FluxFingers Scoreboard Documentation

Release 0.4.10

Florian Rüchel

March 29, 2015

Contents

Welcome to the almighty FluxFingers scoreboard! Behold fools for you will experience the power of the FluxFingers that will bring you joy and love when running your own CTF or competition.

CHAPTER 1

Come Again?

Running a CTF is hard. Designing challenges is hard. Providing a stable infrastructure is hard. But a scoreboard should not be hard, it should be the easiest part of hosting a CTF.

So here it goes: A hopefully simple solution so you can run your own CTF (or different competition). To get you started head over to *Getting Started*! Otherwise, take a look at the table of contents below.

Oh and you can find our Source Code hosted on GitHub.

CHAPTER 2

Documentation for Users

2.1 Getting Started

2.1.1 Installing the Scoreboard

As a Python WSGI application, it can be run in any environment that supports the WSGI protocol.

> **Warning:** Irrespectable of which way you choose, you **must** make sure that the complete `/admin` path is protected by external measures (e.g. HTTP Basic Auth). The backend has no separate protection.

> **Warning:** If not using Nginx, you have to make sure to **manually** set the `Host` header to a static value and not trust the client as Pyramid relies on it to generate URLs!

Quickstart

First of all: The scoreboard uses PostgreSQL (and has only been tested with it). It does explicitly *not* support MySQL as MySQL is crappy software. However, it does not rely on obscure database features so you may get it running with a different server (though not MySQL. Never MySQL.). So go ahead and install PostgreSQL. You can then prepare the database:

```
-- Choose a secure password here!
CREATE USER fluxscoreboard WITH PASSWORD 'mypass';
CREATE DATABASE scoreboard WITH OWNER fluxscoreboard;
```

Now let's install! Adjust the paths to your liking. Also pay attention to the username and group for the web server, it depends on the server you are using and the distribution, it may for example be "www-data" or "http" or "www". You should also use virtualenv instead of your global python installation to run the scoreboard under.

```
mkdir -p /var/www/ctf
cd /var/www/ctf
chown http:http .
virtualenv .
. bin/activate
git clone git@github.com:Javex/fluxscoreboard.git
cd fluxscoreboard/
./run install production
```

Now create a valid configuration by opening `cfg/production.ini` and filling in all relevant values (everything is documented there). The `run` tool has already performed some of the heavy lifting.

Webserver Nginx + gunicorn

This is an example configuration file that can be used with Nginx. For this server you additionally need a reverse proxy that handles the WSGI protocol.

```
upstream fluxscoreboard {
    server 127.0.0.1:6875;
}

server {
    listen 80;
    server_name mydomain.com;
    rewrite ^ https://$server_name$request_uri? permanent;  # enforce https
}

server {
    listen 443 ssl;
    server_name mydomain.com;

    ssl_certificate     /path/to/certificate.crt;
    ssl_certificate_key /path/to/private_key.key;

    access_log /var/log/nginx/fluxscoreboard_access.log;
    error_log /var/log/nginx/fluxscoreboard_error.log;

    # This is not needed if the page itself should be public
    #auth_basic "Restricted";
    #auth_basic_user_file /var/www/ctf/fluxscoreboard/.htpasswd;

    # Security headers
    add_header X-Frame-Options DENY;
    add_header Content-Security-Policy "default-src 'none'; connect-src 'self'; font-src 'self'; img-
    add_header X-XSS-Protection 0;
    add_header Strict-Transport-Security max-age=31536000;


    location / {
        # This file must be available under a sensible name and is reference
        # relative to /etc/nginx (or wherever nginx.conf lies)
        include nginx_proxy.conf;
    }

    location /admin {
        # This file must be available under a sensible name and is reference
        # relative to /etc/nginx (or wherever nginx.conf lies)
        include nginx_proxy.conf;

        # This MUST be active to protect the admin backend. It may NOT be
        # deactivated as it exposes a lot of features including adding a lot of
        # data and sending emails.
        auth_basic "Restricted";
        auth_basic_user_file /var/www/ctf/fluxscoreboard/.htpasswd_admin;
    }

    location /static {
        # A path to the root, i.e. it will have /static appended (+ the
        # searched file). This circumvents the application server as these are
        # static anyway.
        root                    /var/www/ctf/fluxscoreboard/fluxscoreboard;
```

```
        expires                 30d;
        add_header              Cache-Control public;
        access_log              off;
    }
}
```

This defines the base application. It is configured for SSL only access and automatically redirects any HTTP requests. There is not much to change here except that you might want a different path for your application. However, there is a second file that contains the actual options:

```
    # Don't trust the client on this one!
    proxy_set_header        Host $server_name;
    proxy_set_header        X-Real-IP $remote_addr;
    # This is set to assure the Client-Addr is trustworthy
    proxy_set_header        X-Forwarded-For $remote_addr;
    proxy_set_header        X-Forwarded-Proto $scheme;

    client_max_body_size    10m;
    client_body_buffer_size 128k;
    proxy_connect_timeout   60s;
    proxy_send_timeout      90s;
    proxy_read_timeout      90s;
    proxy_buffering         off;
    proxy_temp_file_write_size 64k;
    proxy_redirect          off;
    proxy_pass              http://fluxscoreboard;
```

This file has to be saved somewhere *as-is* and the path in the main configuration has to be adjusted in such a way that it points to it relative to the Nginx main configuration directory (see comments in file). Restart Nginx.

Note: The sample configuration sets the 'Host' header **statically** to whatever `server_name` setting you chose. Do not trust the `$host` header of the client: It may be spoofed but Pyramid relies on it so we have to make sure it is a trusted value!

After Nginx is configured this way you don't have to do much for gunicorn: It already has a valid configuration in the default configuration file (see below).

Todo

Configure gunicorn with proper logging.

Protecting the Admin Backend

Finally, you should protect your backend with HTTP Auth:

```
htpasswd -c -b /var/www/ctf/fluxscoreboard/.htpasswd_admin fluxadmin secretpassword
```

This will protect your backend from unauthorized access.

Cron!

Since calculating points is a heavy task, you should not do it on every request from a user. Instead, we provide a convenience function that regularly updates the points of teams and challenges:

```
./run update_points
```

Make sure to put that in a cron. Run it e.g. every five minutes.

Test it!

This should be it. You should start the server as a test:

```
gunicorn_paster production.ini
```

Try out the server by visiting the domain configured for Nginx. Fix any errors that appear, then enable the `daemon = True` option for gunicorn in `production.ini`. You are now good to go. A simple setup is to just run it and close the shell. However, with this way you have no service, no monitoring and no restarting.

Todo

Add some more details here.

2.2 Use-Case & Features

2.2.1 Is This the Scoreboard I Am Looking For?

Absolutely (*waves hand*). Wait. It may be. Are you trying the run a jeopardy style CTF? Then this *might* be right for you. Are you looking for an enterprise-level firewall? Then this is not for you.

Seriously: If you want to host some kind of competition that has teams and challenges and points and awesomeness then you might want to read on.

2.2.2 What Does It Do?

Here are some of the neat features it has to offer:

- Allows to set start and end time (which allows you three states: before, during and after).
  - Before: Teams can register and watch the list of registered teams. They can log in, edit their profile and stuff. However, the challenges and the design are hidden from them.
  - During: Teams can no longer register (See #4). They can log in, watch the scoreboard (i.e. the other teams points and so on), view and solve challenges and generally play the CTF.
  - After: Registration is off. Login is off. Submitting challenge solutions is also off. The scoreboard is static and does not change anymore. The challenges can still be viewed.
- Archive Mode: Adds a fourth state: When the scoreboard is running in archive mode it is no longer possible to register or login. It behaves similar to the "After" mode but it allows solving challenges again. It then just notifes you if the solution was correct but does not persist this anywhere.
- Avatars: Teams can upload cool images. Beware of animated gifs :-)
- Mass Mail: Want to notify all participants about something? Send a mass mail to all of them. Warning: Teams cannot turn this off so be sure to use it only when really needed.
- Dynamic Challenges: You can write a challenge that does not work along the usual rules: Instead of having a static solution, you can control most parts of a challenge yourself. Theoretically, you could add multiple dynamic

challenges at the same time but this has not really been tested. If you wanna create a dynamic challenge, take a look at *Writing a Dynamic Challenge*.

- Track IPs of teams: Some suckers want to ruin your nice competition by breaking your rules and gaining an unfair advantage. The scoreboard tracks the IP of every participant and stores it for the team. If you notice someone breaking your rules on your challenges, you can look up the IP and get the team name. You can then easily deactivate/ban/warn/harass them. You can also get all known IPs for a team. After the CTF you could also use this for some kind of statistics (e.g. from how many countries your players joined in).
- Log in as any team: An administrator can log in as any team and view the page exactly as they do. This may help during debugging or assisting a team. Of course you should not abuse your power. This will not give you world domination anyway, wrong software for that.
- Challenge Tokens: Identify a team by token to protect your challenge.
- Announcements/Hints: Support as well for global announcements as for hints for particular challenges.
- The rest is pretty basic: Registration, Administration, etc...

You like it? Go a ahead and get started.

2.2.3 Feature Description

Challenge Tokens

Challenge tokens are unique per team and can be used to identify a team against a challenge. An example case would be if rate-limiting by team is desired. You enable them in the admin backend on each challenge that needs a token by setting the "Has Token" option.

