntities.html_class -> String(32)
  - SimpleEntities.html_style -> String(32)
  - FilenameTemplates.target_entity_type -> String(32)

  to be compatible with MySQL.
- **Update:** It is now possible to create TimeLog instances for a Task with PREV status.

9.49 0.2.7.3

- **Fix:** Fixed `Task.update_status_with_dependent_statuses()` method for a Task where there is no dependency but the status is DREV. Now calling `Task.update_status_with_dependent_statuses()` will set the status to RTS if there is no `TimeLog` for that task and will set the status to WIP if the task has time logs.

9.50 0.2.7.2

- **Update:** `TaskJugglerScheduler` is now 466x faster when dumping all the data to TJP file. So with this new update it is taking only 1.5 seconds to dump ~20k tasks to a valid TJP file where it was around ~10 minutes in previous implementation. The speed enhancements is available only to PostgreSQL dialect for now.

9.51 0.2.7.1

- **Fix:** Fixed TimeLog output in one line per task in `Task.to_tjp()`.
- **New:** Added `TaskJugglerScheduler` now accepts a new argument called `compute_resources` which when set to True will also consider *Task.alternative_resources* attribute and will fill `Task.computed_resources` attribute for each Task. With `TaskJugglerScheduler` when the total number of Task is around 15k it will take around 7 minutes to generate this data, so by default it is set to False.

9.52 0.2.7

- **New:** Added `efficiency` attribute to `User` class. See User documentation for more info.

9.53 0.2.6.14

- **Fix:** Fixed an **autoflush** problem in `Studio.schedule()` method.

9.54 0.2.6.13

- **New:** Added `Repository.make_relative()` method, which makes the given path to relative to the repository root. It considers that the path is already in the repository. So for now, be careful about not to pass a path outside of the repository.

9.55 0.2.6.12

- **Update:** `TaskJugglerScheduler.schedule()` method now uses the `Studio.start` and `Studio.end` values for the scheduling range instead of the hardcoded dates.

9.56 0.2.6.11

- **Update:** `Task.create_time_log()` method now returns the created `TimeLog` instance.

9.57 0.2.6.10

- **Fix:** Fixed an `autoflush` issue in `Task.update_status_with_children_statuses()` method.

9.58 0.2.6.9

- **Update:** `Studio.is_scheduling` and `Studio.is_scheduling_by` attributes will not be updated or checked at the beginning of the `Studio.schedule()` method. It is the duty of the user to check those attributes before calling `Studio.schedule()`. This is done in this way because without being able to do a db commit inside `Studio.schedule()` method (which is the case with transaction managers which may be used in web applications like **Stalker Pyramid**) it is not possible to persist and thus use those variables. So, to be able to use those attributes meaningfully the user should set them. Those variables will be set to False and None accordingly by the `Studio.schedule()` method after the scheduling is done.

9.59 0.2.6.8

- **Fix:** Fixed a deadlock in `TaskJugglerScheduler.schedule()` method related with the `Popen.stderr.readlines()` blocking the TaskJuggler process without being able to read the output buffer.

9.60 0.2.6.7

- **Update:** `TaskJugglerScheduler.schedule()` is now using bulk inserts and updates which is way faster than doing it with pure Python. Use `parsing_method` (0: SQL, 1: Python) to choose between SQL or Pure Python implementation. Also updated `Studio.schedule()` to take in a `parsing_method` parameter.

9.61 0.2.6.6

- **Update:** The `cut_in`, `cut_out` and `cut_duration` attribute behaviour and the attribute order is updated in `Shot` class. So, if three of the values are given, then the `cut_duration` attribute value will be calculated from `cut_in` and `cut_out` attribute values. In any case `cut_out` precedes `cut_duration`, and if none of them given `cut_in` and `cut_duration` values will default to 1 and `cut_out` will be calculated by using `cut_in` and `cut_duration`.

9.62 0.2.6.5

- **New:** Entity to Note relation is now Many-to-Many. So one Note can now be assigned more than one Entity.
- **New:** Added alembic revision for `Entity_Notes` table creation and data migration from `Notes` table to `Entity_Notes` table. So all notes are preserved.
- **Fix:** Fixed `Shot.cut_duration` attribute initialization on `Shot` instances restored from database.
- **Fix:** Fixed `Studios.is_scheduling_by` relationship configuration, which was wrongly referencing the `Studios.last_scheduled_by_id` column instead of `Studios.is_scheduled_by_id` column.

9.63 0.2.6.4

- **New:** Added a `Task.review_set(review_number)` method to get the desired set of reviews. It will return the latest set of reviews if `review_number` is skipped or it is None.
- **Update:** Removed `Task.approve()` it was making things complex than it should be.

9.64 0.2.6.3

- **Fix:** Added `Page` to `class_names` in `db.init()`.
- **Fix:** Fixed `TimeLog` tjp representation to use bot the `start` and `end` date values instead of the `start` and `duration`. This is much better because it is independent from the timing resolution settings.

9.65 0.2.6.2

- **Fix:** Fixed `stalker.models.studio.schedule()` method, and prevented it to call `DBSession.commit()` which causes errors if there is a transaction manager.
- **Fix:** Fixed `stalker.models._parse_csv_file()` method for empty computed resources list.

