sandglass Documentation

Release 0.1alpha0

Fabian Büchler, Jerónimo Albi, Theodor Krampf

April 03, 2012

CONTENTS

Sandglass is a working- and project-time tracking application, based on Django.

Note: sandglass is not yet released and still in very early stage of development.

CHAPTER

ONE

CONTENTS

CHAPTER

TWO

GETTING HELP

The primary way of getting help is through our mailing list hosted at google groups.

Also, you may want to follow @fabianbuechler, @jeronimoalbi or @krampf on Twitter.

CHAPTER

THREE

API DOCS

3.1 API Docs

This is **sandglass'** API documentation, where we try to explain implementation details.

There is also the *modindex*, for reference purposes.

3.1.1 Models & Fields

Models

Accountable

class sandglass.models.accountable.**Accountable**(**args*, ***kwargs*)
Abstract base model for **accountable objects**.

An accountable objects basically *has a price*. Depending on the accounting type (by effort or lump-sum), the objects can define an hours budget or a cost estimate (or both).

Activity

class sandglass.models.activity.**Activity**(**args*, ***kwargs*)
Activity(id, created, modified, project_id, task_id, description, activity_type, start, end, duration, locked)

ActivityPeriod

class sandglass.models.activityperiod.**ActivityPeriod**(**args*, ***kwargs*)
Abstract base model for objects which are **active for a certain period of time**.

After the `end` date

Client

class sandglass.models.client.**Client**(**args*, ***kwargs*)
Client(id, created, modified, name)

ContactPerson

class sandglass.models.contactperson.**ContactPerson**(**args*, ***kwargs*)
ContactPerson(id, created, modified, first_name, last_name, email, phone, address, zip_code, city, country)

OverTime

class sandglass.models.overtime.**OverTime**(**args*, ***kwargs*)
OverTime record for a user.

Project

class sandglass.models.project.**Project**(**args*, ***kwargs*)
Project(id, accounting_type, hours_budget, cost_estimate, created, modified, active, active_from, active_to, name, acronym, parent_id, client_id, activity_description_required)

internal
Company-internal projects have no `Client` assigned.

PublicHoliday

class sandglass.models.publicholiday.**PublicHoliday**(**args*, ***kwargs*)
PublicHoliday(id, name, date)

ReportQuery

class sandglass.models.reportquery.**ReportQuery**(**args*, ***kwargs*)
ReportQuery(id, created, modified, name, description, date_from, date_to, include_sub_projects, include_sub_tasks)

ReportSnapshot

class sandglass.models.reportsnapshot.**ReportSnapshot**(**args*, ***kwargs*)
ReportSnapshot(id, created, modified)

ReviewedReport

class sandglass.models.reviewedreport.**ReviewedReport**(**args*, ***kwargs*)
ReviewedReport(id, created, modified, query_id, snapshot_id, verified)

Tag

class sandglass.models.tag.**Tag**(**args*, ***kwargs*)
Model definition for tags.

Tags are added by users with right permissions. Normal users can't create new tags, instead of that they can create a tag alias. A tag alias can be used to have custom tags by user. Field *alias_of* and *owner* are initialized when a Tag instance is an alias.

Task

class sandglass.models.task.**Task**(**args*, ***kwargs*)
Task(id, accounting_type, hours_budget, cost_estimate, created, modified, active, active_from, active_to, name, acronym, parent_id)

Team

class sandglass.models.team.**Team**(**args*, ***kwargs*)
Team(id, created, modified, name, description)

Theme

class sandglass.models.theme.**Theme**(**args*, ***kwargs*)
Theme(id, created, modified, name, description, theme_folder)

TimeModel

class sandglass.models.timemodel.**TimeModel**(**args*, ***kwargs*)
Set of "rules" for employee working hours.

TimeSheet

class sandglass.models.timesheet.**TimeSheet**(**args*, ***kwargs*)
Regular settlement and savepoint for a user's (employee's) status.

For future calculations, the last TimeSheet is used as basis. Every new user should have an initial TimeSheet.

