Timetracker Documentation

Release 1

Aaron France

May 23, 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 Settings files

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

1.3 Timetracker API and code

Here I will outline the API for the timetracker application and any advice I have pertaining to the inner-workings of the code itself.

1.3.1 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.

1.3.2 timetracker.utils.calendar_utils

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

Module Functions

`get_request_data()`	`calendar_wrapper()`
`validate_time()`	`gen_holiday_list()`
`parse_time()`	`ajax_add_entry()`
`ajax_delete_entry()`	`ajax_error()`
`ajax_change_entry()`	`get_user_data()`
`delete_user()`	`useredit()`
`mass_holidays()`	`profile_edit()`
`gen_datetime_cal()`	

`timetracker.utils.calendar_utils.`**`ajax_add_entry`**`(`*request*`)`

Adds a calendar entry asynchronously.

This method is for RUSERs who wish to add a single entry to their TrackingEntries. This method is only available via ajax and obviously requires that users be logged in.

The client-side code which POSTs to this view should contain a json map of, for example:

```
json_map = {
    'entry_date': "2012-01-01",
    'start_time': "09:00",
    'end_time': "17:00",
    'daytype': "WRKDY",
    'breaks': "00:15:00",
}
```

Consider that the UserID will be in the session database, then we simply run some server-side validations and then enter the entry into the db, there are also some client-side validation, which is essentially the same as here. The redundancy for validation is just *good practice* because of the various malicious ways it is possible to subvert client-side javascript or turn it off completely. Therefore, redundancy.

When this view is launched, we create a server-side counterpart of the json which is in request object. We then fill it, passing None if there are any items missing.

We then create a json_data dict to store the json success/error codes in to pass back to the User and inform them of the status of the ajax request.

We then validate the data. Which involves only time validation.

The creation of the entry goes like this: The form object holds purely the data that the TrackingEntry needs to hold, it's also already validated, so, as insecure it looks, it's actually perfectly fine as there has been client-side side validation and server-side validation. There will also be validation on the database level. So we can use **kwargs to instantiate the TrackingEntry and .save() it without much worry for saving some erroneous and/or harmful data.

If all goes well with saving the TrackingEntry, i.e. the entry isn't a duplicate, or the database validation doesn't fail. We then generate the calendar again using the entry_date in the form. We use this date because it's logical to assume that if the user enters a TrackingEntry using this date, then their calendar will be showing this month.

We create the calendar and push it all back to the client. The client-side code then updates the calendar display with the new data.

Parameters **request** – HttpRequest object.

Returns `HttpResponse` object with the mime/application type as json.

Return type `HttpResponse`

`timetracker.utils.calendar_utils.`**`ajax_change_entry`**`(`*request*`)`

Changes a calendar entry asynchronously

This method works in an extremely similar fashion to `ajax_add_entry()`, with modicum of difference. The main difference is that in the add_entry method, we are simply looking for the hidden-id and deleting it from the table. In this method we are *creating* an entry from the form object and saving it into the table.

Parameters **request** – `HttpRequest`

Returns `HttpResponse` with mime/application of JSON

Return type `HttpResponse`

`timetracker.utils.calendar_utils.`**`ajax_delete_entry`**`(`*request*`)`

Asynchronously deletes an entry

This method is for RUSERs who wish to delete a single entry from their TrackingEntries. This method is only available via ajax and obviously requires that users be logged in.

We then create our json_data map to hold our success status and any error codes we may generate so that we may inform the user of the status of the request once we complete.

This part of the code will catch all errors because, well, this is production code and there's no chance I'll be letting server 500 errors bubble to the client without catching and making them sound pretty and plausable. Therefore we catch all errors.

We then take the entry date, and generate the calendar for that year/ month.

Parameters **request** – `HttpRequest`

Returns `HttpResponse` object with mime/application of json

Return type `HttpResponse`

`timetracker.utils.calendar_utils.`**`ajax_error`**(*request*)

Returns a HttpResponse with JSON as a payload

This function is a simple way of instantiating an error when using json_functions. It is decorated with the json_response decorator so that the dict that we return is dumped into a json object.

Parameters **error** – `str` which contains the pretty error, this will be seen by the user so make sure it's understandable.

Returns `HttpResponse` with mime/application of json.

Return type `HttpResponse`

`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.