9.66 0.2.6.1

- **New:** `stalker.models.task.TimeLog` instances are now checking if the dependency relation between the task that receives the time log and the tasks that the task depends to will be violated in terms of the start and end dates and raises a `DependencyViolationError` if it is the case.

9.67 0.2.6

- **New:** Added `stalker.models.wiki.Page` class, for holding a per Project wiki.

9.68 0.2.5.5

- **Fix:** `Review.task` attribute now accepts None but this is mainly done to allow its relation to the `Task` instance can be broken when it needs to be deleted without issuing a database commit.

9.69 0.2.5.4

- **Update:** The following column names are updated:
  - `Tasks._review_number` to `Tasks.review_number`
  - `Tasks._schedule_seconds` to `Tasks.schedule_seconds`
  - `Tasks._total_logged_seconds` to `Tasks.total_logged_seconds`
  - `Reviews._review_number` to `Reviews.review_number`
  - `Shots._cut_in` to `Shots.cut_in`
  - `Shots._cut_out` to `Shots.cut_out`

Also updated alembic migration to create columns with those names.

- **Update:** Updated Alembic revision `433d9caaafab` (the one related with stalker 2.5 update) to also include following updates:
  - Create StatusLists for Tasks, Asset, Shot and Sequences and add all the Statuses in the Task Status Workflow.
  - Remove `NEW` from all of the status lists of Task, Asset, Shot and Sequence.
  - Update all the `PREV` tasks to `WIP` to let them use the new Review Workflow.
  - Update the `Tasks.review_number` to 0 for all tasks.
  - Create StatusLists and Statuses (`NEW`, `RREV`, `APP`) for Reviews.
  - Remove any other status then defined in the Task Status Workflow from Task, Asset, Shot and Sequence status list.

9.70 0.2.5.3

- **Fix:** Fixed a bug in `Task` class where trying to remove the dependencies will raise an `AttributeError` caused by the `Task._previously_removed_dependent_tasks` attribute.

9.71 0.2.5.2

- **New:** Task instances now have two new properties called `path` and `absolute_path`. As in Version instances, these are the rendered version of the related FilenameTemplate object in the related Project. The `path` attribute is Repository root relative and `absolute_path` is the absolute path including the OS dependent Repository path.
- **Update:** Updated alembic revision with revision number "433d9caaafab" to also create Statuses introduced with Stalker v0.2.5.

9.72 0.2.5.1

- **Update:** `Version.__repr__` results with a more readable string.
- **New:** Added a generalized generator called `stalker.models.walk_hierarchy()` that walks and yields the entities over the given attribute in DFS or BFS fashion.
- **New:** Added `Task.walk_hierarchy()` which iterates over the hierarchy of the task. It walks in a breadth first fashion. Use `method=0` to walk in depth first.
- **New:** Added `Task.walk_dependencies()` which iterates over the dependencies of the task. It walks in a breadth first fashion. Use `method=0` to walk in depth first.
- **New:** Added `Version.walk_hierarchy()` which iterates over the hierarchy of the version. It walks in a depth first fashion. Use `method=1` to walk in breadth first.
- **New:** Added `Version.walk_inputs()` which iterates over the inputs of the version. It walks in a depth first fashion. Use `method=1` to walk in breath first.
- **Update:** `stalker.models.check_circular_dependency()` function is now using `stalker.models.walk_hierarchy()` instead of recursion over itself, which makes it more robust in deep hierarchies.
- **Fix:** `db.init()` now updates the statuses of already created status lists for `Task`, `Asset`, `Shot` and `Sequence` classes.

9.73 0.2.5

- **Update:** `Revision` class is renamed to `Review` and introduced a couple of new attributes.
- **New:** Added a new workflow called "Task Review Workflow". Please see the documentation about the new workflow.
- **Update:** `Task.responsible` attribute is now a list which allows multiple responsible to be set for a `Task`.
- **New:** Because of the new "Task Review Workflow" task statuses which are normally created in Stalker Pyramid are now automatically created in Stalker database initialization. The new statuses are **Waiting For Dependency (WFD)**, **Ready To Start (RTS)**, **Work In Progress (WIP)**, **Pending Review (PREV)**, **Has Revision (HREV)**, **On Hold (OH)**, **Stopped (STOP)** and **Completed (CMPL)** are all used in `Task`, `Asset`, `Shot` and `Sequence` status lists by default.
- **New:** Because of the new "Task Review Workflow" also a status list for `Review` class is created by default. It contains the statuses of **New (NEW)**, **Requested Revision (RREV)** and **Approved (APP)**.
- **Fix:** `Users.login` column is now unique.
- **Update:** Ticket workflow in config is now using the proper status names instead of the lower case names of the statuses.
- **New:** Added a new exception called **StatusError** which states the entity status is not suitable for the action it is applied to.
- **New:** `Studio` instance now stores the scheduling state to the database to prevent two scheduling process to override each other. It also stores the last schedule message and the last schedule date and the id of the user who has done the scheduling.
- **New:** The **Task Dependency** relation is now using an **Association Object** instead of just a **Secondary Table**. The `Task.depends` and `Task.dependent_of` attributes are now *association_proxies*.