UserProfile

class sandglass.models.userprofile.**UserProfile**(**args*, ***kwargs*)
Extension model tied to `django.contrib.auth.models.User`.

Adds sandglass-specific user properties.

sandglass.models.userprofile.**create_user_profile**(*sender*, *instance*, *created*, ***kwargs*)
Creates a `UserProfile` as soon as a `User` is created.

Custom Fields

3.1.2 Views & Forms

Views

Forms

3.1.3 Deploy Templates

Deploy Templates

3.1.4 Library

Library

3.1.5 Settings & Configuration

Configuration

Settings

- *modindex*

CHAPTER

FOUR

CONTRIBUTING

4.1 Contributing

Since **sandglass** is an open-source project, it is always looking for contributors. Below, you'll find how to get started on sandglass development and what rules we'd like you to follow, if you wish to contribute code.

4.1.1 Community

The primary way of getting in touch with the sandglass developers is through our mailing list hosted at google groups.

Also, you may want to follow @fabianbuechler, @jeronimoalbi or @krampf on Twitter.

4.1.2 Quickstart

If you would like to contribute to sandglass just follow these quidelines:

1. Sandglass is hosted on bitbucket at https://bitbucket.org/fabianbuechler/sandglass/
2. Create a fork, add your changes there.
3. Send a pull request whenever you feel your code is ready.

4.1.3 Detailed Contribution Process

Contributing Bug-Reports

Bug reports, and of course feature requests, are highly welcome, since they help us to improve the quality of sandglass.

Please use our issuetracker on bitbucket for both alike.

Contributing Code

If you would like to see a new feature in sandglass, please discuss it on our mailinglist up front so that we don't waste your time by not merging your contribution.

- Any code will be reviewed and tested by at least one core developer. Of course, everyone is welcome to give feedback.
- Code *must* be tested. Please *provide unit-tests* for bugfixes and new features alike.
- Documentation should be adapted to your changes, if relevant. New features of course need new documentation.

Syntax conventions

- For Python code, we try to conform to PEP8 standard. Please use the `pep8` command installed with the development requirements (outlined in *Setting up a Development Environment*) to check for compliance.
- For all other code (HTML, CSS, JS and reStructuredText for documentation) we try at least to follow some rules like:
  - 4 *spaces* indentation, except for reStructuredText where sometimes 4 just does not work.
  - Lines should be 79 characters maximum, except for HTML.

In general, take a look at existing code and try to stick to the conventions you can easily spot there.

Contributing Tests

Tests are essential. Having a comprehensive suite of unit-test and integration tests is most important to us. Contributing good tests earns you extra respect.

In general, tests should be:

- Unitary, if possible. Test only one function/method/class.
- Fast. Just make sure your test does not double the time to run the test-suite for everyone else.

Where Tests should go

- Application-specific (speaking of Django apps) tests should go into the `tests` module of the respective application.

  That applies to all `sandglass.*` apps that are found in the `INSTALLED_APPS` setting.
- Project-specific tests should go to the pseudo-app `tests` in the repository root.

  For example, that also applies to tests for the `sandglass.lib` module.

Running the Tests

To run all our tests you need to install tox, change to the repository's root directory, where the `tox.ini` is located, and invoke the `tox` command:

```
$ sudo pip install tox
$ cd path/to/sandglass/code
$ tox
```

Hint: Since *tox* runs tests with multiple interpreters, you should *not* have any virtualenv activated.

This will automatically run the tests in Python 2.6 and 2.7. Also, the documentations (for users and developers) and *test-coverage reports* will be generated and the Python code will be *checked for pep8 compliance*.

If you want to run the tests for a specific environment only, pass the `-e` option to tox:

```
$ tox -e py27
# or
$ tox -e docs
```

To run only a certain test, you can also invoke tox like this:

```
$ tox -e py27 -- tests.SomeUnitTestClass.test_something
```

Test Coverage

The `py27` test environment automatically generates coverage reports. Thus, to generate the reports, you only need to invoke `tox` or `tox -e py27`.

This generates HTML reports in the `/docs/_coverage/` directory.