This function simply deletes a user. We asynchronously delete the user because it provides a better user-experience for the people doing data entry on the form. It also allows the page to not have to deal with a jerky nor have to create annoying 'loading' bars/spinners.

Note This function should not be called directly.

This function should be POSTed to via an Ajax call. Like so:

```
$.ajaxSetup({
    type: "POST",
    url: "/ajax/",        // "ajax" is the url we created in urls.py
    dataType: "json"
});

$.ajax({
    data: {
        user_id: 1
    }
});
```

Once this is received, we check that the user POSTing this data is an administrator, or at least a team leader and we go ahead and delete the user from the table.

Parameters **request** – `HttpRequest`

Returns `HttpResponse` mime/application JSON

Return type `HttpResponse`

`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.

How this works is that, we iterate through each of the entries found in the TrackingEntry QuerySet for {year}/{month}. Create the table>td for that entry then attach the CSS class to that td. This means that each different type of day can be individually styled per the front-end style that is required. The choice to use a custom calendar table is precisely *because of* this fact the jQueryUI calendar doesn't support the individual styling of days, nor does it support event handling with the level of detail which we require.

Each day td has one of two functions assigned to it depending on whether the day was an 'empty' day or a non-empty day. The two functions are called:

```
function toggleChangeEntries(st_hour, st_min, full_st,
                             fi_hour, fi_min, full_fi,
                             entry_date, daytype,
                             change_id, breakLength,
                             breakLength_full)
 // and

 function hideEntries(date)
```

These two functions could be slightly more generically named, as the calendar markup is used in two different places, in the {templates}/calendar.html and the {templates}/admin_view.html therefore I will move to naming these based on their event names, i.e. 'calendarClickEventDay()' and 'calendarClickEventEmpty'.

The toggleChangeEntries() function takes 11 arguments, yes. 11. It's quite a lot but it's all the relevant data associated with a tracking entry.

1. st_hour is the start hour of the tracking entry, just the hour.
2. st_min is the start minute of the tracking entry, just the minute.
3. full_st is the full start time of the tracking entry.
4. fi_hour is the end hour of the tracking entry, just the hour.
5. fi_min is the end minute of the tracking entry, just the minute.
6. full_fi is the full end time of the tracking entry.
7. entry_date is the entry date of the tracking entry.
8. daytype is the daytype of the tracking entry.
9. change_id this is the ID of the tracking entry.
10. breakLength this is the break length's minutes. Such as '15'.
11. This is the breaklength string such as "00:15:00"

The hideEntries function takes a single parameter, date which is the date of the entry you want to fill in the Add Entry form.

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

Parameters

- **year** – Integer for the year required for output, defaults to the current year.
- **month** – Integer for the month required for output, defaults to the current month.
- **day** – Integer for the day required for output, defaults to the current day.
- **user** – Integer ID for the user in the database, this will automatically, be passed to this function. However, if you need to use it in another setting make sure this is passed.

Returns HTML String

`timetracker.utils.calendar_utils.`**`gen_datetime_cal`**(*year*, *month*)

Generates a datetime list of all days in a month

Parameters

- **year** – `int`
- **month** – `int`

Returns A flat list of datetime objects for the given month

Return type `List` containing `datetime.datetime` objects.

`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`

Raises KeyError

`timetracker.utils.calendar_utils.`**`get_user_data`**(*request*)

Returns a user as a json object.

This is a very simple method. First, the `HttpRequest` POST is checked to see if it contains a user_id. If so, we grab that user from the database and take all their relevant information and encode it into JSON then send it back to the browser.

Parameters **request** – `HttpRequest` object

Returns `HttpRequest` with mime/application of JSON

Return type `HttpResponse`

`timetracker.utils.calendar_utils.`**`mass_holidays`**`(`*request*`)`

Adds a holidays for a specific user en masse

This function takes a large amount of holidays as json input, iterates over them, adding or deleting each one from the database.

The json data looks as such:

```
holidays = {
    1: daytype,
    2: daytype,
    3: daytype
    ...
 }
```

And so on, for the entire month. In the request object we also have the month and the year. We use this to create a date to filter the month by, this is so that we're not deleting/changing the wrong month. The year/month are taken from the current table headings on the client. We then check what kind of day it is.