  Also added extra parameters like `dependency_target`, `gap_timing`, `gap_unit` and `gap_model` to the dependency relation. So all of the dependency relations are now able to hold those extra information.

  Updated the `task_tjp_template` to reflect the details of the dependencies that a task has.
- **New:** `ScheduleMixin` class now has some default class attributes that will allow customizations in inherited classes. This is mainly done for `TaskDependency` class and for `the gap_timing`, `gap_unit`, `gap_model` attributes which are in fact synonyms of `schedule_timing`, `schedule_unit` and `schedule_model` attributes coming from the `ScheduleMixin` class. So by using the `__default_schedule_attr_name__` Stalker is able to display error messages complaining about `gap_timing` attribute instead of `schedule_timing` etc.
- **New:** Updating a task by calling `Task.request_revision()` will now set the `TaskDependency.dependency_target` to **'onstart'** for tasks those are depending to the revised task and updated to have a status of **DREV**, **OH** or **STOP**. Thus, TaskJuggler will be able to continue scheduling these tasks even if the tasks are now working together.
- **Update:** Updated the TaskJuggler templates to make the tjp output a little bit more readable.
- **New:** `ScheduleMixin` now creates more localized (to the mixed in class) column and enum type names in the mixed in classes.

  For example, it creates the `TaskScheduleModel` enum type for `Task` class and for `TaskDependency` it creates `TaskDependencyGapModel` with the same setup following the `{{class_name}}{{attr_name}}Model` template.

  Also it creates `schedule_model` column for `Task`, and `gap_model` for `TaskDependency` class.
- **Update:** Renamed the `TaskScheduleUnit` enum type name to `TimeUnit` in `ScheduleMixin`.

9.74 0.2.4

- **New:** Added new class called `Revision` to hold info about Task revisions.
- **Update:** Renamed `ScheduleMixin` to `DateRangeMixin`.
- **New:** Added a new mixin called `ScheduleMixin` (replacing the old one) which adds attributes like `schedule_timing`, `schedule_unit`, `schedule_model` and `schedule_constraint`.
- **New:** Added `Task.tickets` and `Task.open_tickets` properties.
- **Update:** Removed unnecessary arguments (`project_lead`, `tasks`, `watching`, `last_login`) from User class.
- **Update:** The `timing_resolution` attribute is moved from the `DateRangeMixin` to `Studio` class. So instances of classes like `Project` or `Task` will not have their own timing resolution anymore.
- **New:** The `Studio` instance now overrides the values on `stalker.defaults` on creation and on load, and also the `db.setup()` function lets the first `Studio` instance that it finds to update the defaults. So it is now possible to use `stalker.defaults` all the time without worrying about the Studio settings.
- **Update:** The `Studio.yearly_working_days` value is now always an integer.
- **New:** Added a new method `ScheduleMixin.least_meaningful_time_unit()` to calculate the most appropriate timing unit and the value of the given seconds which represents an interval of time.

  So it will convert 3600 seconds to 1 hours, and 8424000 seconds to 1 years if it represents working time (`as_working_time=True`) or 2340 hours if it is representing the calendar time.
- **New:** Added a new method to `ScheduleMixin` called `to_seconds()`. The `to_seconds()` method converts the given schedule info values (`schedule_timing`, `schedule_unit`, `schedule_model`) to seconds considering if the given `schedule_model` is work time based ('effort' or 'length') or calendar time based ('duration').
- **New:** Added a new method to `ScheduleMixin` called `schedule_seconds` which you may recognise from `Task` class. What it does is pretty much the same as in the `Task` class, it converts the given schedule info values to seconds.
- **Update:** In `DateRangeMixin`, when the `start`, `end` or `duration` arguments given so that the duration is smaller then the `defaults.timing_resolution` the `defaults.timing_resolution` will be used as the `duration` and the `end` will be recalculated by anchoring the `start` value.
- **New:** Adding a `TimeLog` to a `Task` and extending its schedule info values now will always use the least meaningful timing unit. So expanding a task from 16 hours to 18 hours will result a task with 2 days of schedule (considering the `daily_working_hours = 9`).
- **Update:** Moved the `daily_working_hours` attribute from `Studio` class to `WorkingHours` class as it was much related to this one then `Studio` class. Left a property with the same name in the `Studio` class, so it will still function as it was before but there will be no column in the database for that attribute anymore.

9.75 0.2.3.5

- **Fix:** Fixed a bug in `stalker.models.auth.LocalSession` where stalker was complaining about "copy_reg" module, it seems that it is related to this bug.

9.76 0.2.3.4

- **Update:** Fixed a little bug in Link.extension property setter.
- **New:** Moved the stalker.models.env.EnvironmentBase class to "Anima Tools" python module.

- **Fix:** Fixed a bug in stalker.models.task.Task._responsible_getter() where it was always returning the greatest parents responsible as the responsible for the child task when the responsible is set to None for the child.
- **New:** Added `stalker.models.version.Version.naming_parents` which returns a list of parents starting from the closest parent Asset, Shot or Sequence.
- **New:** `stalker.models.version.Version.nice_name` now generates a name starting from the closest Asset, Shot or Sequence parent.