Contributing Documentation

Documentation is often considered to be even more important than the code itself. So, contributing docs — just like tests — earns you extra respect.

Please stick to the following rules:

- We use Sphinx and thus reStructuredText, so that's what you should use.
- Everything should be written in plain English.
- Not making any assumptions about the readers. Link to external documentation (for libraries, etc.) and explain even the obvious.

Building the Documentation

If you have Sphinx installed in your virtualenv, you can build the documentation by changing to the respective directory and using the `make` command:

```
(sandglass)$ cd path/to/sandglass/code/
(sandglass)$ cd docs/
(sandglass)$ make clean
(sandglass)$ make html
```

Also, as outlined in *Running the Tests*, you can use tox to build the documentation:

```
$ cd path/to/sandglass/code/
$ tox -e docs
```

4.2 Setting up a Development Environment

As for production environments, Python 2.6 or 2.7 is required to run sandglass. Also, using virtualenv (and virtualenvwrapper, of course) is recommended to create an isolated environment.

Since the code is managed in a Mercurial repository, of course you'll need a working installation of that.

Instructions on how to install these tools can be found in their respective documentation.

To install sandglass for development, follow these steps:

1. Make sure you have the global dependencies installed. Namely, these are:
   - Java
   - YUI Compressor

   You'll find installation instructions on the respective websites.

2. Create a virtualenv (be sure to use distribute instead of setuptools):

```
$ mkvirtualenv -p /usr/bin/python2.7 --no-site-packages --distribute \
   sandglass
```

3. Check out sandglass sourcecode from bitbucket:

```
$ cd path/to/your/projects/
$ hg clone ssh://hg@bitbucket.org/fabianbuechler/sandglass sandglass
$ cd sandglass
```

4. Activate the virtualenv and install sandglass development egg:

```
$ workon sandglass
(sandglass)$ python setup.py develop
```

   This installs sandglass from the code in the current directory. It also gives you the `sandglass` command.

5. Install development-specific dependencies using pip:

```
(sandglass)$ pip install -r requirements.txt
```

   This installs stuff like Sphinx for building documentation or pep8 for checking code syntax, but also little helpers like IPython or the django-debug-toolbar.

6. Create a installation using the `sandglass init` command:

```
(sandglass)$ cd ..
(sandglass)$ sandglass init --template=develop sandglass-dev
(sandglass)$ cd sandglass-dev
```

   The `sandglass init` command uses the deploy template "develop" to create a directory with all files required to run sandglass for you.

7. Adapt basic `settings.py` to fit your needs.

   For information about that check Django's documentation on settings.

   Make sure to set the correct path to your `yuicompressor` binary. Default is:

```
PIPELINE_YUI_BINARY = '/usr/local/bin/yuicompressor'
```

   But for Ubuntu, it might be:

```
PIPELINE_YUI_BINARY = '/usr/bin/yui-compressor'
```

8. Initialize your database:

```
(sandglass)$ python manage.py syncdb
# follow the instructions
(sandglass)$ python manage.py migrate --all
```

9. Start sandglass development server:

   Sandglass is using Django's runserver for development, either use:

```
(sandglass)$ python manage.py runserver
```

   or

```
(sandglass)$ sandglass serve
```

   to start it. Then open your browser at http://localhost:8000/ to check if everything is working correctly.

4.3 Sandglass Development Documentation

Note: These are development notes for the sandglass core developers.

4.3.1 Features

- users / groups / permissions
- companies / workspaces
- clients / projects
- tasks + description
- tagging for tasks (activities) and time-entries
- worktime tracking
  - daily start/end (multiple)
  - weekly target hours
  - yearly holiday budget
  - floating / fixed time models
  - overtime budget
- monthly reports done by employee (user)
  - assigns overtime (tags) or daily
  - review cycle by admin (personal manager?)
  - workinghours entries collected in approved reports are locked
- API internal and external
  - for internal use by all our code
  - RESTful for external apps connecting to the data
- project configuration templates

4.3.2 Data Model

Common Base Models

TimeStampedModel [TSM] (abstract)

Automatically manages `created` and `modified` timestamps. Provided by *django-extensions*.

