Timetracker Documentation

Release 1

Aaron France

May 20, 2012

CONTENTS

CHAPTER

ONE

CONTENTS:

1.1 Dependencies

1.1.1 Python specific

- Python v2.7
- Django v1.4
- simplejson (included with Django)
- MySQLdb Python MySQL bindings

1.1.2 Misc

- Apache+mod_wsgi
- Any SMTP Server
- MySQL Server

1.1.3 Detailed Instructions

Below will outline all the required steps to prepare your system.

1.1.4 Install Python

If you're using Windows, this will simply be a case of heading to the main Python website and downloading the Python 2.7 binary and installing. It is extremely vital that this particular version of Python is used otherwise, the later (and some earlier) versions are code-incompatible. Either through syntax differences or some modules are not available.

If you're on Linux, any debian-based system will have the python2 package in it's repos. Most distros will have the package required.

1.1.5 Install Django

If you're using Windows, head to the Django website and download and unzip the 1.4 source then follow the below:

```
cd django1.4
python setup.py install
```

Other than taking an inordinate amount of time. This will then install Django 1.4 onto your system.

If you're on Linux, please refer to the available packages in your repos to find the exact name of the package. However, the steps will include:

```
for Arch:
sudo pacman -S python-django
Debian-based:
sudo apt-get install python-django
```

1.1.6 simplejson

This should come with django, but, for whatever reason you wish to use an externally sourced version, please go to Google and download the latest via any links found on there.

1.1.7 MySQLdb Python bindings

This part is actually a little tricky for Windows. Therefore we have decided to link to a pre-compiled binary for the MySQLdb Python bindings. It can be found here.

If you're on Linux, please refer to your distro's repos for what the package name is, however, some hints are below:

```
on Arch:
sudo pacman -S mysql-python
Debian-based:
sudo apt-get install mysql-python
```

This covers the installation portion of the code-dependencies.

Next is preparing your system for the software which needs to be installed.

1.1.8 Apache

Windows:

Head to the Apache website and download the Apache 2.2 binary and install.

Linux:

```
On Arch:
sudo pacman -S apache
Debian based:
sudo apt-get install apache
```

1.1.9 mod_wsgi

Download the latest mod_wsgi.so file from the mod_wsgi downloads page found, on here.

Then, depending on the system you are using, put it into your Apache modules directory.

Windows:

```
C:\Program Files\Apache Software Foundation\Apache2.2\bin\
```

Linux:

```
/etc/httpd/modules
```

Then for both, modify your httpd.conf file so that it enabled the mod_wsgi.so module:

```
LoadModule wsgi_module <path_to_modules_dir>/mod_wsgi.so
```

1.2 Code

1.2.1 Basic Server Settings module

This module is used for the DJANGO_SETTINGS_MODULE when the development server is being used. This is because there are different levels of logging and the location of the logs is completely different from when the production server is being used.

Another reason is that usually (and you should be) the production server is being run as a daemon user (http user in the case of Apache) and this causes problems for certain portions of code, particularly logging or anything that requires write access to the filesystem, thus, the separation of production settings and the development one.

TODO: Create a base settings module and import attributes from there, overriding when we need to.

1.2.2 Apache Settings module

This module is used for the DJANGO_SETTINGS_MODULE for when apache is being used. This is because there are different levels of logging when using the dev server and when using the apache server. They also are running under different user profiles and therefore have different filesystem access rights. On Windows this isn't a problem but on Linux it makes it a lot easier to use two separate settings files.

TODO: Create a base settings module and import attributes from there, overriding when we need to.

platform All

synopsis Module which contains settings for running the app on Apache

1.2.3 timetracker.views

Views which are mapped from the URL objects in urls.py

platform All

synopsis Module which contains view functions that are mapped from urls

`timetracker.views.`**`add_change_user`**(*request*, ***kwargs*)

Creates the view for changing/adding users

This is the view which generates the page to add/edit/change/remove users, the view first gets the user object from the database, then checks it's user_type. If it's an administrator, their authorization table entry is found then used to create a select box and it's HTML markup. Then pushed to the template. If it's a team leader, their manager's authorization table is used instead.

Parameters **request** – Automatically passed contains a map of the httprequest

Returns HttpResponse object back to the browser.

`timetracker.views.`**`admin_view`**(*request*, ***kwargs*)

This view checks to see if the user logged in is either a team leader or an administrator. If the user is an administrator, their authorization table entry is found, iterated over to create a select box and it's HTML markup,

sent to the template. If the user is a team leader, then *their* manager's authorization table entry is found and used instead. This is to enable team leaders to view and edit the team in which they are on but also make it so that we don't explicitly have to duplicate the authorization table linking the team leader with their team.