9.77 0.2.3.3

- **New:** `Ticket` action methods (`resolve`, `accept`, `reassign`, `reopen`) now return the created `TicketLog` instance.

9.78 0.2.3.2

- **Update:** Added tests for negative or zero fps value in Project class.
- **Fix:** Minor fix to `schedule_timing` argument in Task class, where IDEs where assuming that the value passed to the `schedule_timing` should be integer where as it accepts floats also.
- **Update:** Removed `bg_color` and `fg_color` attributes (and columns) from Status class. Use SimpleEntity.html_class and SimpleEntity.html_style attributes instead.
- **New:** Added `Project.open_tickets` property.

9.79 0.2.3.1

- **Fix:** Fixed an inconvenience in SimpleEntity.__init__() when a date_created argument with a value is later than datetime.datetime.now() is supplied and the date_updated argument is skipped or given as None, then the date_updated attribute value was generated from datetime.datetime.now() this was causing an unnecessary ValueError. This is fixed by directly copying the date_created value to date_updated value when it is skipped or None.

9.80 0.2.3

- **New:** SimpleEntity now have two new attributes called `html_style` and `html_class` which can be used in storing cosmetic html values.

9.81 0.2.2.3

- **Update:** Note.content attribute is now a synonym of the Note.description attribute.

9.82 0.2.2.2

- **Update:** Studio.schedule() now returns information about how much did it take to schedule the tasks.
- **Update:** Studio.to_tjp() now returns information about how much did it take to complete the conversion.

9.83 0.2.2.1

- **Fix:** Task.percent_complete() now calculates the percent complete correctly.

9.84 0.2.2

- **Update:** Added cascade attributes to all necessary relations for all the classes.
- **Update:** The Version class is not mixed with the StatusMixin anymore. So the versions are not going to be statusable anymore. Also created alembic revision (a6598cde6b) for that update.

9.85 0.2.1.2

- **Update:** TaskJugglerScheduler and the Studio classes are now returning the stderr message out of their `schedule()` methods.

9.86 0.2.1.1

- **Fix:** Disabled some deep debug messages on TaskJugglerScheduler._parse_csv_file().
- **Fix:** Fixed a flush issue related to the Task.parent attribute which is lazily loaded in Task._schedule_seconds_setter().

9.87 0.2.1

- **Fix:** As usual distutil thinks `0.2.0` is a lower version number than `0.2.0.rc5` (I should have read the documentation again and used `0.2.0.c5` instead of `0.2.0.rc5`) so this is a dummy update to just to fix the version number.

9.88 0.2.0

- **Update:** Vacation tjp template now includes the time values of the start and end dates of the Vacation instance.

9.89 0.2.0.rc5

- **Update:** For a container task, `Task.total_logged_seconds` and `Task.schedule_seconds` attributes are now using the info of the child tasks. Also these attributes are cached to database, so instead of querying the child tasks all the time, the calculated data is cached and whenever a TimeLog is created or updated for a child task (which changes the `total_logged_seconds` for the child task) or the `schedule_timing` or `schedule_unit` attributes are updated, the cached values are updated on the parents. Allowing Stalker to display percent_complete info of a container task without loading any of its children.
- **New:** Added `Task.percent_complete` attribute, which calculates the percent of completeness of the task based on the `Task.total_logged_seconds` and `Task.schedule_seconds` attributes.
- **Fix:** Added `TimeLog.__eq__()` operator to more robustly check if the time logs are overlapping.

- **New:** Added `Project.percent_complete`, `Percent.total_logged_seconds` and `Project.schedule_seconds` attributes.
- **Update:** `ScheduleMixin._validate_dates()` does not set the date values anymore, it just return the calculated and validated `start`, `end` and `duration` values.
- **Update:** `Vacation` now can be created without a `User` instance, effectively making the `Vacation` a `Studio` wide vacation, which applies to all users.
- **Update:** `Vacation.__strictly_typed__` is updated to `False`, so there is no need to create a `Type` instance to be able to create a `Vacation`.
- **New:** `Studio.vacations` property now returns the `Vacation` instances which has no *user*.
- **Update:** `Task.start` and `Task.end` values are no more read from children Tasks for a container task over and over again but calculated whenever the start and end values of a child task are changed or a new child is appended or removed.
- **Update:** `SimpleEntity.description` validation routine doesn't convert the input to string anymore, but checks the given description value against being a string or unicode instance.
- **New:** Added `Ticket.summary` field.
- **Fix:** Fixed `Link.extension`, it is now accepting unicode.

9.90 0.2.0.rc4

- **New:** Added a new attribute to `Version` class called `latest_version` which holds the latest version in the version queue.
- **New:** To optimize the database connection times, `stalker.db.setup()` will not try to initialize the database every time it is called anymore. This leads a ~4x speed up in database connection setup. To initialize a newly created database please use:

```
# for a newly created database
from stalker import db
db.setup() # connects to database
db.init()  # fills some default values to be used with Stalker

# for any subsequent access just use (don't need to call db.init())
db.setup()
```

- **Update:** Removed all `__init_on_load()` methods from all of the classes. It was causing SQLAlchemy to eagerly load relations, thus slowing down queries in certain cases (especially in `Task.parent` -> `Task.children` relation).
- **Fix:** Fixed `Vacation` class tj3 format.
- **Fix:** `Studio.now` attribute was not properly working when the `Studio` instance has been restored from database.