`created` Object creation date and time.

`modified` Last modification date and time.

Accountable [ACC] (abstract)

Groups common attributes for accountable objects. These are the type of accounting (by effort / lump-sum), the status, the estimated costs and the budgeted hours.

accounting_type Accounting by `effort` or `lump-sum`. [enum]

hours_budget Estimated effort for this accountable in hours. [float]

cost_estimate Estimated costs for this accountable in `currency`. [float]

currency Currency for all related amounts. Defaults to `settings.SANDGLASS_DEFAULT_CURRENCY`. [string]

ActivityPeriod [AP] (abstract)

Groups common attributes for projects with a certain activity period.

active Defines if new hours can be booked onto the object. [boolean]

active_from Start date of activity period. [date]

active_to End date of activity period. [date]

User-Management

User

The main application user or company employee. Provided by `django.contrib.auth`.

username Unique username. [string]

first_name [string]

last_name [string]

email [string]

password Encrypted password. [string]

last_login Date and time of last login. [datetime]

... and some more

UserProfile

Inherits from *TSM*.

An extension to *User* that adds sandglass-specific user properties to the `User` model.

user Link to the *User* model. [1to1]

failed_logins Counter for account locking after n failed logins. [int]

The number of failed login attempts that locks an account can be specified in a global configuration variable.

date_locked When was the account locked. [timestamp]

teams Reference to *Teams* including this user. [M2M]

Note: This is different from `User.groups` (to `django.contrib.auth.models.Group`) which is for grouping permissions. Teams are solely for grouping users.

---

status_message Optional additional message describing the user's status. [string]

time_model Reference to the *TimeModel* assigned to this user. [FK]

date_entry Date the employee started working for the company. [date]

avatar Avatar image for this user. Maybe rather use something like django-avatar. [image/FK?]

theme Select sandglass styling theme (colors, fonts, background image, ...). Reference to the *Theme* selected by this user. [FK]

Contact information Basic contact info like

- phone numbers
- additional email addresses
- work- and home addresses
- room number
- ...

Note: Currently only add the fields we need. Maybe extend the model later in a flexible way using an entity-attribute-value model.

Team

Inherits from *TSM*.

Group of users/employees in terms of workgroup or department, not permissions.

Note: Might better be replaced by `django.contrib.auth.models.Group`.

---

name Team name. [string]

description Optional short description of the team's purpose. [string]

users Back-reference to *Users* in this team. [M2M]

TimeModel

Inherits from *TSM*.

Defines a set of „rules" the work-time of *employees*. TimeModels can be reused for multiple users and should be defined by administrators.

workdays_per_week Number of workdays per week. Defaults to five. [integer]

daily_hours Hours the employee should do daily. The standard workday.

If daily hours should be defined on a daily basis, set `daily_hours=None` and use `daily_hours_<weekday>` instead.

Defaults to `weekly_hours ÷ workdays_per_week`. [float/time?]

E.g. 38.5 hours per week ÷ 5 workdays = 7.7 hours per day.

daily_hours_<weekday> Daily hours for `weekday` (mon-sun) if defined on daily basis.

Defaults to `None`. [float/time?]

weekly_hours Hours an employee should do weekly. The standard work-week.

Should be the product of `workdays_per_week × daily_hours` or the sum of all `daily_hours_<weekday>` values. [float/time?]

yearly_vacation The number of vacation days the employee may consume per year. [int]

In combination with all *Activity* entryies for the user, `activity_type == vacation`, and the entry date (`UserProfile.date_entry`), this can be used to calculate the employee's vacation quota for any date (without using the TimeSheet savepoints).

Example: the user has 25 days per year of vacation and started working for the company on the 17th of April 2011.