Parameters **request** – Automatically passed contains a map of the httprequest

Returns HttpResponse object back to the browser.

timetracker.views.**ajax**(*request*)

Ajax request handler, dispatches to specific ajax functions depending on what json gets sent.

Any additional ajax views should be added to the ajax_funcs map, this will allow the dispatch function to be used. Future revisions could have a kind of decorator which could be applied to functions to mutate some global map of ajax dispatch functions. For now, however, just add them into the map.

The idea for this is that on the client-side call you would construct your javascript call with something like the below (using jQuery):

```
$.ajaxSetup({
    type: 'POST',
    url: '/ajax/',
    dataType: 'json'
});

$.ajax({
    data: {
        form: 'functionName',
        data: 'data'
    }
});
```

Using this method, this allows us to construct a single view url and have all ajax requests come through here. This is highly advantagious because then we don't have to create a url map and construct views to handle that specific call. We just have some server-side map and route through there.

The lookup and dispatch works like this:

1. Request comes through.
2. Request gets sent to the ajax view due to the client-side call making a request to the url mapped to this view.
3. The form type is detected in the json data sent along with the call.
4. This string is then pulled out of the dict, executed and it's response sent back to the browser.

Parameters **request** – Automatically passed contains a map of the httprequest

Returns HttpResponse object back to the browser.

timetracker.views.**edit_profile**(*request*, **args*, ***kwargs*)

View for sending the user to the edit profile page

This view is a simple set of fields which allow all kinds of users to edit pieces of information about their profile, currently it allows uers to edit their name and their password.

Parameters **request** – Automatically passed contains a map of the httprequest

Returns HttpResponse object back to the browser.

timetracker.views.**explain**(*request*, **args*, ***kwargs*)

Renders the Balance explanation page

This page renders a simple template to show the users how their balance is calculated. This view takes the user object, retrieves a couple of fields, which are user.shiftlength and the associated values with that datetime objects, constructs a string with them and passes it to the template as the users 'shiftlength' attribute. It then takes the count of working days in the database so that the user has an idea of how many days they have tracked altogether. Then it calculates their total balance and pushes all these strings into the template.

Parameters **request** – Automatically passed contains a map of the httprequest

Returns HttpResponse object back to the browser.

timetracker.views.**forgot_pass**(*request*)

Simple view for resetting a user's password

This view has a dual function. The first function is to simply render the initial page which has a field and the themed markup. On this page a user can enter their e-mail address and then click submit to have their password sent to them.

The second function of this page is to respond to the change password request. In the html markup of the 'forgotpass.html' page you will see that the intention is to have the page post to the same URL which this page was rendered from. If the request contains POST information then we retrieve that user from the database, construct an e-mail based on that and send their password to them. Finally, we redirect to the login page.

Parameters **request** – Automatically passed contains a map of the httprequest

Returns HttpResponse object back to the browser.

timetracker.views.**holiday_planning**(*request*, **args*, ***kwargs*)

Generates the full holiday table for all employees under a manager

First we find the user object and find whether or not that user is a team leader or not. If they are a team leader, which set a boolean flag to show the template what kind of user is logged in. This is so that the team leaders are not able to view certain things (e.g. Job Codes).

If the admin/tl tries to access the holiday page before any users have been assigned to them, then we just throw them back to the main page. This is doubly ensuring that they can't access what would otherwise be a completely borked page.

Parameters **request** – Automatically passed contains a map of the httprequest

Returns HttpResponse object back to the browser.

timetracker.views.**index**(*request*)

This function serves the base login page. TODO: Make this view check to see if the user is already logged in and if so, redirect.

This function shouldn't be directly called, it's invocation is automatic

Parameters **request** – Automatically passed. Contains a map of the httprequest

Returns A HttpResponse object which is then passed to the browser

timetracker.views.**login**(*request*)

This function logs the user in, directly adding the session id to a database entry. This function is invoked from the url mapped in urls.py. The url is POSTed to, and should contain two fields, the use_name and the pass word field. This is then pulled from the database and matched against, what the user supplied. If they match, the user is then checked to see what *kind* of user their are, if they are ADMIN or TEAML they will be sent to the administrator view. Else they will be sent to the user page.

This function shouldn't be directly called, it's invocation is automatic from the url mappings.

Parameters **request** – Automatically passed. Contains a map of the httprequest

Returns A HttpResponse object which is then passed to the browser

timetracker.views.**logout**(*request*)
Simple logout function

This function will delete a session id from the session dictionary so that the user will need to log back in order to access the same pages.