Afterwards, the token will be displayed on the challenge description in the frontend. The team will then send it to your challenge. From there you can visit `/verify_token/TOKEN` where you replace `TOKEN` with the provided token. You will receive a simple response: `0` if the token was invalid, `1` if it was valid. Currently, you need to manually look up tokens in the database to find out to which team they belong to.

CHAPTER 3

Documentation for Developers

Warning: The developer documentation has not been updated for some time. It likely contains outdated or invalid information. You can use it as a reference but if in doubt between code and documentation, follow the code. This would need someone to look at it but who would want to do that?

Here is a high level overview of the project:

- `/alembic/` contains the database migration files. Like, when your database wants to migrate to different countries or something along the lines of that.
- `/cfg/` contains all the configuration. Good configurations will be automatically generated by the `run` script.
- `/data/` contains all other data sources needed for the installation.
- `/docs/` contains documentation of the project (javex tries to keep up with documenting stuff, qll writes godlike code which documents itself exclusively to himself).
- `/fluxscoreboard/` contains the actual Pyramids project. Please don't look inside it. I really don't want to explain all folders in there, too :-/
- `/log/` contains all log files. That one is cleverly named, huh?.
- `/tests/` contains all unit tests.
- `/tmp/` I DON'T EVEN KNOW WHAT IT CONTAINS LOL.

3.1 Installation

After accepting that you are unworthy of setting up the project by yourself, you'll find the *run* script helpful. The only thing you have to do is:

```
./run install development
```

Isn't that a nifty bad boy? It asks you everything it needs. So. Cool. I wonder what kind of genius wrote this. Make sure you execute this in a virtual environment. If you really can't help it pass the *–no-venv* flag to skip the virtual env check.

You can then execute the following command to execute all unit tests:

```
./run test
```

Finally you might want to take a look at this sexy web application. The following command will run a development server on port 6543.

```
./run serve
```

If you want to have greater control over the port and shit, run:

```
pserve cfg/development.ini port=8000
```

3.1.1 Interacting with the Database

The database is built with SQLAlchemy which is *the* Python ORM. It has the ability to very easily query the data as classes but beyond that also has the ability to write queries that result in literal SQL strings only we never have to worry about any SQL Injection. Also it takes a lot of type checks, coercing and whatnot pain out of databases.

Please not that this only introduces small bits of how some stuff is done in this application. The introduction to SQLAlchemy itself can be found on their page (the most important part being the ORM tutorial.

Basic Querying

So let's get started. To perform a query, we need the session on which all queries are executed. Note that the `DBSession` is scoped to the current thread. In our context that means that calling it will always give the *exact same* session back as long as we are in the same thread. So it is not required to pass it around all the time. Here's how you get a session to start querying:

```
from fluxscoreboard.models import DBSession
dbsession = DBSession()
```

Now we can execute a query, for example, get all challenges:

```
from fluxscoreboard.models.challenges import Challenge
all_challenges = dbsession.query(Challenge)
```

Note that we did not actually execute a query yet but instead have a query object. It produces results once used as an iterable, e.g. `list(all_challenges)` or `all_challenges.all()`. Then you have a list of challenges. However, here is one thing to note: You should always return query, i.e. not call `all` or similar to produce actual results. That is done those items are actually **used**. This is advantageous in two ways: For one, it supports the lazy technique by which a query is only emitted once the results are actually need. And for the other part, you can refine such queries when they are not yet results. So you could modify the query to only get challenges which are manual:

```
manual_challenges = all_challenges.filter(Challenge.manual == True)
```

Now this is a query that can be executed and it will limit the results. And you should remember to always pass around queries, never the actual lists, until they are used.

Note: This technique works so well because any `sqlalchemy.orm.query.Query` class produces an iterable.

Pagination

The query technique described above is also very useful for pagination, because the `webhelpers.paginate.Page` class supports recieving an iterable (so a query) and can also recieve an `item_count`. This is fortunate because we can find the number of items from a query by exeuting `sqlalchemy.orm.query.Query.count()` on it:

```
page = Page(manual_challenges, page=1,
            item_count=manual_challenges.count(), url=page_url)
```

Don't worry too much about the details here, just note that we could reuse our query to get the item count (which is far more efficient than fetching the complete list to determine it).

The one Exception

There is one exception for only passing queries: The `sqlalchemy.orm.query.Query.one()` method. This function can be used to ensure there was exactly **one** result returned and throw an Exception if no result or multiple results were found. This is a very useful function to query a page with a single item because then you can be sure you have exactly one item and not more.

Upgrading the Database

During development it will likely happen that you need to upgrade the database schema at some point which can lead to problems for multiple reasons. For one, we are using an ORM and since we are not reflecting, we make changes in Python code, but these are not transferred into the database automatically. Secondly, when distributing changes to other instances, the database needs to be adjusted as well. Thus, we use Alembic.

With Alembic, upgrades to the database are managed automatically. In this section, you will find a small overview of how to run default commands that cover the most basic way of doing it in our application. For anything beyond that, check out the original Alembic documentation which covers all those topics.

Suppose you made a change to your database. Before you can run alembic in any way you **must** do one thing first: Have a working configuration. This could be your `development.ini` if working locally or it could be the running `production.ini`. The default configuration files already contain a good working configuration. The most important part is that you already have a valid database configuration so you can actually connect to the database and make changes. Seems legit, right?

Note: For the rest of this section, the configuration used will be `development.ini`. You can switch this out with whatever configuration file name you are using (e.g. `production.ini`).

After you have done this, let's get to work. The easiest and fastest way is to let Alembic try to detect your changes. This will work for most cases, but it will, for example, fail on renaming of columns or tables and instead detect a pair of add and delete. Anyway, this is how you let alembic create a basic file for your changes:

```
alembic -c development.ini revision --autogenerate -m "YOUR_CHANGE_DESCRIPTION"
```

Alembic will generate a file in the `alembic/versions` directory and give you its name. Open it and check the `upgrade` and `downgrade` methods: All you need to do is make sure that these do what you need. Afterwards, you should testrun your upgrade:

```
alembic -c development.ini upgrade head
```

This should be straightforward and you should now have an up to date database. Check that it works, i.e. run unit tests, do manual testing, etc. and then make sure downgrade works. For this, you need to find out the previous revision which is specified in the file you opened above as the variable `down_revision`. Lets say it is `123abc` (if there's no previous revision, specify `base`):

```
alembic -c development.ini downgrade 123abc
```

Note: You don't need to specify the full string, just specify as much as is needed to have an unambigous identification, e.g. just `123`.

And that's it. No more work has to be done here. You can now run the upgrade again to get your database up to date and work more on the application.

3.1.2 Displaying and Using Forms

Forms a built using WTForms. This allows us to map forms to classes, build validators and let it take the pain out of forms. There is no real, perfect forms library but with WTForms it is at least easier than writing the HTML directly.

Defining a Basic Form

To define a form derive from the `wtforms.form.Form` class and start defining your fields (for a more thorough definition and explanation refer to the original documentation):

```
from wtforms.form import Form
from wtforms.fields.simple import TextField, SubmitField

class SampleForm(Form):
    text = TextField("Sample Text")

    submit = SubmitField("Submit")
```

Using a Form

Now we can use this form (and load it with data) from a view:

```
def some_view(request):
    form = SampleForm(request.POST)
```

Then we make sure it is valid:

```
if not form.validate():
    # do something
    pass
```

But what does validate do?

Defining Validation on a Form

To get validation, we define a list of validators on a field. For example:

```
from fluxscoreboard.forms.validators import required_validator
class SampleForm(Form):
    text = TextField("Sample Text",
                     validators=[required_validator]
                    )
    ...
```

For details on validators already provided, see the documentation for `fluxscoreboard.forms.validators`. There are already some common validators for length and other stuff defined. Also pay attention that you should have a validator that checks for the allowed database length (e.g. if you database column is `Unicode(255)` check that it is not longer than 255 characters).

From- & Database-Interaction

Getting a form into the database is easy.

```
# Create the form
form = SampleForm(request.POST)
# Create the database object
dbitem = Sample()
# Fill it with the form data
form.populate_obj(dbitem)
```

Of course, this only works when the form fields and database names are the same. You can also manually map the fields together if you want.

Filling a form from the database is also easy:

```
# Load the object (maybe from database)
dbitem = get_item()
# Create the form
form = SampleForm(request.POST, dbitem)
```

What does this do? It loads the database item from somewhere and then instantiates the form as previously but now it fills the missing fields from the database.

You can combine the two approaches above to retrieve a form to be edited and then save it back to the database:

```
def my_view(request):
    dbitem = get_item()
    form = SampleForm(request.POST, dbitem)
    if request.method == 'POST':
        if not form.validate():
            # handle it
            pass
        form.populate_obj(dbitem)
        # redirect or something
    return {'form': form}
```

This approach will load an item from the database, then fill the form correctly and on a `GET` request just display it. Once editing is done and the form gets submitted we now load the form data into the database item.

Note: This is a fairly simple example. Normally you would want to track something liket the id in a field and use that to query the database and also have some nice wrapping stuff that displays messages, redirects on success and notifies the user of any problems.

Todo

There is currently an ugly display of buttons on the Action listings as they don't have a 100% width. However setting it to 100% causes Overflow of larger text. Both is not nice :(

3.1.3 Templating in the Application

For templating we use Mako Templates. I think it is not necessary to explain the advantages of a template engine over manual print statements and string replacement. This documentation should only show some nice practices that can make your life easier.