The vacation quota for the first year is calculated like this:

```
import calendar
import datetime
yearly_vacation = 25
entry_date = datetime.date(2011, 4, 17)
# vacation quota for year of entry
day_of_year = entry_date.timetuple().tm_yday  # 107
days_in_year = 366 if calendar.isleap(entry_date.year) else 365  # 365
entry_quota = int(round(yearly_vacation * day_of_year / days_in_year))
# 7 days vacation in the first year
```

Then, let's say the vacation history evolved that way (not taking weekends into account):

- 2011-04-17: +7 days (initial)
- 2011-12-04 - 2011-12-08: -5 days
- 2012-01-01: +25 days (yearly budget 2012)
- 2012-03-03: -1 day
- 2012-06-10 - 2012-06-23: -15 days

So, the current quota for 2012-06-24 is `7 - 5 + 25 - 1 - 15 == 11`.

Generically, for any date, the current vacation quota calcuates as:

```
# number of years between date_entry and date
date = datetime.date(2012, 6, 24)
nr_years = date.year - entry_date.year  # 1
# nr of days added to the quota in the meanwhile
quota_add = nr_years * yearly_vacation  # 25
# consumed days from Activity, type == vacation before date
quota_sub = Activity.objects.filter(
    user__pk=id,
    activity_type='vacation',
    end__lte=date).count()  # e.g. 21
# current vacation quota
vacation_quota = entry_quota + quota_add - quota_sub  # 11
```

timesheet_interval Length of the usual interval in which the employee should do settlements. [choice]

One of weekly, bi-weekly, monthly, quarter-yearly, half-yearly, yearly.

day_start_from Begin of the time-range an employee should start his/her workday at. [time]

If workday start-times should be defined on a daily basis, set `day_start_from` and `day_start_to` to `None` and use `day_start_<weekday>_<from/to>` instead.

day_start_to End of the time-range an emplyee should start his/her workday at. [time]

day_end_from Begin of the time-range an employee should end his/her workday at. [time]

day_end_to End of the time-range an employee should end his/her workday at. [time]

day_start_<weekday>_from Like `day_start_from` for `weekday` (mon-sun) if defined on a daily basis. [time]

day_start_<weekday>_to Like `day_start_to` for `weekday` (mon-sun) if defined on a daily basis. [time]

day_end_<weekday>_from Like `day_end_from` for `weekday` (mon-sun) if defined on a daily basis. [time]

day_end_<weekday>_to Like `day_end_to` for `weekday` (mon-sun) if defined on a daily basis. [time]

Project-Structure

Client

Inherits from *TSM*.

Client (company or person) of the company comissioning at least one project. This does not link to a

name Client's name or company name. [string]

avatar Avatar image for this user. Maybe rather use something like django-avatar. [image/FK?]

contact_persons Reference to *contact persons* for this client. These are general contacts, without relation to a specific project for this client.

ContactPerson

Inherits from *TSM*.

Generic set of contact information of a person working for a client in general or on a specific project.

first_name [string]

last_name [string]

email [string]

phone [string]

Further contact information

- additional phone numbers
- additional email addresses
- address
- room number

- department, position

Note: Currently only add the fields we need. Maybe extend the model later in a flexible way using an entity-attribute-value model.

Project

Inherits from *TSM* and *ACC*.

Defines one project for a ceratain client (or internal, for the company itsself), that defines a list of tasks (project parts) available to book hours to.

Projects can be cloned to easily create new projects with the same attributes.

name Project title. [string]

acronym Project title acronym and keyboard shortcut. [string]

parent Self-reference for grouping purposes. Limit to one level. [FK]

client Reference to the contracting *Client*. [FK]

If no client is defined, the project is company-internal.

tasks List of tasks defined for this project. [M2M]

activity_description_required Flag indicating if any *Activity* assigned to this project requires a description. [boolean]

avatar Avatar image for this user. Maybe rather use something like django-avatar. [image/FK?]

contact_persons Reference to *contact persons* for this client. These are project-specific contact persons.

Task

Inherits from *TSM* and *ACC*.

Main activity categorization for projects. Describes project phases, parts and areas.

name Task title. [string]

acronym Task title acronym and keyboard shortcut. [string]

parent Self-reference for grouping purposes. Limit to one level. [FK]

default_tags List of default *tags* for any *Activity* recorded for this project. [M2M]

Tracking

Activity

Inherits from *TSM*.