If the daytype is 'empty' then we attempt to retrieve the day mapped to that date, if there's an entry, we delete it. This is because when the holiday page is rendered it shows whether or not that day is assigned. If it was assigned and now it's empty, it means the user has marked it as empty.

If the daytype is *not* empty, then we create a new TrackingEntry instance using the data that was the current step of iteration through the holiday_data dict. This will be a number and a daytype. We have the user we're uploading this for and the year/month from the request object. We also choose sensible defaults for what we're not supplied with, i.e. we're not supplied with start/end times, nor a break time. This is because the holiday page only deals with *non-working-days* therefore we can track these days with zeroed times.

If at this point an IntegrityError is raised, it means one of two things: we can either have a duplicate entry, in which case we retrieve that entry and change it's daytype, or we can have a different error, in which case we wrap up working with this set of data and return an error to the browser.

If all goes well, we mark the return object's success attribute with True and return.

Parameters **request** – `HttpRequest`

Returns `HttpResponse` with mime/application as JSON

Note All exceptions are caught, however here is a list:

Raises `IntegrityError` `DoesNotExist` `ValidationError` `Exception`

`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.

Access Level: All

First we pull out the user instance that is currently logged in. Then as with most ajax functions, we construct a map to receive what should be in the in the POST object. This view specifically deals with changing a Name, Surname and Password. Any other data is not required to be changed.

Once this data has been populated from the POST object we then retrieve the string names for the attributes and use setattr to change them to what we've been supplied here.

Parameters **request** – `HttpRequest`

Returns `HttpResponse` with mime/application as JSON

`timetracker.utils.calendar_utils.`**`useredit`**(*request*)

This function both adds and edits a user

- Adding a user

Adding a user via ajax. This function cannot be used outside of an ajax request. This is simply because there's no need. If there ever is a need to synchronously add users then I will remove the @request_check from the function.

The function shouldn't be called directly, instead, you should POST to the ajax view which points to this via `timetracker.urls` you also need to include in the POST data. Here is an example call using jQuery:

```
$.ajaxSetup({
    type: "POST",
    dataType: "json"
});

$.ajax({
    url: "/ajax/",
    data: {
        'user_id': "aaron.france@hp.com",
        'firstname': "Aaron",
        'lastname': "France",
        'user_type': "RUSER",
        'market': "BK",
        'process': "AR",
        'start_date': "2012-01-01"
        'breaklength': "00:15:00"
        'shiftlength': "00:07:45"
        'job_code': "ABC123"
        'holiday_balance': 20,
        'mode': "false"
    }
});
```

You would also create success and error handlers but for the sake of documentation lets assume you know what you're doing with javascript. When the function receives this data, it first checks the 'mode' attribute of the json data. If it contains 'false' then we are looking at an 'add_user' kind of request. Because of this, and the client-side validation that is done. We simply use some **kwargs magic on the `timetracker.tracker.models.Tbluser` constructor and save our Tbluser object.

Providing that this didn't throw an error and it may, the next step is to create a Tblauthorization link to make sure that the user that created this user instance has the newly created user assigned to their team (or to their manager's team in the case of team leaders). We make the team leader check, if it's a team leader we call get_administrator() on the authorized user and then save the newly created user into the Tblauthorization instance found. Once this has happened we send the user an e-mail informing them of their account details and the password that we generated for them.

- Editing a user

This function also deals with the *editing* of a user instance, it's possible that this functionality will be refactored into it's own function but for now, we have both in here.

Editing a user happens much the same as adding a user save for some very minor differences:

```
$.ajaxSetup({
   type: "POST",
   dataType: "json"
});

$.ajax({
    url: "/ajax/",
    data: {
        'user_id': "aaron.france@hp.com",
        'firstname': "Aaron",
        'lastname': "France",
        'user_type': "RUSER",
        'job_code': "ABC456"
        'holiday_balance': 50,
        'mode': 1
    }
});
```

You may notice that the amount of data isn't the same. When editing a user it is not vital that all attributes of the user instance are changed and/or sent to this view. This is because of the method used to assign back to the user instance the changes of attributes (getattr/setattr).

The attribute which determines that the call is an edit call and not a add user call is the mode, if the mode is not false and is a number.

When we first step into this function we look for the mode attribute of the json data. If it's a number then we look up the user with that user_id we then step through each attribute on the request map and assign it to the user object which we retrieved from the database.