Creating a New Template

Simple create a file in the `templates/` directory with a suffix of `.mako` and put the following content in it to get started:

```
<%inherit file="base.mako"/>
```

This one line integrates your template with the default style. The next step is to add the HTML you want. For the syntax of Mako look at the *Mako documentation*.

Page Style

The page is/was developed with Bootstrap 3 which provides some nice styles to work with when a design is not ready yet. You are encouraged to work with the bootstrap classes as we will work to have a stylesheet that defines the same classes and can act as a drop-in replacement thus allowing us to quickly change the style of the page when we want.

Using Defs with Forms

When displaying forms you will find yourself repeating over and over (either when displaying similar forms or even inside a form for each field). You can use defs to make your life a whole lot easier. For example, take this def that renders a form.

```
<%def name="render_form(action, form, legend, display_cancel=True)">
    <form method="POST" action="${action}" class="form-horizontal">
        <legend>${legend}</legend>
        % for field in [item for item in form if item.name not in ["id", "submit", "cancel"]]:
        <div class="form-group">
            ${field.label(class_="col-4 control-label")}
            <div class="col-8">
    ## This is a really ugly solution to a limitation of WTForms. It would be a lot nicer to rebuild
                ${field(class_="form-control")}
                % for msg in getattr(field, 'errors', []):
                    <div class="alert alert-danger">${msg}</div>
                % endfor
            </div>
        </div>
        % endfor
        <div class="col-4"></div>
        <div class="col-8">
            % if getattr(form, 'id', None) is not None:
                ${form.id()}
            % endif
            % if display_cancel and hasattr(form, 'cancel'):
                ${form.cancel(class_="btn btn-default")}
            % endif
            ${form.submit(class_="btn btn-primary")}
        </div>
    </form>
</%def>
```

This is a somewhat complex piece of code but it shows nicely how to build a reusable def: This will render a simple form with a legend and all fields horizontally aligned. It will first display all fields in the order defined in the form but skip special fields. Then it will display the special fields in the right way.

The actual implementation has some additional complexity that can help in certain situations to a look at the template files won't hurt. This example is from `templates/_form_functions.mako`.

nl2br

The `fluxscoreboard.util.nl2br()` function can be used to turn newlines into line breaks for improved display in HTML. It's usage is described on the API documentation

3.1.4 Times and Timezones

Working with times is relatively easy if you obey one rule: **UTC, always!**. That means: Always store times in UTC so that when to receive them, they are UTC even if the object is naive.

Now

To get the current time use `datetime.datetime.utcnow()` or `time.time()`. This yields UTC time (in the case of `time` this is irrelevant since the Epoch is timezone independent.

Converting Times

Conversion between a timestampe and a datetime object can be done with these functions (note that there is also the representation of timetuple which can also be helpful sometimes).

- timestamp –> datetime: `datetime.datetime.utcfromtimestamp()`
- datetime –> timestamp: `datetime.datetime.utctimetuple()` and `calendar.timegm()`

Displaying Times

The most easy way to display a time is to use the `fluxscoreboard.util.tz_str()` function:

```
from pytz import utc
from fluxscoreboard.util import tz_str
from datetime import datetime
local_time_str = tz_str(utc.localize(datetime.utcnow()),
                        request.team.timezone)
```

This is very easy because the team property is available in the request and it already know the teams timezone. Where you get the timestamp depends on what you are working on. For example, all timestamps in the database are managed in such a way that they already have the UTC timezone attached. Thus, for example, to display an announcements' time, you would go like this:

```
from fluxscoreboard.util import tz_str
from fluxscoreboard.model.news import News
news = News()
local_time_str = tz_str(news.timestamp, request.team.timezone)
```

More In-Depth Explanation

To display times, we need a timezone to display it in. Teams provide their timezone on registration so that is the timezone we will use for display of local timestamps. To get a localized timestamp from a naive timestamp (e.g. one generated from the methods described in Now) we use the pytz module to first build a UTC timestamp (remember, **always UTC**?) which we can then localize:

```
from pytz import utc, timezone
from datetime import datetime
timestamp = utc.localize(datetime.utcnow())
local_time = timestamp.astimezone(timezone("Europe/Berlin"))
```

This converts a utc timestamp into localized German time.

New Timestamps

When you introduces new timestamps, always remember: **UTC, always**! If you want to create a specific time, attach a timezone upon creation:

```
from pytz import utc
from datetime import datetime
timestamp = datetime(2013, 1, 1, tzinfo=utc)
```

If you are saving it somewhere (i.e. database) save it as a naive UTC timestamp (saves you a lot of trouble) and then, upon retrieval, convert it to a timezone aware UTC timestamp. For example, take a look at `fluxscoreboard.models.news.News` which shows how you could implement this transparently.

3.1.5 Routing and Views

The system's routing and views are closely related and so here is a joined explanation of both. If you want to add a new view, there has to be a way to reach it and that is via routes. So let's start off by adding a new view.

Note: Further explanation of routes and views can be found on the Pyramid documentation.

Adding a New View

Views should be defined in the `fluxscoreboard.views` package. There you will find `fluxscoreboard.views.front.BaseView` from which you should derive all frontend views (as it provides access to some helpful attributes). You can also add your view to an existing view class which is most likely more what you want so let's go down that road and define a new frontend view:

```
class FrontView(BaseView):
    ...

    @logged_in_view(route_name='my_route', renderer='my_route.mako')
    def my_route(self):
        # Do view stuff here
        return {}
```

Here's what we did: We add a method to the class and decorate it with `fluxscoreboard.views.front.logged_in_view`. This decorator protects the view so only logged-in teams can see it. We assigned it the route name `my_route` and also specified a template `my_route.mako`. A template isn't required, but usually only in a case where you redirect after you are with your view. After the view is done we return an empty `dict`. This contains the data passed to the template (in our case there is no data).

This was most of the work of adding a view. However, we still haven't defined how to add a view. For this, you open up `fluxscoreboard/routes.py` and add a new route, like this:

```
routes = [ ...
          ('my_route', '/my_route'),
```

```
    ...
]
```

As you can see, adding a new route is easy: Now, whenever someone comes to the site and heads to `/my_route`, he will see whatever the view `my_route` wants to display.

3.1.6 Writing a Dynamic Challenge

Dynamic challenges allow for different behavior concerning points calculation, how challenges are solved and displayed and so on. This chapter describes what is possible here and how to achieve it.

Configuration

Behavior of the module can be changed by its configuration. These parameters all must be placed in a module-global dictionary called `configuration` and they should only relate to global configuration states, for everything else use the database and functions provided below.

`allow_multiple` is a boolean flag that determines whether more than one instance of this module is allowed. Most of the times you will want to set this to `False`.

Functions

The following functions can be defined to implement the dynamic behavior. They are all mandatory.

`activate`: Called during application startup and receives to parameters: `config` and `settings`, the first being the pyramid configurator instance and the second being the settings dictionary. You should register your routes here.

`render`: Called when the challenge should be rendered. Return everything you want to display for this challenge. Pay close attention to cover all the available states, e.g. archive mode, ctf has ended, etc. Look at the default challenge template for all important cases. Gets two parameters: The `challenge` represents the challenge instance this module belongs to, `request` is the current pyramid request object.

`get_ponints`: Return an integer denoting the number of points awarded for this challenge. Receives a `team` object representing the current team.

`get_points_query`: Receive an optional parameter on which class to work on, by default the `Team` table should be taken. Return a scalar subquery that fetches the current team's points from the database. It should yield the same result as the `get_points` function except that this result will be calculated in the database.

`title`: Return a unicode string that denotes the title of the module to be displayed in the backend, i.e. something that identifies your module. Not shown in the frontend.

`install`: Install the module, performing any action that is required only on application installation.

Models & Views

Models and views are handled the same way as in the rest of the application: Use the `Base` class and `view_config` decorator. To register a view you can also work inside `activate`.

Templates

Put them in a directory called `dynamic` in the `templates` folder. You can also create a subdirectory there if you want.