A record of activity. This is the main unit of measure for sandglass. Activities can describe either work on a project, unassigned time, breaks or days where employees are away sick or on vacation.

user Reference to the *User* this activity is recorded for. [FK]

project Reference to the *Project* this activity is recorded for. Can be `None` whenever the activity is not project related. Required for project-related tasks (`type == working`). [FK]

task Reference to the *Task* this activity is recorded for. Can be `None` whenever the activity is not project related. Required if `type == working` and any tasks are defined for the assigned `project`. [FK]

description Usually optional description of the activity. Can be mandatory if `project.activity_description_required` is `True`.

activity_type Indicator of the kind of activity described. [enum] Can be one of:

- working
- out of office
- unassigned
- break
- on vacation
- sick
- on leave

start Start of activity. [datetime]

end End of activity. `None` if still running. [datetime]

duration Duration of the activity in days. [int]

Denormalized difference between `end` and `start` for faster retrieval of the number of days this activity must be accounted for. Must be updated in `Activity.save`.

tags List of assigned *tags*. [M2M]

locked Flag indicating if this activity has already been reviewed in some report. [boolean]

Note: Potential problem with project-time settlement reports (for invoicing) and employees' monthly timesheets (for payroll accounting). We might need different lock states here.

Tag

Inherits from *TSM*.

name Tag title. [string]

acronym Tag title acronym and keyboard shortcut. [string]

description [string]

tag_type Indicator of the kind of tag. [enum] Can be one of:

- `activity`

  Tagging for *activities* regarding the type of activity. Examples are design, development, templating, frontend, testing, consulting, presentation, coordination, meetings, ...

- `accounting`

  Tagging for *activities* regarding the price of the activity. Examples are editorial work, development, senior consulting, ...

- `time`

  Tagging for *activities* regarding the type of time the activity was carried out at. Examples are normal work-time, overtime, business trips, assembling, ...

`hourly_price` Hourly price for each tagged activity in `currency`. [float]

`currency` Currency for all related amounts. Defaults to `settings.SANDGLASS_DEFAULT_CURRENCY`. [string]

OverTime

Inherits from *TSM*.

A record of overtime. This is an additional tracking unit, next to *Activity*, which is used to keep track of employees' overtime budget.

OverTime, not to be confused with the employees' *flexible working hours balance*, needs to be **kept track of manually**.

For time-models with flexible working hours, the daily, weekly or monthly balance can easily be calculated for any date by summing the hours done in the period and balancing that with the `daily_hours`, `weekly_hours` or `daily_hours × days_in_month` (defined in the *employee's time-model*) respectively.

OverTime, on the other hand, is subject to *legal constraints* and *contractual agreements*. As such, there is no generically valid way of determining if any given activity is to be seen as overtime or not.

By analyzing the *employee's time-model*, sandglass will try to *suggest overtime periods* which the user can accept or refuse.

Employees can also *consume overtime* by requesting payout in their montly timesheets or manually compensating their flexible working hours balance.

`user` Reference to the *User* who this overtime is accounted for. [FK]

`date` Day of overtime activity. [date]

`factor` Percentual additional value of the overtime hours. [float/decimal]

Usual values would be 0.5 (50%, 1:1,5) or 1 (100%, 1:2).

`duration` Length of overtime period. [timedelta]

Negative values can be provided as a matter of consuming ones overtime budget.

`activities` Optional link to the *activities* that are affected. [M2M]

Reporting

TimeSheet

Inherits from *TSM*.

Montly (or weekly, bi-weekly, ...) settlement of the users' activities. This acts as basis for salary calculations and as savepoint of the users' status.

The employees' vacation budget, flexible working-hours balance, overtime budget and the like are stored here and thus the development of those values in time is easily reproducable.

Any calculations should always use the last verified timesheet of a user as basis.