Parameters **request** – `HttpRequest`

Returns `HttpResponse` with mime/application of JSON

Raises `Integrity` `Validation` and `Exception`

Note Please remember that all exceptions are caught here and to make sure that things are working be sure to read the response in the browser to see if there are any errors.

`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`

1.3.3 timetracker.utils.datemaps

Maps of useful data

Django has several of these built-in but they are annoying to use.

`WEEK_MAP_MID`: This is a map of the days of the week along with the mid-length string for that value. For example:

```
WEEK_MAP_MID = {
    0: 'Mon',
    1: 'Tue',
    ...
}
```

`WEEK_MAP_SHORT`: This is similar except using a longer string.

`MONTH_MAP`: This is a map of the months which refer to a two-element tuple which has the short code for the month and the long string for that month. I.e. 'JAN' and 'January'.

`WORKING_CHOICES`: This is a tuple of two-element tuples which contain the only possible working day possibilites.

`ABSENT_CHOICES`: This is a tuple of two-element tuples which contain the only possible absent day possibilities.

`DAYTYPE_CHOICES`: This is both `WORKING_CHOICES` and `ABSENT_CHOICES` joined together to give all the daytype possibilities.

`timetracker.utils.datemaps.`**`float_to_time`**`(`*timefloat*`)`

Takes a float and returns the same representation of time.

Parameters timefloat – This is a `float` which needs to be represented as a timestring.

Return type `str` such as '00:12' or '09:15'

`timetracker.utils.datemaps.`**`generate_select`**`(`*data*, *id=''*`)`

Generates a select box from a tuple of tuples

```
generate_select((
    ('val1', 'Value One'),
    ('val2', 'Value Two'),
    ('val3', 'Value Three')
))
```

will return:-

```
<select id=''>
   <option value="val1">Value One</option>
   <option value="val2">Value Two</option>
   <option value="val3">Value Three</option>
</select>
```

Parameters data – This is a tuple of tuples (can also be a list of lists. But tuples will behave more efficiently than lists and who likes mutation anyway?

Return type `str`/HTML

`timetracker.utils.datemaps.`**`pad`**`(`*string*, *padchr='0'*, *amount=2*`)`

Pads a string

Parameters

- **string** – This is the string you want to pad.
- **padchr** – This is the character you want to pad the string with.
- **amount** – This is the length of the string you want the input end up.

Return type `str`

1.3.4 timetracker.utils.error_codes

Database errors give out a tuple of information. The first of which is a number, we can grab that and check what kind of error we're getting

Currently this is somewhat small as there aren't many errors that get thrown consistantly to warrant putting their codes into a module.

`DUPLICATE_ENTRY`: This is thrown when the database validation reports that there is already an entry in the database with conflicting values for whatever is set as a Unique and/or UniqueTogether.

`CONNECTION_REFUSED`: This error is thrown when the SMTP server is down and/or is not responding.

1.3.5 timetracker.utils.decorators

Module to for sharing decorators between all modules

`timetracker.utils.decorators.`**`admin_check`**`(`*func*`)`

Wrapper to see if the view is being called as an admin

This works by 1) Checking if there is a user_id in the session table. 2) If that user is a real user in the database and 3) if that user's is_admin() returns True.

Parameters **func** – A function with a request object as a parameter

Returns Nothing directly, it returns the function it decorates.

Raises `Http404` error

`timetracker.utils.decorators.`**`json_response`**`(`*func*`)`

Decorator function that when applied to a function which returns some json data will be turned into a HttpResponse

This is useful because the call site can literally just call the function as it is without needed to make a http-response.

Parameters **func** – Function which returns a dictionary

Returns `HttpResponse` with mime/application as JSON

`timetracker.utils.decorators.`**`loggedin`**`(`*func*`)`

Decorator to make sure that the view is being accessed by a logged in user.

This works by simply checking that the user_id in the session table is 1) There and 2) A real user. If either of these aren't satisfied we throw back a 404.

We also log this.

Parameters **func** – A function which has a request object as a parameter

Returns Nothing directly, it returns the function it decorates

Raises `Http404` error

`timetracker.utils.decorators.`**`request_check`**`(`*func*`)`

Decorator to check an incoming request against a few rules

This function should decorate any function which is supposed to be accessed by and only by Ajax. If we see that the function was accessed by any other means, we raise a `Http404` and give up processing the page.