Example

This is the (old, removed) flag challenge. It is not exactly up to date and does not completely use the interface specified above but an old version, but you get the idea.

```
# encoding: utf-8
from __future__ import unicode_literals, print_function, absolute_import
from fluxscoreboard.config import ROOT_DIR
from fluxscoreboard.models import Base, DBSession
from fluxscoreboard.models.challenge import Challenge
from fluxscoreboard.models.team import get_team_by_ref
from fluxscoreboard.util import now
from fluxscoreboard.views.front import BaseView
from pyramid.httpexceptions import HTTPFound
from pyramid.renderers import render
from pyramid.view import view_config
from requests.exceptions import RequestException
from sqlalchemy.orm import relationship, backref
from sqlalchemy.orm.exc import NoResultFound, MultipleResultsFound
from sqlalchemy.orm.mapper import validates
from sqlalchemy.schema import ForeignKey, Column
from sqlalchemy.sql.expression import func
from sqlalchemy.types import Integer, Unicode, BigInteger
from tempfile import mkdtemp
import csv
import logging
import os.path
import requests
import shutil
import socket
import transaction
import zipfile


log = logging.getLogger(__name__)
allow_multiple = False
"""Whether multiple instances of this are allowed"""
# GeoIP database
db_url = 'http://geolite.maxmind.com/download/geoip/database/GeoIPCountryCSV.zip'


class FlagView(BaseView):
    @view_config(route_name='ref', renderer="json")
    def ref(self):
        """
        Public view that is invoked at the ``ref`` route to which a client
        delivers a ``ref_id`` by which a team is found. This ID is then used
        to find the team it belongs to and upgrade its flag count if the
        location the client was from was not already in the list of registered
        locations for the team.
        """
        if self.archive_mode:
            flash_msg = "This challenge cannot be solved in archive mode."
            self.request.session.flash(flash_msg, 'error')
            return HTTPFound(location=self.request.route_url('home'))
        try:
            challenge = (DBSession.query(Challenge).
                         filter(Challenge.module == 'flags').one())
        except NoResultFound:
```

```
            ret = {'success': False, 'msg': ("There is no challenge for flags "
                                             "right now")}
            return ret
        except MultipleResultsFound:
            ret = {'success': False, 'msg': ("More than one challenge is "
                                             "online. This shouldn't happen, "
                                             "contact FluxFingers.")}
            return ret
        if (not challenge.online or
                self.request.settings.submission_disabled or
                now() > self.request.settings.ctf_end_date):
            ret = {'success': False}
            if not challenge.online:
                ret["msg"] = "Challenge is offline."
            elif self.request.settings.submission_disabled:
                ret["msg"] = "Submission is disabled."
            elif now() > self.request.settings.ctf_end_date:
                ret["msg"] = "CTF is over."
            return ret
        ref_id = self.request.matchdict["ref_id"]
        try:
            team = get_team_by_ref(ref_id)
        except NoResultFound:
            ret = {'success': False,
                   'msg': "Team not found."}
            return ret
        loc = get_location(self.request.client_addr)
        ret = {'success': True}
        if loc is None:
            log.warn("No valid location returned for IP address '%s' for "
                     "team '%s' with ref id '%s'"
                     % (self.request.client_addr, team, ref_id))
            ret["success"] = False
            ret["msg"] = ("No location found. Try a different IP from that "
                          "range.")
            return ret
        ret["location"] = loc
        try:
            t = transaction.savepoint()
            team.flags.append(loc)
            DBSession.flush()
        except Exception:
            ret["msg"] = "Location already registered."
            t.rollback()
        else:
            ret["msg"] = "Location successfully registered."
        return ret


class TeamFlag(Base):
    """
    Represent a quasi-many-to-many relationship between teams and flags. But
    the flags table is only present as a module-global variable and not in the
    database as it can be considered static (see :func:`install` for possible
    caveats).

    Recommended access to this is just going through a teams ``flags``
    attribute as it directly represents the flags already solves as a list of
```

```
    strings.

    .. todo::
        Once list is turned into a set of strings, update this documentation
        accordingly.
    """
    __tablename__ = 'team_flag'
    team_id = Column(Integer, ForeignKey('team.id'), primary_key=True)
    flag = Column(Unicode(2), primary_key=True)
    team = relationship("Team",
                        backref=backref("team_flags",
                                        cascade="all, delete-orphan"))

    def __init__(self, flag, **kwargs):
        kwargs["flag"] = flag
        Base.__init__(self, **kwargs)


class GeoIP(Base):
    """
    A mapping of an IP range to country codes. IP ranges are integers as they
    are natively anyway (4 blocks of 8 bit) and are stored this way for easier
    comparison.
    """
    ip_range_start = Column(BigInteger, primary_key=True,
                            autoincrement=False)
    ip_range_end = Column(BigInteger, nullable=False, unique=True, index=True)
    country_code = Column(Unicode(2), nullable=False)

    @staticmethod
    def ip_str(int_ip):
        """
        Turn an IP integer (such as those stored in the database) into a string
        for easier human-readable representation.
        """
        hex_ = hex(int_ip)[2:]
        if hex_.endswith("L"):
            hex_ = hex_[:-1]
        return socket.inet_ntoa(hex_.zfill(8).decode("hex"))

    @staticmethod
    def ip_int(str_ip):
        """
        Turn a human-readable string IP addressinto an integer IP address.
        """
        return int(socket.inet_aton(str_ip).encode("hex"), 16)

    @validates('ip_range_start', 'ip_range_end')
    def check_ip_range(self, key, ip):
        assert ip <= 0xFFFFFFFF
        assert ip >= 0
        return ip


def display(challenge, request):
    """
    Render the output for the challenge view. Displays a description and a
    grid of flags that can be visited.
```

```
    """
    from fluxscoreboard.models.team import get_team
    flags = []
    team = get_team(request)
    solved_flags = 0
    team_flags = set(team.flags) if team else set()
    for row in xrange(15):
        flag_row = []
        for col in xrange(15):
            index = row * 15 + col
            if index < len(flag_list):
                flag = flag_list[index]
                visited = flag in team_flags
                if visited:
                    solved_flags += 1
                flag_row.append((flag, visited))
        flags.append(flag_row)
    params = {'challenge': challenge,
              'flags': flags,
              'flag_stats': (solved_flags, len(flag_list)),
              'team': team}
    return render('dynamic_flags.mako', params, request)


def points_query(cls=None):
    """
    Returns a scalar query element that can be used in a ``SELECT`` statement
    to be added to the points query. The parameter ``cls`` can be anything
    that SQLAlchemy can correlate on. If left empty, it defaults to the
    standard :cls`fluxscoreboard.models.team.Team`, which is normally fine.
    However, if multiple teams are involved (as with the ranking algorithm)
    one might pass in an alias like this:

    .. code-block:: python
        inner_team = aliased(Team)
        dynamic_points = flags.points_query(inner_team)

    This will then correlate on a specific alias of ``Team`` instead of the
    default class.
    """
    if cls is None:
        from fluxscoreboard.models.team import Team
        cls = Team
    subquery = (DBSession.query(func.count('*')).
                filter(TeamFlag.team_id == cls.id).
                correlate(cls))
    return func.coalesce(subquery.as_scalar(), 0)


def points(team):
    return len(team.flags)


def get_location(ip):
    query = (DBSession.query(GeoIP.country_code).
             filter(GeoIP.ip_range_start <= GeoIP.ip_int(ip)).
             filter(GeoIP.ip_range_end >= GeoIP.ip_int(ip)))
    country_code, = query.first() or ("",)
```

```
    if country_code not in flag_list:
        log.info("Retrieved invalid country code '%s' for IP address %s. "
                 % (country_code, ip))
        return None
    else:
        return country_code


def title():
    return "Geolocation Flags (%s)" % __name__


def install(connection, with_update=True):
    geoip_fname = 'GeoIPCountryWhois.csv'
    geoip_file = os.path.join(ROOT_DIR, 'data', geoip_fname)
    if with_update:
        try:
            r = requests.get(db_url)
        except RequestException as e:
            log.error("Could not download current database because requests "
                      "threw an exception. This only means that the database will "
                      "not be up to date but we will use the old cached version. "
                      "Requests reported the following: '%s'" % e)
        else:
            tmpdir = mkdtemp()
            zipname = os.path.join(tmpdir, os.path.basename(db_url))
            with open(zipname, "w") as f:
                f.write(r.content)
            zip_ = zipfile.ZipFile(zipname)
            zip_.extractall(tmpdir)
            extracted_csv = os.path.join(tmpdir, geoip_fname)
            shutil.move(extracted_csv, geoip_file)
            shutil.rmtree(tmpdir)
    data = []
    available_country_codes = set()
    with open(geoip_file) as f:
        csv_ = csv.reader(f)
        for row in csv_:
            ip_int_start = int(row[2])
            ip_int_end = int(row[3])
            country_code = unicode(row[4].lower())
            if country_code not in flag_list:
                if country_code in flag_exceptions:
                    # Don't import it
                    continue
                else:
                    raise ValueError("The country code '%s' is not in the "
                                     "list of flags. It has the following "
                                     "data attached: '%s'"
                                     % (country_code, row))
            available_country_codes.add(country_code)
            item = {'ip_range_start': ip_int_start,
                    'ip_range_end': ip_int_end,
                    'country_code': country_code}
            data.append(item)
    log.info("Adding %d rows to database" % len(data))
    dialect = connection.dialect.name
    if dialect == "sqlite":
```

```
        chunk_size = 300
    elif dialect == "mysql":
        chunk_size = 10000
    else:
        chunk_size = len(data)

    while data:
        connection.execute(GeoIP.__table__.insert().values(data[:chunk_size]))
        data = data[chunk_size:]
    unreachable_countries = set(flag_list) - available_country_codes
    if unreachable_countries:
        log.warning("There are a number of countries that will not be "
                    "reachable for the teams because it is not present in our "
                    "database even though we display their flag. These "
                    "are the country codes that cannot be reached: '%s'"
                    % list(unreachable_countries))


flag_list = ['ad', 'ae', 'af', 'ag', 'ai', 'al', 'am', 'ao', 'aq',
             'ar', 'as', 'at', 'au', 'aw', 'az', 'ba', 'bb', 'bd', 'be',
             'bf', 'bg', 'bh', 'bi', 'bj', 'bm', 'bn', 'bo', 'br', 'bs',
             'bt', 'bw', 'by', 'bz', 'ca', 'cg', 'cf', 'cd', 'ch', 'ci',
             'ck', 'cl', 'cm', 'cn', 'co', 'cr', 'cu', 'cv', 'cy', 'cz',
             'de', 'dj', 'dk', 'dm', 'do', 'dz', 'ec', 'ee', 'eg', 'eh',
             'er', 'es', 'et', 'fi', 'fj', 'fm', 'fo', 'fr', 'ga', 'gb',
             'gd', 'ge', 'gg', 'gh', 'gi', 'gl', 'gm', 'gn', 'gp', 'gq',
             'gr', 'gt', 'gu', 'gw', 'gy', 'hk', 'hn', 'hr', 'ht', 'hu',
             'id', 'mc', 'ie', 'il', 'im', 'in', 'iq', 'ir', 'is', 'it',
             'je', 'jm', 'jo', 'jp', 'ke', 'kg', 'kh', 'ki', 'km', 'kn',
             'kp', 'kr', 'kw', 'ky', 'kz', 'la', 'lb', 'lc', 'li', 'lk',
             'lr', 'ls', 'lt', 'lu', 'lv', 'ly', 'ma', 'md', 'me', 'mg',
             'mh', 'mk', 'ml', 'mm', 'mn', 'mo', 'mq', 'mr', 'ms', 'mt',
             'mu', 'mv', 'mw', 'mx', 'my', 'mz', 'na', 'nc', 'ne', 'ng',
             'ni', 'nl', 'no', 'np', 'nr', 'nz', 'om', 'pa', 'pe', 'pf',
             'pg', 'ph', 'pk', 'pl', 'pr', 'ps', 'pt', 'pw', 'py', 'qa',
             're', 'ro', 'rs', 'ru', 'rw', 'sa', 'sb', 'sc', 'sd', 'se',
             'sg', 'si', 'sk', 'sl', 'sm', 'sn', 'so', 'sr', 'st', 'sv',
             'sy', 'sz', 'tc', 'td', 'tg', 'th', 'tj', 'tl', 'tm', 'tn',
             'to', 'tr', 'tt', 'tv', 'tw', 'tz', 'ua', 'ug', 'us', 'uy',
             'uz', 'va', 'vc', 've', 'vg', 'vi', 'vn', 'vu', 'ws', 'ye',
             'za', 'zm', 'zw']


# These are flags that exist in the original database but we do not recognize
# them
flag_exceptions = set(['eu', 'a2', 'yt', 'ap', 'tk', 'wf', 'cw', 'ss', 'a1',
                       'sh', 'cx', 'mf', 'gs', 'gf', 'cc', 'bl', 'nf', 'um',
                       'sj', 'bq', 'sx', 'mp', 'io', 'tf', 'ax', 'fk', 'pn',
                       'nu', 'pm'])
```