`user` Reference to the *User* this timesheet is about. [FK]

name Title of the timesheet. [string]

Will be something like "Montly timesheet, January 2012, Name".

description Description of the timesheet's purpose. [string]

date_from Start of date range limit for the timesheet. [datetime]

date_to End of date range limit for the timesheet. [datetime]

time_model_snapshot Serialized version of the *TimeModel* that was valid for a *User* at the time the TimeSheet was created.

verified Flag to indicate if the report has been verified by a supervisor. [boolean]

flextime_payout Amount of time, the *User* chooses to be paid out from his/her flextime budget. [timedelta]

overtime_payout Amount of time, the *User* chooses to be paid out from his/her overtime budget. [timedelta]

vacation_budget The employee's vacation budget from now on. [int]

This value is valid for the period *after* the TimeSheet.

flextime_balance The employee's flextime budget from now on. [timedelta]

If the employee does not have a flextime model, this is optional.

This value is valid for the period *after* the TimeSheet.

overtime_budget The employee's overtime budget from now on. [timedelta]

This value is valid for the period *after* the TimeSheet.

ReportQuery

Inherits from *TSM*.

Saved filter configuration (user, team, client, project, ...) for generating a custom report.

name Title of the report. [string]

description Description of the report's purpose. [string]

date_from Start of date range limit for the report. [datetime]

date_to End of date range limit for the report. [datetime]

users List to *users* to limit the report to. [M2M]

teams List of *teams* to limit the report to. [M2M]

clients List of *clients* to limit the report to. [M2M]

projects List of *projects* to limit the report to. [M2M]

include_sub_projects Should children of the projects be included? [boolean]

tasks List of *tasks* to limit the report to. [M2M]

include_sub_tasks Should children of the tasks be included? [boolean]

tags List of *tags* to limit the report to. [M2M]

ReportSnapshot

Inherits from *TSM*.

Serialized report data stored for archiving purposes. No whatsoever connection to live data is kept in the snapshots. All required data is serialized so that changes don't affect the report output.

`name` Title of the report. [string]

`description` Description of the report's purpose. [string]

Serialized data The same fields as `ReportQuery` but with serialized lists of objects. Serialization format is JSON.

ReviewedReport

Inherits from *TSM*.

Combination of *ReportQuery* and *ReportSnapshot* for reports which need to be verified by a supervisor.

`report_query` Reference to a *ReportQuery* which will be used before the report is `verified`. [FK]

`report_snapshot` Reference to a *ReportSnapshot* which will be used after the report was `verified`. [FK]

`verified` Flag to indicate if the report has been verified by a supervisor. [boolean]

Miscellaneous

PublicHoliday

Synchronization storage for ical feeds providing public holiday data.

`name` Name of the holiday (e.g. Christmas)

`date` [date]

Settings

Theme

Inherits from *TSM*.

Everyone likes other colors best. So let's make some themes!

`name` Name of the theme.

`description` Short description of the theme. [string]

`theme_folder` Identifier / path-prefix for theme CSS and images. (e.g. located in `static/sandglass/themes/<theme_folder>/theme.css|background.png`).

4.3.3 Requirements by WIENFLUSS

Author Michael Stenitzer <stenitzer@wienfluss.net>

Structures and Administration

Project-Assignment

- Clients
  - Projects
    - Phases
      - Tasks

A template-system for phases including the corresponding tasks would be great. These templates could be assigned to projects.

Project-time tracking

- User (required)
- Date (required)
- Duration (required)
- Project / phase / task (required in deepest available level [1])
- Text (optional / required [2])
- Working-time class (e.g. normal, overtime, business trip, assembling, ...)
- Accounting class (hourly rates)

Working-time tracking

There is maximum one working-time entry per day, interrupted by any number of breaks.

- Working-time: start, end (required)

  Attention: the end time can fall to the next calendar day. Should the first break during the next day automatically end the working-time entry for the current day?

- Breaks: start, end
- Duration vacation times
- Duration illness, consultation, official tasks, special leaves, special holidays (by religion), ...
- Working-time calendar [3] (workday, weekend, holiday, ...)

[1]
- if no phases and tasks are defined, assignment on project level is sufficient.
- if phases and tasks are defined, selecting them is mandatory.

[2] depends on project settings
[3] can be managed in a calendar

Working-time classes

Defined by the administrator, optional.

- Types (normal, overtime, business trip, assembling, ...)
- Description
- ?