9.91 0.2.0.rc3

- **New:** Added a new attribute to `Task` class called `responsible`.
- **Update:** Removed `Sequence.lead_id` use `Task.reponsible` instead.
- **Update:** Updated documentation to include documentation about Configuring Stalker with `config.py`.
- **Update:** The `duration` argument in `Task` class is removed. It is somehow against the idea of having `schedule_model` and `schedule_timing` arguments (`schedule_model='duration'` is kind of the same).

- **Update:** Updated `Task` class documentation.

9.92 0.2.0.rc2

- **New:** Added `Version.created_with` attribute to track the environment or host program name that a particular `Version` instance is created with.

9.93 0.2.0.rc1

- **Update:** Moved the Pyramid part of the system to another package called `stalker_pyramid`.
- **Fix:** Fixed `setup.py` where importing `stalker` to get the `__version__` variable causing problems.

9.94 0.2.0.b9

- **New:** Added `Version.latest_published_version` and `Version.is_latest_published_version()`.
- **Fix:** Fixed `Version.__eq__()`, now Stalker correctly distinguishes different Version instances.
- **New:** Added `Repository.to_linux_path()`, `Repository.to_windows_path()`, `Repository.to_osx_path()` and `Repository.to_native_path()` to the `Repository` class.
- **New:** Added `Repository.is_in_repo(path)` which checks if the given path is in this repo.

9.95 0.2.0.b8

- **Update:** Renamed **Version.version_of** attribute to **Version.task**.
- **Fix:** Fixed **Version.version_number** where it was not possible to have a version number bigger than 2.
- **Fix:** In **db.setup()** Ticket statuses are only created if there aren't any.
- **Fix:** Added **Vacation** class to the registered class list in stalker.db.

9.96 0.2.0.b7

- **Update: Task.schedule_constraint** is now reflected to the tjp file correctly.
- **Fix: check_circular_dependency()** now checks if the **entity** and the **other_entity** are the same.
- **Fix: Task.to_tjp()** now correctly add the dependent tasks of a container task.
- **Fix: Task.__eq__()** now correctly considers the parent, depends, resources, start and end dates.
- **Update: Task.priority** is now reflected in tjp file if it is different than the default value (500).
- **New::** Added a new class called **Vacation** to hold user vacations.
- **Update:** Removed dependencies to `pyramid.security.Allow` and `pyramid.security.Deny` in couple of packages.
- **Update:** Changed the way the `stalker.defaults` is created.
- **Fix: EnvironmentBase.get_version_from_full_path()**, **EnvironmentBase.get_versions_from_path()**, **EnvironmentBase.trim_repo_path()**, **EnvironmentBase.find_repo** methods are now working properly.

- **Update:** Added **Version.absolute_full_path** property which renders the absolute full path which also includes the repository path.
- **Update:** Added **Version.absolute_path** property which renders the absolute path which also includes the repository path.

9.97 0.2.0.b6

- **Fix:** Fixed **LocalSession._write_data()**, previously it was not creating the local session folder.
- **New:** Added a new method called **LocalSession.delete()** to remove the local session file.
- **Update: Link.full_path** can now be set to an empty string. This is updated in this way for **Version** class.
- **Update:** Updated the formatting of **SimpleEntity.nice_name**, it is now possible to have uppercase letters and camel case format will be preserved.
- **Update**: **Version.take_name** formatting is enhanced.
- **New**: **Task** class is now mixed in with **ReferenceMixin** making it unnecessary to have **Asset**, **Shot** and **Sequence** classes all mixed in individually. Thus removed the **ReferenceMixin** from **Asset**, **Shot** and **Sequence** classes.
- **Update**: Added **Task.schedule_model** validation and its tests.
- **New**: Added **ScheduleMixin.total_seconds** and **ScheduleMixin.computed_total_seconds**.

9.98 0.2.0.b5

- **New: Version** class now has two new attributes called `parent` and `children` which will be used in tracking of the history of Version instances and track which Versions are derived from which Version.
- **New: Versions** instances are now derived from **Link** class and not **Entity**.
- **Update:** Added new revisions to **alembic** to reflect the change in **Versions** table.
- **Update: Links.path** is renamed to **Links.full_path** and added three new attributes called **path**, **filename** and **extension**.
- **Update:** Added new revisions to alembic to reflect the change in **Links** table.
- **New:** Added a new class called **LocalSession** to store session data in users local filesystem. It is going to be replaced with some other system like **Beaker**.
- **Fix:** Database part of Stalker can now be imported without depending to **Pyramid**.
- **Fix:** Fixed documentation errors that **Sphinx** complained about.

9.99 0.2.0.b4

- No changes in SOM.

9.100 0.2.0.b3

- **Update:** FilenameTemplate's are not `strictly typed` anymore.
- **Update:** Removed the FilenameTemplate type initialization, FilenameTemplates do not depend on Types anymore.