3.1.7 Send Mail

Sending mails is done with the help of pyramid_mailer. To send a mail, you need to get the mailer, create a message and add it to the sending queue. Here's how you do it:

```
from pyramid_mailer import get_mailer
from pyramid_mailer.message import Message

mailer = get_mailer(request)
message = Message(subject="My Subject",
                  recipients=["test@example.com"],
                  body="Test Message",
                 )
mailer.send(message)
```

You can also send html mails. For this it is recommended to use the templating system and render it to HTML for a better display. Here's how you do that:

```
from pyramid.renderers import render
# Code from above
...
message = Message(subject="My HTML Mail",
                  recipients=["test@example.com"],
                  html=render('mail_test.mako',
                              {},
                              request=request,
                             )
                 )
# Code from above
...
```

This way you can send an email with an HTML body that is comfortably rendered from a template. The empty dictionary is the data passed to the template and the request comes from the web application.

Transaction Support

The mailing system support transactions and thus makes sure that mails are only sent when the transaction succeeds. Thus, when an exception occurs, the mail is not sent. This provides a high level of integration and allows us to send mails thoughtlessly but be sure they are never sent until the actual application succeeded. However, this has a small performance impact on the response time: The email is only dispatched once the application completes and that means it cannot be deferred in any way: The application will only return once the email is sent. This results in a small noticeable delay.

3.1.8 Unit Testing

Todo

Write chapter on how to unittest with py.test

The API documentation basically dumps docstrings into here. And some functions don't even have that. Look at it, cherish that there is something. Then look at the code, it might speak to you.

3.1.9 `fluxscoreboard.models`

class `fluxscoreboard.models.`**`Base`**(***kwargs*)

Base class for all ORM classes. Uses `BaseCFG` configuration.

class fluxscoreboard.models.**BaseCFG**
Class that contains custom configuration for a sqlalchemy.ext.declarative.declarative_base() to be used with the ORM. It automatically figures out a tablename (thus no need to set __tablename__).

fluxscoreboard.models.**DBSession = <sqlalchemy.orm.scoping.scoped_session object at 0x7feb77628550>**
Database session factory. Returns the current threadlocal session.

class fluxscoreboard.models.**RootFactory**(*request*)
Skeleton for simple ACL permission protection.

Challenge Models & Functions

class fluxscoreboard.models.challenge.**Challenge**(***kwargs*)
A challenge in the system.

Attributes: id: The primary key column.

title: Title of the challenge.

text: A description of the challenge.

solution: The challenge's solution

points: How many points the challenge is worth.

online: Whether the challenge is online.

manual: If the points for this challenge are awareded manually.

category_id: ID of the associated category.

category: Direct access to the Category.

author: A simple string that contains an author (or a list thereof).

dynamic: Whether this challenge is dynamically handled. At the default of False this is just a normal challenge, otherwise, the attribute module must be set.

module: If this challenge is dynamic, it must provide a valid dotted python name for a module that provides the interface for validation and display. The dotted python name given here will be prefixed with fluxscoreboard.dynamic_challenges. from which the module will be loaded and made available on using it.

module: Loads the module from the module name and returns it.

published: Whether the challenge should be displayed in the frontend at all.

points
The points of a challenge which is either the value assigned to it or, if the challenge is manual, the manual_challenge_points object to indicate that the points are manually assigned.

class fluxscoreboard.models.challenge.**Category**(***kwargs*)
A category for challenges.

Attributes: id: Primary key of category.

name: Name of the category.

challenges: List of challenges in that category.

class fluxscoreboard.models.challenge.**Submission**(***kwargs*)
A single submission. Each entry means that this team has solved the corresponding challenge, i.e. there is no solved flag: The existence of the entry states that.

Attributes: `team_id`: Foreign primary key column of the team.

`challenge_id`: Foreign primary key column of the challenge.

`timestamp`: A UTC-aware `datetime.datetime` object. When assigning a value always pass either a timezone-aware object or a naive UTC datetime. Defaults to `datetime.datetime.utcnow()`.

`bonus`: How many bonus points were awared.

`team`: Direct access to the team who solved this challenge.

`challenge`: Direct access to the challenge.

class `fluxscoreboard.models.challenge.`**Feedback**(***kwargs*)

`fluxscoreboard.models.challenge.`**get_all_challenges**()
Return a query that gets **all** challenges.

`fluxscoreboard.models.challenge.`**get_online_challenges**()
Return a query that gets only those challenges that are online.

`fluxscoreboard.models.challenge.`**get_submissions**()
Creates a query to **eagerly** load all submissions. That is, all teams and challenges that are attached to the submissions are fetched with them.

`fluxscoreboard.models.challenge.`**get_all_categories**()
Get a list of all available categories.

`fluxscoreboard.models.challenge.`**check_submission**(*challenge*, *solution*, *team*, *settings*)
Check a solution for a challenge submitted by a team and add it to the database if it was correct.

Args: `challenge`: An instance of `Challenge`, the challenge to check the solution for.

`solution`: A string, the proposed solution for the challenge.

`team`: Team that submitted the solution.

Returns: A tuple of `(result, msg)`. `result` indicates whether the solution was accpeted (and added to the database) or not. The message returns a string with either a result (if `result == False`) or a congratulations message.

`fluxscoreboard.models.challenge.`**manual_challenge_points = <ManualChallengePoints instance>**
A static value that is returned instead of an actual number of points.

`fluxscoreboard.models.challenge.`**update_playing_teams**(*connection*)
Update the number of playing teams whenever it changes.

`fluxscoreboard.models.challenge.`**update_challenge_points**(*connection*, *update_team_count=True*)
Update the points on each challenge to reflect their current worth.

Country Models & Functions

class `fluxscoreboard.models.country.`**Country**(***kwargs*)
A country in the database. Basically only a name for different locations of teams.

`fluxscoreboard.models.country.`**get_all_countries**()
Get a query that fetches a list of all countries from the database.

News Models & Functions

class fluxscoreboard.models.news.**News**(**kwargs)
A single announcement, either global or for a challenge, depending on the challenge_id attribute.

Attributes: id: The primary key.

timestamp: A UTC-aware datetime.datetime object. When assigning a value always pass either a timezone-aware object or a naive UTC datetime. Defaults to datetime.datetime.utcnow().

message: The text of the announcement.

published: Whether the announcement is displayed in the frontend.

challenge_id: If present, which challenge this announcement belongs to.

challenge: Direct access to the challenge, if any.

class fluxscoreboard.models.news.**MassMail**(**kwargs)
An entry of a mass mail that was sent.

Attributes: id: The primary key.

timestamp: A UTC-aware datetime.datetime object. When assigning a value always pass either a timezone-aware object or a naive UTC datetime. Defaults to datetime.datetime.utcnow().

subject: The subject of the mail

message: The body of the mail

recipients: A list of recipients that have recieved this mail. Internally this is stored as a json encoded list.

from_: The address which was used as the From: field of the mail.

fluxscoreboard.models.news.**get_published_news**()

Team Models & Functions

class fluxscoreboard.models.team.**Team**(**kwargs)
A team represented in the database.