Accounting classes

Defined by the administrator, optional.

- Types (hourly rates or similar)
- Description
- ?

Working-time calendar

- Types defined on a daily basis: workday, weekend, holiday
- possibly in iCal or similar standard format
- possibly defined by rules
- export/import functionality
- manageable by the administrator

Clients

Defined by the administrator, optional.

- Name
- ?

Projects

Defined by the administrator, required.

- Short name
- Name
- Status: active, inactive
- Type: internal, external, ...
- Accounting: by effort, lump-sum
- Description required? (maybe inherited)
- User constraints
- Possible further properties:

  - flag: support billing on a regular basis
  - end-date
  - accounting period
  - target date for projects / phases / ...

Phases

Defined by the administrator, optional.

- Name
- Status: active, inactive
- Accounting: by effort, lump-sum
- Description required?
- User constraints

Tasks

Defined by the administrator (?), optional.

- Name
- Status: active, inactive
- Accounting: by effort, lump-sum
- Description required?
- User constraints

User Management

- ???

Optional features:

- Target working-times (per workday)

  Attention: for some employment statuses this might depend on th weekday.

- Vacation quota
- Overtime budget

Interfaces

The entry interfaces should complement each other and be usable by personal likings or current needs.

General requirements:

- learning: current entry values (last used projects, ...) are automatically proposed by the system
- flexible: the system can be used on different interfaces (e.g. mobile)

- very good keyboard-, mouse- and touchscreen-support: shortcuts for keyboard-users, least possible keyboard usage for mobile users
- visual feedback for current status: current recording of working-time or break; project-times already assigned to the workday?
- the different interfaces (time-tracking, attendance-monitoring, ...) should be interweaved synergetically.

Time-tracking

The user captures his working-times and project-times (usually ex-post) via an online interface. The capturing is for the current day by default but optionally also for any past days.

Project-time tracking

- Select project, phase and task
- Assign time
  - possibly in different time units (quarters of an hour, ...)
  - via a spinbox (+/-)
- Textual description
- Set further optional properties

Working-time tracking

- Start- and end-time
- Breaks: start- and end-time
- Special times

Tracking of special times

User captures times of absence for longer periods solely for workdays (e.g. vacations or illnesses).

- Start- and end-date
- Hours per day

Time recording

During a workday a user records working-times, breaks and project-times directly. Implementation via an online interface or optionally with an app (desktop or mobile).

- Start / stop / break for working-times
- Start / stop for project-times
- Direct editing / correction of recorded times is possible

Attendence monitoring

Companies / departments can view the current attendance status of employees via an online interface.

- Static overview of all employees
  - where applicable with additional information: name, initials, phone, room
  - where applicable with optional information of the user
- Monitoring
  - not yet attendant
  - working
  - on break
  - closing time
  - absent (including optional message)
- Batch capturing of absence-messages

Reporting

Use-cases:

- Working-time controlling
  - evaluation of employees' working-times for arbitrary periods
  - use by employees and administrators
- Working-time settlement
  - goal is the evaluation and settlement of employees' working-times for one month for the payroll accounting
  - ideally working-time entries for an already evaluated period cannot be edited anymore
  - use by administrators
- Project-time controlling
  - evaluation of project-times for single projects for arbitrary periods and arbitrary employees
  - use by employees and administrators
- Project-time controlling overview
  - aggregation of captured project-times for all active projects (possibly grouped by phases)
  - use by administrators
- Project-time settlement
  - goal is the evaluation and settlement of project-times for arbitrary periods
  - ideally project-time entries for an already evaluated period cannot be edited anymore
  - it should be obvious which periods have already been evaluated
  - use by administrators

Questions & Considerations

- Do we need a possibility to save certain queries / reports?
- Do we need a possibility to set evaluation-points for certain queries / reports (reminder)?

Possible interfaces:

- Project-accounting, invoicing
- Reminder for project-settlements

CHAPTER

FIVE

CHANGELOG

5.1 Changelog

5.1.1 0.1 - unreleased

-

PYTHON MODULE INDEX

S