- **Update:** Added back the `plural_class_name` (previously `plural_name`) property to the ORMClass class, so all the classes in SOM now have this new property.
- **Update:** Added `accepts_references` attribute to the EntityType class.
- **New:** The Link class has a new attribute called `original_filename` to store the original file names of link files.
- **New:** Added **alembic** to the project requirements.
- **New:** Added alembic migrations which adds the `accepts_references` column to `EntityTypes` table and `original_name` to the `Links` table.

9.101 0.2.0.b2

- Stalker is now compatible with Python 2.6.
- Task:
  - **Update:** Tasks now have a new attribute called `watchers` which holds a list of User instances watching the particular Task.
  - **Update:** Users now have a new attribute called `watching` which is a list of Task instances that this user is watching.
- TimeLog:
  - **Update:** TimeLog instances will expand Task.schedule_timing value automatically if the total amount of logged time is more than the schedule_timing value.
  - **Update:** TimeLogs are now considered while scheduling the task.
  - **Fix:** TimeLogs raises OverBookedError when appending the same TimeLog instance to the same resource.
- Auth:
  - **Fix:** The default ACLs for determining the permissions are now working properly.

9.102 0.2.0.b1

- WorkingHours.is_working_hour() is working now.
- WorkingHours class is moved from stalker.models.project to stalker.models.studio module.
- `daily_working_hours` attribute is moved from stalker.models.project.Project to stalker.models.studio.Studio class.
- Repository path variables now ends with a forward slash even if it is not given.
- Updated Project classes validation messages to correlate with Stalker standard.
- Implementation of the Studio class is finished. The scheduling works like a charm.
- It is now possible to use any characters in SimpleEntity.name and the derived classes.
- Booking class is renamed to TimeLog.

9.103 0.2.0.a10

- Added new attribute to WorkingHours class called `weekly_working_hours`, which calculates the weekly working hours based on the working hours defined in the instance.

- Task class now has a new attribute called `schedule_timing` which is replacing the `effort`, `length` and `duration` attributes. Together with the `schedule_model` attribute it will be used in scheduling the Task.
- Updated the config system to the one used in oyProjectManager (based on Sphinx config system). Now to reach the defaults:

```
# instead of doing the following
from stalker.conf import defaults # not valid anymore

# use this
from stalker import defaults
```

  If the above idiom is used, the old `defaults` module behaviour is retained, so no code change is required other than the new lower case config variable names.

9.104 0.2.0.a9

- A new property called `to_tjp` added to the SimpleEntity class which needs to be implemented in the child and is going to be used in TaskJuggler integration.
- A new attribute called `is_scheduled` added to Task class and it is going to be used in Gantt charts. Where it will lock the class and will not try to snap it to anywhere if it is scheduled.
- Changed the `resolution` attribute name to `timing_resolution` to comply with TaskJuggler.
- ScheduleMixin:
  - Updated ScheduleMixin class documentation.
  - There are two new read-only attributes called `computed_start` and `computed_end`. These attributes will be used in storing of the values calculated by TaskJuggler, and will be used in Gantt Charts if available.
  - Added `computed_duration`.
- Task:
  - Arranged the TaskJuggler workflow.
  - The task will use the effort > length > duration attributes in *to_tjp* property.
- Changed the license of Stalker from BSD-2 to LGPL 2.1. Any version previous to 0.2.0.a9 will be still BSD-2 and any version from and including 0.2.0.a9 will be distributed under LGPL 2.1 license.
- Added new types of classes called Schedulers which are going to be used in scheduling the tasks.
- Added TaskJugglerScheduler, it uses the given project and schedules its tasks.

9.105 0.2.0.a8

- TagSelect now can be filled by setting its `value` attribute (Ex: TagSelect.set('value', data))
- Added a new method called `is_root` to Task class. It is true for tasks where there are no parents.
- Added a new attribute called `users` to the Department class which is a synonym for the `members` attribute.
- Task:
  - Task class is now preventing one of the dependents to be set as the parent of a task.
  - Task class is now preventing one of the parents to be set as the one of the dependents of a task.
  - Fixed `autoflush` bugs in Task class.

- Fixed *admin* users department initialization.
- Added `thumbnail` attribute to the SimpleEntity class which is a reference to a Link instance, showing the path of the thumbnail.
- Fixed Circular Dependency bug in Task class, where a parent of a newly created task is depending to another task which is set as the dependee for this newly created task (T1 -> T3 -> T2 -> T1 (parent relation) -> T3 -> T2 etc.).

9.106 0.2.0.a7

- Changed these default setting value names to corresponding new names:
  - `DEFAULT_TASK_DURATION` -> `TASK_DURATION`
  - `DEFAULT_TASK_PRIORITY` -> `TASK_PRIORITY`
  - `DEFAULT_VERSION_TAKE_NAME` -> `VERSION_TAKE_NAME`
  - `DEFAULT_TICKET_LABEL` -> `TICKET_LABEL`
  - `DEFAULT_ACTIONS` -> `ACTIONS`
  - `DEFAULT_BG_COLOR` -> `BG_COLOR`
  - `DEFAULT_FG_COLOR` -> `FG_COLOR`
- stalker.conf.defaults:
  - Added default settings for project working hours (`WORKING_HOURS`, `DAY_ORDER`, `DAILY_WORKING_HOURS`)
  - Added a new variable for setting the task time resolution called `TIME_RESOLUTION`.
- stalker.models.project.Project:
  - Removed Project.project_tasks attribute, use Project.tasks directly to get all the Tasks in that project. For root task you can do a quick query:

    ```
    Task.query.filter(Task.project==proj_id).filter(Task.parent==None).all()
    ```

    This will also return the Assets, Sequences and Shots in that project, which are also Tasks.
  - Users are now assigned to Projects by appending them to the Project.users list. This is done in this way to allow a reduced list of resources to be shown in the Task creation dialogs.
  - Added a new helper class for Project working hour management, called WorkingHours.
  - Added a new attribute to Project class called `working_hours` which holds stalker.models.project.WorkingHours instances to manage the Project working hours. It will directly be passed to TaskJuggler.
- stalker.models.task.Task:
  - Removed the Task.task_of attribute, use Task.parent to get the owner of this Task.
  - Task now has two new attributes called Task.parent and Task.children which allow more complex Task-to-Task relation.
  - Secondary table name for holding Task to Task dependency relation is renamed from `Task_Tasks` to `Task_Dependencies`.
  - check_circular_dependency function is now accepting a third argument which is the name of the attribute to be investigated for circular relationship. It is done in that way to be able to use the same function in searching for circular relations both in parent/child and depender/dependee relations.
- ScheduleMixin:

  - Added a new attribute to ScheduleMixin for time resolution adjustment. Default value is 1 hour and can be set with stalker.conf.defaults.TIME_RESOLUTION. Any finer time than the resolution is rounded to the closest multiply of the resolution. It is possible to set it from microseconds to years. Although 1 hour is a very reasonable resolution which is also the default resolution for TaskJuggler.
  - ScheduleMixin now uses datetime.datetime for the start and end attributes.
  - Renamed the `start_date` attribute to `start`.
  - Renamed the `end_date` attribute to `end`
- Removed the TaskableEntity.
- Asset, Sequence and Shot classes are now derived from Task class allowing more complex Task relation combined with the new parent/child relation of Tasks. Use Asset.children or Asset.tasks to reach the child tasks of that asset (same with Sequence and Shot classes).
- stalker.models.shot.Shot:
  - Removed the sequence and introduced sequences attribute in Shot class. Now one shot can be in more than one Sequence. Allowing more complex Shot/Sequence relations..
  - Shots can now be created without a Sequence instance. The sequence attribute is just used to group the Shots.
  - Shots now have a new attribute called `scenes`, holding Scene instances. It is created to group same shots occurring in the same scenes.
- In tests all the Warnings are now properly handled as Warnings.
- stalker.models.ticket.Ticket:
  - Ticket instances are now tied to Projects and it is now possible to create Tickets without supplying a Version. They are free now.
  - It is now possible to link any SimpleEntity to a Ticket.
  - The Ticket Workflow is now fully customizable. Use stalker.conf.defaults.TICKET_WORKFLOW dictionary to define the workflow and stalker.conf.defaults.TICKET_STATUS_ORDER for the order of the ticket statuses.
- Added a new class called `Scene` to manage Shots with another property.
- Removed the `output_path` attribute in FilenameTemplate class.
- Grouped the templates for each entity under a directory with the entity name.

9.107 0.2.0.a6

- Users now can have more than one Department.
- User instances now have two new properties for getting the user tickets (User.tickets) and the open tickets (User.open_tickets).
- New shortcut Task.project returns the Task.task_of.project value.
- Shot and Asset creation dialogs now automatically updated with the given Project instance info.
- User overview page is now reflection the new design.

9.108 0.2.0.a5

- The `code` attribute of the SimpleEntity is now introduced as a separate mixin. To let it be used by the classes it is really needed.

- The `query` method is now converted to a property so it is now possible to use it like a property as in the SQLAlchemy.orm.Session as shown below:

```
from stalker import Project
Project.query.all() # instead of Project.query().all()
```

- ScheduleMixin.due_date is renamed to ScheduleMixin.end_date.
- Added a new class attribute to SimpleEntity called `__auto_name__` which controls the naming of the instances and instances derived from SimpleEntity. If `__auto_name__` is set to True the `name` attribute of the instance will be automatically generated and it will have the following format:

```
{{ClassName}}_{{UUID4}}
```

  Here are a couple of naming examples:

```
Ticket_74bb46b0-29de-4f3e-b4e6-8bcf6aed352d
Version_2fa5749e-8cdb-4887-aef2-6d8cec6a4faa
```

- Fixed an autoflush issue with SQLAlchemy in StatusList class. Now the status column is again not nullable in StatusMixin.