Attributes: id: Primary key

name: The name of the team.

password: The password of the team. If setting the password, pass it as cleartext. It will automatically be encrypted and stored in the database.

email: E-Mail address of the team. Verified if team is active.

country_id: Foreign Key specifying the location of the team.

local: Whether the team is local at the conference.

token: Token for E-Mail verification.

reset_token: When requesting a new password, this token is used.

challenge_token: Unique token for each team they can provide to a challenge so this challenge can do rate-limiting or banning or whatever it wants to do.

active: Whether the team's mail address has been verified and the team can actively log in.

timezone: A timezone, specified as a string, like "Europe/Berlin" or something that, when coerced to unicode, turns out as a string like this. Must be valid timezone.

acatar_filename: The filename under which the avatar is stored in the static/images/avatars directory.

size: The size of the team.

country: Direct access to the teams fluxscoreboard.models.country.Country attribute.

get_solvable_challenges()
Return a list of challenges that the team can solve right now. It returns a list of challenges that are

- online
- unsolved by the current team
- not manual or dynamic (i.e. solvable by entering a solution)

get_unsolved_challenges()
Return a query that produces a list of all unsolved challenges for a given team.

validate_password(*password*)
Validate the password agains the team. If it matches return True else return False.

fluxscoreboard.models.team.**get_all_teams**()
Get a query that returns a list of all teams.

fluxscoreboard.models.team.**get_active_teams**()
Get a query that returns a list of all active teams.

fluxscoreboard.models.team.**get_team_solved_subquery**(*team_id*)
Get a query that searches for a submission from a team for a given challenge. The challenge is supposed to come from an outer query.

Example usage:

```
team_solved_subquery = get_team_solved_subquery(team_id)
challenge_query = (DBSession.query(Challenge,
                                   team_solved_subquery))
```

In this example we query for a list of all challenges and additionally fetch whether the currenttly logged in team has solved it.

fluxscoreboard.models.team.**get_number_solved_subquery**()
Get a subquery that returns how many teams have solved a challenge.

Example usage:

```
number_of_solved_subquery = get_number_solved_subquery()
challenge_query = (DBSession.query(Challenge,
                                   number_of_solved_subquery)
```

Here we query for a list of all challenges and additionally fetch the number of times it has been solved. This subquery will use the outer challenge to correlate on, so make sure to provide one or this query makes no sense.

fluxscoreboard.models.team.**get_team**(*request*)
Get the currently logged in team. Returns None if the team is invalid (e.g. inactive) or noone is logged in or if the scoreboard is in archive mode.

fluxscoreboard.models.team.**get_team_by_id**(*team_id*)

fluxscoreboard.models.team.**register_team**(*form*, *request*)
Create a new team from a form and send a confirmation email.

Args: form: A filled out fluxscoreboard.forms.front.RegisterForm.

request: The corresponding request.

Returns: The `Team` that was created.

`fluxscoreboard.models.team.`**`send_activation_mail`**(*team*, *request*)
Send activation mail to particular team.

`fluxscoreboard.models.team.`**`confirm_registration`**(*token*)
For a token, check the database for the corresponding team and activate it if found.

Args: `token`: The token that was sent to the user (a string)

Returns: Either `True` or `False` depending on whether the confirmation was successful.

`fluxscoreboard.models.team.`**`login`**(*email*, *password*)
Check a combination of credentials for validaity and either return a reason why it failed or return the logged in team.

Args: `email`: The email address of the team.

`password`: The corresponding password.

Returns: A three-tuple of `(result, message, team)`. `result` indicates whether the login was successful or not. In case of failure `msg` contains a reason why it failed so it can be logged (but **not** printed - we don't want to give any angle to an attacker). If the login was successful, `msg` is `None`. Finally, if the login succeeded, `team` contains the found instance of `Team`. If login failed, `team` is `None`.

`fluxscoreboard.models.team.`**`password_reminder`**(*email*, *request*)
For an email address, find the corresponding team and send a password reset token. If no team is found send an email that no user was found for this address.

`fluxscoreboard.models.team.`**`check_password_reset_token`**(*token*)
Check if an entered password reset token actually exists in the database.

`fluxscoreboard.models.team.`**`TEAM_GROUPS = [u'group:team']`**
Groups are just fixed: If a team is logged in it belongs to these groups.

`fluxscoreboard.models.team.`**`groupfinder`**(*userid*, *request*)
Check if there is a team logged in, and if it is, return the default `TEAM_GROUPS`.

class `fluxscoreboard.models.team.`**`TeamIP`**(***kwargs*)

`fluxscoreboard.models.team.`**`update_score`**(*connection*, *update_all=True*)
Update the score of all teams. If `update_all` is set, the points for all challenges are updated beforehand as well.

This is your one-shot function to create up-to-date points for everything.

Settings Models & Functions

class `fluxscoreboard.models.settings.`**`Settings`**(***kwargs*)
Represents application settings. Do **not** insert rows of this. There must always be only **one** row which is updated. This is preferred to having multiple rows with only key->value because with this we can enforce types and give an overview over available settings.

The most straightforward usage is calling `get()` to retrieve the current settings which you can then also edit and they will be saved automatically.

The following settings are available:

`submission_disabled`: A boolean that describes whether currently submissions are allowed or disabled. Default: `False`

`ctf_start_date`: A timezone-aware datetime value that describes when the CTF should start. Before that, the application will behave differently, e.g. may not allow login.

`ctf_end_date`: When the CTF will end. Same type as `ctf_start_date`.

`ctf_started`: This is a property that can only be read and just compares `ctf_start_date` with the current time to find out whether the CTF has already started or not.

`archive_mode`: When the scoreboard is in archive mode, the frontend will not allow alteration to the database. Additionally, the whole system is public so everyone can get their solutions checked. This is then verified and the result is returned, but it is not added to the database. The following things will change in archive mode:

- No registration
- No login
- Start / End times ignored
- Solutions can be submitted but will only return the result, not enter something into the databse
- Challenges are public in addition to the scoreboard

`ctf_state`: Which time state the CTF currently is in. Relevant for permissions etc.

Custom Column Types

class `fluxscoreboard.models.types.`**`TZDateTime`**(**args*, ***kwargs*)
Coerces a tz-aware datetime object into a naive utc datetime object to be stored in the database. If already naive, will keep it.

On return of the data will restore it as an aware object by assuming it is UTC.

Use this instead of the standard `sqlalchemy.types.DateTime`.

class `fluxscoreboard.models.types.`**`Timezone`**(**args*, ***kwargs*)
Represent a timezone, storing it as a unicode string but giving back a `datetime.tzinfo` instance.

class `fluxscoreboard.models.types.`**`Module`**(**args*, ***kwargs*)
Represent a python module from the `dynamic_challenges` submodule. Input is a string but the return value will always be a real python module.

3.1.10 `fluxscoreboard.views`

This package contains all views, organized in classes. It is divided into frontend views in `fluxscoreboard.views.front` and backend views in `fluxscoreboard.views.admin`.

3.1.11 `fluxscoreboard.views.front`

class `fluxscoreboard.views.front.`**`BaseView`**(*request*)
A base class for all other frontpage views. If you build a frontend view class, derive from this. You can access the current logged in team from the `team` property. A list of menu items will be present in `menu`, which returns different items based on whether the user is logged in.

`current_state`
A pair of `ctf_state, logged_in` where `ctf_state` represents the current state as per settings and `logged_in` is a boolean that shows whether the user is currently logged in to a team.

`menu`
Get the current menu items as a list of tuples `(view_name, title)`.

title
From the menu get a title for the page.

class fluxscoreboard.views.front.**SpecialView**(*request*)
Contains special views, i.e. pages for status codes like 404 and 403.

forbidden()
A forbidden view that only returns a 403 if the user isn't logged in otherwise just redirect to login.

notfound()
Renders a 404 view that integrates with the page. The attached template is `404.mako`.

class fluxscoreboard.views.front.**FrontView**(*request*)
All views that are part of the actual page, i.e. the scoreboard and anything surrounding it. Most views in here **must** be protected by `logged_in_view` and not the usual `pyramid.view.view_config`. Some exceptions may exist, such as the `ref()` view.

challenge()
A view of a single challenge. The query is very similar to that of `challenges()` with the limitation that only one challenge is fetched. Additionally, this page displays a form to enter the solution of that challenge and fetches a list of announcements for the challenge.

challenges()
A list of all challenges similar to the scoreboard view in a table. It has a very complex query that gets all challennges together with a boolean of whether the current team has solved it, and the number of times this challenge was solved overall. This list of tuples `(challenge, team_solved, number_solved_total)` is then given to the template and rendered.

home()
A view for the page root which just redirects to the `scoreboard` view.

news()
Just a list of all announcements that are currently published, ordered by publication date, the most recent first.

scoreboard()
The central most interesting view. This contains a list of all teams with their points, sorted with the highest points on top. The most complex part of the query is the query that calculates the sum of points right in the SQL.

submit_solution()
A special form that, in addition to the form provided by `challenge()`, allows a user to submit solutions for a challenge. The difference here is that the challenge is chosen from a select list. Otherwise it is basically the same and boils down to the same logic.

teams()
Only a list of teams.

class fluxscoreboard.views.front.**UserView**(*request*)
This view is used for everything user- (or in our case team-) related. It contains stuff like registration, login and confirmation. It depends on the purpose of the view whether to make it a `logged_in_view` or a `pyramid.view.view_config`.

confirm_registration(*self_wrap*, **args*, ***kwargs*)
After a registration has been made, the team recieves a confirmation mail with a token. With this token the team activates its account by visitng this view. It fetches the team corresponding to the token and activates it.

login(*self_wrap*, **args*, ***kwargs*)
A view that logs in the user. Displays a login form and in case of a `POST` request, handles the login by checking whether it is valid. If it is, the user is logged in and redirected to the frontpage.

logout()
A simple view that logs out the user and redirects to the login page.

profile()
Here a team can alter their profile, i.e. change their email, password, location or timezone. The team name is fixed and can only be changed by administrators.

register(*self_wrap*, **args*, ***kwargs*)
Display and handle registration of new teams.