We also make a redundant check to see if the user is logged in.

Parameters **func** – Function which has a request object parameter.

Returns The function which it decorates.

Raises `Http404`

1.3.6 timetracker.tracker.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.

1.3.7 timetracker.tracker.admin

This module directly deals with how models are interacted with in the admin interface. This has no direct impact for users but it is useful for webmasters administrating the application.

class `timetracker.tracker.admin.`**AuthAdmin**(*model*, *admin_site*)

Creates access to and customizes the admin interface to the Tblauthorization instances. We add the __unicode__ and the display_users functions so that the display allows us to view the team associated with the administrator and the administrator's printed representation.

class `timetracker.tracker.admin.`**TrackerAdmin**(*model*, *admin_site*)

Creates access to and customizes the admin interface to the TrackingEntry instances. We have no special functions or list_display additions because the default values are useful enough as the interface to edit these items is far more useful and better programmed than the basic model editor the admin interface provides.

class `timetracker.tracker.admin.`**UserAdmin**(*model*, *admin_site*)

Creates access to and customizes the admin interface to the tbluser instances. We give the list_display of __unicode__ and a 2nd type of display_user_tyep because this shows the printed representation of these values and makes it easier to navigate a large selection of users.

actions is a list of functions which are in the 'Actions' list, there is a default value of 'delete selected <table>' which Django inserts automatically. We add here the send_password_reminder function defined above.

`timetracker.tracker.admin.`**`send_password_reminder`**(*modeladmin*, *request*, *queryset*)

Send an e-mail reminder to all the selected employees.

This appears as an option in the 'Action' list in the admin interface for when editing the tbluser instances. This allows you to send an e-mail reminder en masse to all users selected.

1.3.8 timetracker.tracker.forms

Forms used for user input for the tracker app

Several forms are extremely simple and thus it's not required to 'hard-code' them into HTML templates, which would couple our output with the view functions which simply isn't good MVC practice.

Module Overview

`EntryForm`	`AddForm`
`Login`	

class `timetracker.tracker.forms.`**`AddForm`**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*)

Add entry form

This Class creates a form which allows users to add an entry into the timetracking portion of the app. See `ChangeEntry` for a detailed description of these two classes as they have a high coupling factor.

class `timetracker.tracker.forms.`**`EntryForm`**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*)

Change entry form

This Class creates a form which allows users to change an existing entry which they have added to the timetracking portion of the site. The fields match up precisely to the `AddForm` because the data will be the same. We have duplicated the code here because we need to explicitly set the id values using the widget.attrs.update on the widget dict. This is so that each widget could be styled individually and so that we have something easily accessbile by front-end javascript code.

class `timetracker.tracker.forms.`**`Login`**(*data=None*, *files=None*, *auto_id='id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.util.ErrorList'>*, *label_suffix=':'*, *empty_permitted=False*)

Basic login form

This form renders into a very simple login field, with fields identified differently, both for styling and the optional javascript by-name handling. Whilst this form is extremely simple, coding one each and every time we wish to use one is just dumb and it's much easier to code it once and import it into our project.

1.4 Client-side Javascript modules

1.4.1 Calendar Module

`ajaxCall`(*form*)

Creates an ajax call depending on what called the function. Server-side there is a view at domain/ajax/ which is designed to intercept all ajax calls.

The idea is that you define a function, add it to the ajax view's dict of functions along with a tag denoting it's name, and then pass the string to the 'form_type' json you sent to that view.

In this particular ajax request function we're pulling out form data depending on what form calls the ajaxCall.

paramemter form This argument is the string identifier of the form from which you wish to send the data from. The possible choices are the Add Form and the Change Form.

returns This function returns false so that the form doesn't try to carry on with it's original function.

`onOptionChange`(*element*)

Specific selections determine which elements of the form are disabled. For example there is no need to allow people to change their working time for a vacation day. Similarly, if they have previously selected a vacation day, then we need to re-enable the form else they will no longer be able to enter the time into the fields.

paramemter element This argument is the string identifier of the form from which you wish to send the data from. The possible choices are the Add Form and the Change Form.

returns True. This always returns true to signify to any programmatic callers that we have finished the function. There are no error codes or errors thrown.

CHAPTER

TWO

INDICES AND TABLES

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

PYTHON MODULE INDEX

INDEX