9.109 0.2.0.a4

- Added a new class called EntityType to hold all the available class names and capabilities.
- Version class now has a new attribute called `inputs` to hold the inputs of the current Version instance. It is a list of Link instances.
- FilenameTemplate classes `path` and `filename` attributes are no more converted to string, so given a non string value will raise TypeError.
- Structure.custom_template now only accepts strings and None, setting it to anything else will raise a TypeError.
- Two Type's for FilenameTemplate's are created by default when initializing the database, first is called "Version" and it is used to define FilenameTemplates which are used for placing Version source files. The second one is called "Reference" and it is used when injecting references to a given class. Along with the FilenameTemplate.target_entity_type this will allow one to create two different FilenameTemplates for one class:

```
# first get the Types
vers_type = Type.query()\
            .filter_by(target_entity_type="FilenameTemplate")\
            .filter_by(type="Version")\
            .first()

ref_type = Type.query()\
           .filter_by(target_entity_type="FilenameTemplate")\
           .filter_by(type="Reference")\
           .first()

# lets create a FilenameTemplate for placing Asset Version files.
f_ver = FilenameTemplate(
    target_entity_type="Asset",
    type=vers_type,
    path="Assets/{{asset.type.code}}/{{asset.code}}/{{task.type.code}}",
    filename="{{asset.code}}_{{version.take_name}}_{{task.type.code}}_v{{'%03d
↪'|version.version_number}}{{link.extension}}"
    output_path="{{version.path}}/Outputs/{{version.take_name}}"
)
```

(continues on next page)

(continued from previous page)

```
# and now define a FilenameTemplate for placing Asset Reference files.
# no need to have an output_path here...
f_ref = FilenameTemplate(
    target_entity_type="Asset",
    type=ref_type,
    path="Assets/{{asset.type.code}}/{{asset.code}}/References",
    filename="{{link.type.code}}/{{link.id}}{{link.extension}}"
)
```

- stalker.db.register() now accepts only real classes instead of class names. This way it can store more information about classes.
- Status.bg_color and Status.fg_color attributes are now simple integers. And the Color class is removed.
- StatusMixin.status is now a ForeignKey to a the Statuses table, thus it is a real Status instance instead of an integer showing the index of the Status in the related StatusList. This way the Status of the object will not change if the content of the StatusList is changed.
- Added new attribute Project.project_tasks which holds all the direct or indirect Tasks created for that project.
- User.login_name is renamed to User.login.
- Removed the `first_name`, `last_name` and `initials` attributes from User class. Now the `name` and `code` attributes are going to be used, thus the `name` attribute is no more the equivalent of `login` and the `code` attribute is doing what was `initials` doing previously.

9.110 0.2.0.a3

- Status class now has two new attributes `bg_color` and `fg_color` to hold the UI colors of the Status instance. The colors are Color instances.

9.111 0.2.0.a2

- SimpleEntity now has an attribute called `generic_data` which can hold any kind of `SOM` object inside and it is a list.
- Changed the formatting rules for the `name` in SimpleEntity class, now it can start with a number, and it is not allowed to have multiple whitespace characters following each other.
- The `source` attribute in Version is renamed to `source_file`.
- The `version` attribute in Version is renamed to `version_number`.
- The `take` attribute in Version is renamed to `take_name`.
- The `version_number` in Version is now generated automatically if it is skipped or given as None or it is too low where there is already a version number for the same Version series (means attached to the same Task and has the same `take_name`.
- Moved the User class to `stalker.models.auth module`.
- Removed the `stalker.ext.auth` module because it is not necessary anymore. Thus the User now handles all the password conversions by itself.
- `PermissionGroup` is renamed back to Group again to match with the general naming of the authorization concept.
- Created two new classes for the Authorization system, first one is called Permission and the second one is a Mixin which is called ACLMixin which adds ACLs to the mixed in class. For now, only the User and Group classes are mixed with this mixin by default.

- The declarative Base class of SQLAlchemy is now created by binding it to a ORMClass (a random name) which lets all the derived class to have a method called `query` which will bypass the need of calling `DBSession.query(class_)` but instead just call `class_.query()`:

```
from stalker.models.auth import User
user_1 = User.query().filter_by(name='a user name').first()
```

9.112 0.2.0.a1

- Changed the `db.setup` arguments. It is now accepting a dictionary instead of just a string to comply with the SQLAlchemy scaffold and this dictionary should contain keys for the SQLAlchemy engine setup. There is another utility that comes with Pyramid to setup the database under the *scripts* folder, it is also working without any problem with stalker.db.
- The `session` variable is renamed to `DBSession` and is now a scopped session, so there is no need to use `DBSession.commit` it will be handled by the system it self.
- Even though the `DBSession` is using the Zope Transaction Manager extension normally, in the database tests no extension is used because the transaction manager was swallowing all errors and it was a little weird to try to catch this errors out of the `with` block.
- Refactored the code, all the models are now in separate python files, but can be directly imported from the main stalker module as shown:

```
from stalker import User, Department, Task
```

  By using this kind of organization, both development and usage will be eased out.
- `task_of` now only accepts TaskableEntity instances.
- Updated the examples. It is now showing how to extend SOM correctly.
- Updated the references to the SOM classes in docstrings and rst files.
- Removed the `Review` class. And introduced the much handier Ticket class. Now reviewing a data is the process of creating Ticket's to that data.
- The database is now initialized with a StatusList and a couple of Statuses appropriate for Ticket instances.
- The database is now initialized with two Type instances ('Enhancement' and 'Defect') suitable for Ticket instances.
- StatusMixin now stores the status attribute as an Integer showing the index of the Status in the `status_list` attribute but when asked for the value of `StatusMixin.status` attribute it will return a proper Status instance and the attribute can be set with an integer or with a proper Status instance.

Index

F

K

L

P

R

S

T

V

W

Y