3.1.12 fluxscoreboard.views.admin

class fluxscoreboard.views.admin.**AdminView**(*request*)
The view for everything corresponding to administration. The views here are not protected because they must be protected from the outside, i.e. HTTP Authorization or similar.

_admin_delete(*route_name*, *DatabaseClass*, *title*, *title_plural=None*)
Generic function to delete a single item from the database. Its arguments have the same meaning as explained in `_admin_list()` with the addition of `title_plural` which is just a pluraized version of the `title` argument. Also returns something that can be returned directly to the application.

Note: To avoid problems with cascade instead of just emitting an SQL `DELETE` statement, this queries for all affected objects (should be one) and deletes them afterwards. This ensures that the Python-side cascades appropriately delete all dependent objects.

_admin_edit(*route_name*, *FormClass*, *DatabaseClass*, *title*)
A generic function for a view that is invoked after an edit (or add) has been performed. It is separate from that of `AdminView._admin_list()` to keep the code cleaner. It has the same parameters and return types but can only be invoked as a `POST`.

_admin_list(*route_name*, *FormClass*, *DatabaseClass*, *title*, *change_query=None*)
A generic function for all views that contain a list of things and also a form to edit or add entries.

Note: This only handles items with their own single primary key and not anything with composite foreign keys.

Args: `route_name`: A string containing the name of the route to which the admin should be redirected aver an edit was saved. For example `"admin_challenges"`.

`FormClass`: The class of the form that should be displayed at the bottom of the page to edit or add items. For example `fluxscoreboard.forms.admin.ChallengeForm`.

`DatabaseClass`: The ORM class from the model that is used to add and fetch items. For example `fluxscoreboard.models.challenge.Challenge`.

`title`: A string that expresses a singular item, for example `"Challenge"`. Will be used for flash messages.

`change_query`: A function that receives one parameter (a query), modifies it and returns the new query. May for example be used to modify the order or refine results. Optional.

Returns: A dictionary or similar that can be directly returned to the application to be rendered as a view.

An example usage might be like this:

```
def challenges(self):
return self._admin_list('admin_challenges', ChallengeForm,
                        Challenge, "Challenge")
```

_admin_toggle_status(*route_name*, *DatabaseClass*, *title=u''*, *status_types={False: False*, *True: True}*, *status_variable_name=u'published'*, *status_messages={False: u'Unpublished %(title)s'*, *True: u'Published %(title)s'}*)

Generic function that allows to toggle a special status on the challenge. By default it toggles the `published` property of any given item.

Many arguments are the same as in `_admin_list()` with these additional arguments:

`status_types`: A two-element dictionary that contains `True` and `False` as keys and any value that describes the given status. For example: If the "unpublished" status is described by the string "offline", then the value for key `False` would be `"offline"`. It depends on the database model, which value is used here. The default is just a boolean mapping.

`status_variable_name`: What is the name of the property in the model that contains the status to be changed. Defaults to "published".

`status_messages`: The same keys as for `status_types` but as values contains messages to be displayed, based on which action was the result. Gives access to the `title` variable via `%(title)s` inside the string. The defaults are sensible values for the the default status. Most likely you want to change this if changing `status_variable_name`.

Returns: A dictionary or similar that can be directly returned from a view.

_list_retparams(*page*, *form*, *is_new=None*)

Get a dictionary of parameters to return to a list + edit form view.

`page` must be an instance of `webhelpers.paginate.Page` and `form` must be an instance of the form to be displayed (whatever that is).

admin()

Root view of admin page, redirect to announcements.

categories()

A view to list, add and edit categories. Implemented with `_admin_list()`.

category_delete()

A view to delete a category. Implemented with `_admin_delete()`.

category_edit()

This view accepts an edit form, handles it and reacts accordingly (either redirect or, on error, show errors). Implemented with `_admin_edit()`.

chalenge_toggle_published()

Switch a challenge between published and unpublished.

challenge_delete()

A view to delete a challenge. Implemented with `_admin_delete()`.

challenge_edit()

This view accepts an edit form, handles it and reacts accordingly (either redirect or, on error, show errors). Implemented with `_admin_edit()`.

challenge_feedback()

Display feedback list.

challenge_toggle_status()
A view to toggle the online/offline status of a challenge. Implemented with `_admin_toggle_status()`.

challenges()
A view to list, add and edit challenges. Implemented with `_admin_list()`.

items(*DatabaseClass*)
Construct a simple query to the database. Even though it is dead simple it is factored out because it is used in more than one place.

massmail()
Send a massmail to all users in the system. It also stores the sent mail and its recipients in the database to keep a permanent record of sent messages.

massmail_single()
View a single massmail that was sent.

news()
A view to list, add and edit announcements. Implemented with `_admin_list()`.

news_delete()
A view to delete an announcement. Implemented with `_admin_delete()`.

news_edit()
This view accepts an edit form, handles it and reacts accordingly (either redirect or, on error, show errors). Implemented with `_admin_edit()`.

news_toggle_status()
A view to publish or unpublish an announcement. Implemented with `_admin_toggle_status()`.

page(*items*)
Return a `webhelpers.paginate.Page` instance for an `items` iterable.

redirect(*route_name*, *current_page=None*)
For a given route name and page number get a redirect to that page. Convenience method for writing clean code.

settings()
Adjust runtime application settings.

submissions()
List, add or edit a submission. This is different because it consists of composite foreign keys and thus needs separate though similar logic. But in the end it is basically the same functionality as with the other list views.

submissions_delete()
Delete a submission.

team_activate()
De-/Activate a team.

team_cleanup()
Remove ALL inactive teams. Warning: **DANGEROUS**

team_delete()
Delete a team.

team_edit()
This view accepts an edit form, handles it and reacts accordingly (either redirect or, on error, show errors). Implemented with `_admin_edit()`.

team_ips()
A list of IPs per team.

team_regenerate_token()
Manually regenerate the teams challenge token

team_resend_activation()
Resend the activation mail for a team.

team_toggle_local()
Toggle the local attribute of a team.

teams()
List, add or edit a team.

test_login()
If there is at least one team, log in as it to see the page.

3.1.13 fluxscoreboard.forms

Form classes exist to define, render and validate form submission. See *Displaying and Using Forms* for details.

Custom Forms

class fluxscoreboard.forms.**CSRFForm**(*formdata=None*, *obj=None*, *prefix=u''*, *csrf_context=None*, ***kwargs*)

Todo

Document.

3.1.14 fluxscreoboard.forms.front

Frontend Forms

This module contains all forms for the frontend. Backend forms can be found in fluxscoreboard.forms.admin. The forms here and there are described as they should behave, e.g. "Save the challenge", however, this behaviour has to be implemented by the developer and is not done automatically by it. However, validatation restrictions (e.g. length) are enforced. But alone, without a database to persist them, they are mostly useless.

class fluxscoreboard.forms.front.**RegisterForm**(*formdata=None*, *obj=None*, *prefix=u''*, *csrf_context=None*, ***kwargs*)

Registration form for new teams.

Attrs: name: The name of the team, required and must have a length between 1 and 255.

email: The teams E-Mail address, required, must be the same as email_repeat, must have a length between 5 and 255. It is also checked that this email is not registered yet.

email_repeat: Must have the same value as email.

password: The team's password, must be between 8 and 1024 characters long.

password_repeat: Must have the same value as password.

country: A list of available countries in the database to choose from.

`timezone`: The timezone to apply for the team. When not selected, UTC is used, otherwise this localizes times displayed on the frontpage.

`submit`: The submit button.

class `fluxscoreboard.forms.front.`**`LoginForm`**(*formdata=None*, *obj=None*, *prefix=u''*, *csrf_context=None*, ***kwargs*)

Login form for teams that are activated.

Attrs: `email`: Login email address.

`password`: Login password.

`login`: Submit button.

class `fluxscoreboard.forms.front.`**`ForgotPasswordForm`**(*formdata=None*, *obj=None*, *prefix=u''*, *csrf_context=None*, ***kwargs*)

A form to get an email for a forgotten password.

class `fluxscoreboard.forms.front.`**`ResetPasswordForm`**(*formdata=None*, *obj=None*, *prefix=u''*, *csrf_context=None*, ***kwargs*)

A form to finish a started reset process.

class `fluxscoreboard.forms.front.`**`ProfileForm`**(*formdata=None*, *obj=None*, *prefix=u''*, *csrf_context=None*, ***kwargs*)

A form to edit a team's profile.