Parameters **request** – Automatically passed contains a map of the httprequest

Returns A HttpResponse object which is passed to the browser.

timetracker.views.**user_view**(*request*, **args*, ***kwargs*)
Generates a calendar based on the URL it receives. For example: domain.com/calendar/{year}/{month}/{day}, also takes a day just in case you want to add a particular view for a day, for example. Currently a day-level is not in-use.

Note The generated HTML should be pretty printed

Parameters

- **request** – Automatically passed contains a map of the httprequest
- **year** – The year that the view will be rendered with, default is the current year.
- **month** – The month that the view will be rendered with, default is the current month.
- **day** – The day that the view will be rendered with, default is the current day

Returns A HttpResponse object which is passed to the browser.

1.2.4 timetracker.models

Definition of the models used in the timetracker app

platform All

synopsis Module which contains view functions that are mapped from urls

class timetracker.tracker.models.**Tblauthorization**(**args*, ***kwargs*)
Links Administrators (managers) with their team.

This table is a many-to-many relationship between Administrators and any other any user type in TEAML/RUSER.

This table is used to explicitly show which people are in an Administrator's team. Usually in SQL-land, you would be able to instanstiate multiple rows of many-to-many relationships, however, due to the fact that working with these tables in an object orientated fashion is far simpler adding multiple relationships to the same `Tblauthorization` object, we re-use the relationship when creating/adding additional `Tblauthorization` instances.

This means that, if you were to need to add a relationship between an Administrator and a RUSER, then you would need to make sure that you retrieve the `Tblauthorization` object *before* and save the new link using that instance. Failure to do this would mean that areas where the .get() method is employed would start to throw Tblauthorization.MultipleObjectsReturned.

In future, and time, I would like to make it so that the .save() method is overloaded and then we can check if a `Tblauthorization` link already exists and if so, save to that instead.

display_users()
Method which generates the HTML for the admin views

This method is depracated in favour of not actually using the admin interface to interact with `Tblauthorization` instances too much. That and, it's not unicode-safe.

Return type `string`

manager_view()
Method which negates needing to retrieve the users via the Tblauthorization.objects.all() method which is, needless to say, a mouthfull.

Return type QuerySet

teamleader_view()
Method which provides a shortcut to retrieving the set of users which are available to the TeamLeader based upon the rules of what they can access.

Return type QuerySet

class timetracker.tracker.models.**Tbluser**(**args*, ***kwargs*)
Models the user table and provides the admin interface with the niceties it needs.

This model is the central pillar to this entire application.

The permissions for a user is determined in the view functions, not in a table this is a design choice because there is only a minimal set of things to have permissions *over*, so it would be overkill to take full advantage of the MVC pattern.

User

The most general and base type of a User is the *RUSER*, which is shorthand (and what actually gets stored in the database) for Regular User. A regular user will only be able to access a specific section of the site.

Team Leader

The second type of User is the *TEAML*, this user has very similar level access as the administrator type but has only a limited subset of their access rights. They cannot have a team of their own, but view the team their manager is assigned. They cannot view and/or change job codes, but they can create new users with all the *other* information that they need. They can view/create/add/change holidays of themselves and the users that are assigned to their manager.

Administrator

The third type of User is the Administrator/*ADMIN*. They have full access to all functions of the app. They can view/create/change/delete members of their team. They can view/create/add/change holidays of all members of their team and themselves. They can create users of any type.

display_user_type()
Function for displaying the user_type in admin.

Note This method shouldn't be called directly.

Return type string

get_administrator()
Returns the Tbluser who is this instances Authorization link

Returns A Tbluser instance

Return type Tbluser

get_holiday_balance(*year*)
Calculates the holiday balance for the employee

This method loops over all TrackingEntry entries attached to the user instance which are in the year passed in, taking each entries day_type and looking that up in a value map.

Values can be:

1. Holiday: Remove a day
2. Work on Public Holiday: Add two days

2.Return for working Public Holiday: Remove a day

Parameters **year** (int) – The year in which the holiday balance should be calculated from

Return type Integer

get_total_balance(*ret='html'*)
Calculates the total balance for the user.

This method iterates through every TrackingEntry attached to this user instance which is a working day, multiplies the user's shiftlength by the number of days and finds the difference between the projected working hours and the actual working hours.

The return type of this function is different depending on the argument supplied.

Note To customize how the CSS class is determined when using the html mode you will need to change the ranges in the tracking_class_map attribute.

Parameters **ret** – Determines the return type of the function. If it is not supplied then it defaults to 'html'. If it is 'int' then the function will return an integer, finally, if the string 'dbg' is passed then we output all the values used to calculate and the final value.

Return type string or integer

is_admin()
Returns whether or not the user instance is an admin type user. The two types of 'admin'-y user_types are ADMIN and TEAML.

Return type boolean

name()
Utility method for returning users full name. This is useful for when we are pretty printing users and their names. For example in e-mails and or when we are displaying users on the front-end.

Return type string

tracking_entries(*year=2012*, *month=5*)
Returns all the tracking entries associated with this user.

This is particularly useful when required to make a report or generate a specific view of the tracking entries of the user.

Parameters

- **year** – The year in which the QuerySet should be filtered by. Defaults to the current year.
- **month** – The month in which the QuerySet should be filtered by. Defaults to the current month.

Return type QuerySet

class timetracker.tracker.models.**TrackingEntry**(**args*, ***kwargs*)
Model which is used to enter working logs into the database.

A tracking entry consists of several fields:-

1.Entry date: The date that the working log happened.

2.Start Time: The start time of the working day.

3.End Time: The end time of the working day.

4.Breaks: Any breaks taken during that day.

5.Day Type: The type of working log.

Again, the TrackingEntry model is a core component of the time tracking application. It directly links users with the time-spent at work and the the type of day that was.

1.2.5 timetracker.utils.calendar_utils

Module for collecting the utility functions dealing with mostly calendar tasks, processing dates and creating time-based code.

`timetracker.utils.calendar_utils.`**`ajax_add_entry`**(*request*)
Adds a calendar entry asynchronously

`timetracker.utils.calendar_utils.`**`ajax_change_entry`**(*request*)
Changes a calendar entry asynchronously

`timetracker.utils.calendar_utils.`**`ajax_delete_entry`**(*request*)
Asynchronously deletes an entry

`timetracker.utils.calendar_utils.`**`ajax_error`**(*request*)
Returns a HttpResponse with JSON as a payload with the error code as the string that the function is called with

`timetracker.utils.calendar_utils.`**`calendar_wrapper`**(*function*)
Decorator which checks if the calendar function was called as an ajax request or not, if so, then the the wrapper constructs the arguments for the call from the POST items

Parameters **function** – Literally just gen_calendar.

Return type Nothing directly because it returns gen_calendar's

`timetracker.utils.calendar_utils.`**`delete_user`**(*request*)
Asynchronously deletes a user

`timetracker.utils.calendar_utils.`**`gen_calendar`**(**args*, ***kwargs*)
Returns a HTML calendar, calling a database user to get their day-by-day entries and gives each day a special CSS class so that days can be styled individually.

The generated HTML should be 'pretty printed' as well, so the output code should be pretty readable.

`timetracker.utils.calendar_utils.`**`gen_datetime_cal`**(*year*, *month*)
Generates a datetime list of all days in a month

`timetracker.utils.calendar_utils.`**`gen_holiday_list`**(*admin_user*, *year=2012*, *month=5*)
Outputs a holiday calendar for that month.

For each user we get their tracking entries, then iterate over each of their entries checking if it is a holiday or not, if it is then we change the class entry for that number in the day class' dict. Adds a submit button along with passing the user_id to it.

Parameters

- **admin_user** – `timetracker.tracker.models.Tbluser` instance.
- **year** – `int` of the year required to be output, defaults to the current year.
- **month** – `int` of the month required to be output, defaults to the current month.

Returns A partially pretty printed html string.

Return type `str`

`timetracker.utils.calendar_utils.`**`get_request_data`**(*form*, *request*)
Given a form and a request object we pull out from the request what the form defines.

i.e.:

```
form = {
    'data1': None
}
```

get_request_data(form, request) will then fill that data with what's in the request object.

Parameters

- **form** – A dictionary of items which should be filled from
- **request** – The request object where the data should be taken from.

Returns A dictionary which contains the actual data from the request.

Return type `dict`

`timetracker.utils.calendar_utils.`**`get_user_data`**(*request*)
Returns a user as a json object

`timetracker.utils.calendar_utils.`**`mass_holidays`**(*request*)
Adds a holidays for a specific user en masse

`timetracker.utils.calendar_utils.`**`parse_time`**(*timestring*, *type_of=<type 'int'>*)
Given a time string will return a tuple of ints, i.e. "09:44" returns [9, 44] with the default args, you can pass any function to the type argument.

Parameters

- **timestring** – String such as '09:44'
- **type_of** – A type which the split string should be converted to, suitable types are: `int`, `str` and `float`.

`timetracker.utils.calendar_utils.`**`profile_edit`**(*request*)
Asynchronously edits a user's profile

`timetracker.utils.calendar_utils.`**`useredit`**(*request*)
Adds a user to the database asynchronously

`timetracker.utils.calendar_utils.`**`validate_time`**(*start*, *end*)
Validates that the start time is before the end time

Parameters

- **start** – String time such as "09:45"
- **end** – String time such as "17:00"

Return type `boolean`

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

t

INDEX

U

V