Attrs: `email`: The email address. Required

`old_password`: The old password, needed only for a password change.

`password`: The password. Optional, only needed if wanting to change.

`password_repeat`: Repeat the new password.

`avatar`: Display an avatar and upload a new one.

`country`: Change location. Required.

`timezone`: Change timezone. Required.

`submit`: Save changes

`cancel`: Abort

class `fluxscoreboard.forms.front.`**`SolutionSubmitForm`**(*formdata=None*, *obj=None*, *prefix=u''*, *csrf_context=None*, ***kwargs*)

A form to submit a solution for a challenge on a single challenge view. This form does not keep track of which challenge this is.

Attrs: `solution`: The proposed solution for the challenge. Required.

`submit`: Submit the solution.

class `fluxscoreboard.forms.front.`**`SolutionSubmitListForm`**(*formdata=None*, *obj=None*, *prefix=u''*, *csrf_context=None*, ***kwargs*)

A form to submit a solution for any challenge selected from a list. Keeps track of which challenge the solution was submitted for. Subclass of `SolutionSubmitForm` and derives all attributes from it. New attributes defined here:

`challenge`: A list of challenges to choose from. Required.

class fluxscoreboard.forms.front.**FeedbackForm**(*formdata=None*, *obj=None*, *prefix=u''*, *csrf_context=None*, ***kwargs*)

3.1.15 fluxscoreboard.forms.admin

class fluxscoreboard.forms.admin.**NewsForm**(*formdata=None*, *obj=None*, *prefix=u''*, *csrf_context=None*, ***kwargs*)

Form to add or edit an announcement.

Attrs: message: The announcement message. Required.

published: Whether the announcement should be published. Required.

challenge: A list of challenges to choose from, alternatively a blank field with text "– General Announcement –" to not assign it to a specific challenge.

id: Hidden field keeps track of challenge.

submit: Save the announcement to the database.

cancel: Do not save the announcement.

class fluxscoreboard.forms.admin.**ChallengeForm**(*formdata=None*, *obj=None*, *prefix=u''*, *csrf_context=None*, ***kwargs*)

Form to add or edit a challenge.

Attrs: title: Title of the challenge. Required.

text: Description of the challenge. Required.

solution: Solution. Required.

base_points: How many base points is this challenge worth? Only required if the challenge is not manual, otherwise not allowed to be anything other than 0 or empty.

online: If the challenge is online.

manual: If the points for this challenge are given manually.

id: Track the id of the challenge.

submit: Save challenge.

cancel: Abort saving.

class fluxscoreboard.forms.admin.**CategoryForm**(*formdata=None*, *obj=None*, *prefix=u''*, *csrf_context=None*, ***kwargs*)

Form to add or edit a category.

Attrs: name: Title of the category. Required.

id: Track the id of the category.

submit: Save category.

cancel: Abort saving.

class fluxscoreboard.forms.admin.**TeamForm**(*formdata=None*, *obj=None*, *prefix=u''*, *csrf_context=None*, ***kwargs*)

Form to add or edit a team. The same restrictions as on fluxscoreboard.forms.front.RegisterForm apply.

Attrs: name: The name of the name of the team. Required.

password: The team's password. Only required if the team is new.

`email`: E-Mail address. Required.

`country`: Location of team. Required.

`active`: If the team is active, i.e. able to log in.

`local`: If the team is local or remote.

`id`: Tracks the id of the team.

`submit`: Save the team.

`cancel`: Don't save.

class `fluxscoreboard.forms.admin.`**`IPSearchForm`**(*formdata=None*, *obj=None*, *prefix=u''*, *csrf_context=None*, ***kwargs*)

Form to search for an IP address and find the resulting team(s).

class `fluxscoreboard.forms.admin.`**`SubmissionForm`**(*formdata=None*, *obj=None*, *prefix=u''*, *csrf_context=None*, ***kwargs*)

Form to add or edit a submission of a team.

Attrs: `challenge`: The `fluxscoreboard.models.challenge.Challenge` to be chosen from a list. Required.

`team`: The `fluxscoreboard.models.team.Team` to be chosem from a list. Required.

`bonus`: How many bonus points the team gets. Defaults to 0.

`submit`: Save.

`cancel`: Abort.

class `fluxscoreboard.forms.admin.`**`MassMailForm`**(*formdata=None*, *obj=None*, *prefix=u''*, *csrf_context=None*, ***kwargs*)

A form to send a massmail to all users.

Attrs: `from_`: The sending address. Recommended to set it to `settings["default_sender"]`, e.g.:

```
if not form.from_.data:
    settings = self.request.registry.settings
    form.from_.data = settings["mail.default_sender"]
```

`subject`: A subject for the E-Mail.

`message`: A body for the E-Mail.

`submit`: Send the mail.

`cancel`: Don't send.

class `fluxscoreboard.forms.admin.`**`ButtonForm`**(*formdata=None*, *obj=None*, *prefix=u''*, *csrf_context=None*, *title=None*, ***kwargs*)

A form that gives a button and an ID. Useful for having a simple action that is identified in the forms `action` attribute. This provides CSRF support and the ability to POST submit commands such as edit or delete.

class `fluxscoreboard.forms.admin.`**`SubmissionButtonForm`**(*formdata=None*, *obj=None*, *prefix=u''*, *csrf_context=None*, *title=None*, ***kwargs*)

Special variant of `ButtonForm` that is tailored for the composite primary key table `submission`. Instead of having one `id` field it has one field `challenge_id` identifying the challenge and a field `team_id` identifiying the team.

class `fluxscoreboard.forms.admin.`**`TeamCleanupForm`**(*formdata=None*, *obj=None*, *prefix=u''*, *csrf_context=None*, *title=None*, ***kwargs*)

class fluxscoreboard.forms.admin.**SettingsForm**(*formdata=None*, *obj=None*, *prefix=u''*, *csrf_context=None*, ***kwargs*)

3.1.16 fluxscoreboard.forms._validators

Contains validators for forms with custom messages. Some validators are based on the original present validators, others are own creations with extended functionality.

fluxscoreboard.forms._validators.**email_unique_validator**(*form*, *field*)
A validator to make sure the entered email is unique and does not exist yet.

fluxscoreboard.forms._validators.**name_unique_validator**(*form*, *field*)
A validator to make sure the entered team name is unique and does not exist yet.

fluxscoreboard.forms._validators.**greater_zero_if_set**(*form*, *field*)

fluxscoreboard.forms._validators.**password_length_validator_conditional**(*form*, *field*)
A validator that only checks the length of the password if one was provided and otherwise just returns `True`. Used so an item can be edited without entering the password for the team.

fluxscoreboard.forms._validators.**password_required_if_new**(*form*, *field*)
A validator that only requires a password if the team is newly created, i.e. its id is `None`.

fluxscoreboard.forms._validators.**password_required_and_valid_if_pw_change**(*form*, *field*)
A validator that only requires a field to be set if a password change is intended, i.e. if the `password` field is set. It also checks that the entered password is correct.

fluxscoreboard.forms._validators.**password_max_length_if_set_validator**(*form*, *field*)
Only apply the `password_max_length_validator` if the field is set at all.

fluxscoreboard.forms._validators.**password_min_length_if_set_validator**(*form*, *field*)
Only apply the `password_min_length_validator` if the field is set at all.

fluxscoreboard.forms._validators.**required_or_not_allowed**(*field_list*, *validator=<wtforms.validators.Required object at 0x7feb73369850>*)
Enforces that a field is required _only_ if none of the fields in `field_list` are set. Pass in an alternative `validator` to allow for passing of validaton control down to another validator.

fluxscoreboard.forms._validators.**required_except**(*field_list*)
Enforces a required constraint only if none of the fields in `field_list` are set. The fields in `field_list` must be strings with names from other form fields.

fluxscoreboard.forms._validators.**not_dynamic**(*form*, *field*)
Checks that the "dynamic" checkbox is not checked.

fluxscoreboard.forms._validators.**only_if_dynamic**(*form*, *field*)
Enforces that this field is only allowed if the challenge is dynamic (and in that case it **must** be set).

fluxscoreboard.forms._validators.**dynamic_check_multiple_allowed**(*form*, *field*)
Checks if multiple fields are allowed and if they are not and there already is a challenge with this dynamic type, then fail.

class fluxscoreboard.forms._validators.**AvatarSize**(*max_size*, *unit=u'MB'*, *message=None*)
A validator class for the size of a file. Pass it a `max_size` to set the value of maximum size. The unit is

determined by the `unit` parameter and defaults to 'MB'. Supported units are 'B', 'KB' and 'GB'. Optionally, you can pass in a custom message which has access to the `max_size` and `unit` parameters: `"Maximum size: %(max_size)d%(unit)s"`.

class `fluxscoreboard.forms._validators.`**RecaptchaValidator**
Validates captcha by using reCaptcha API

3.1.17 fluxscoreboard.forms._fields

This module contains some custom fields and widgets.

class `fluxscoreboard.forms._fields.`**AvatarWidget**
A widget that renders the current avatar above the form to upload a new one.

class `fluxscoreboard.forms._fields.`**AvatarField**(*label=None*, *validators=None*, *filters=()*, *description=u''*, *id=None*, *default=None*, *widget=None*, *_form=None*, *_name=None*, *_prefix=u''*, *_translations=None*)
An avatar upload field with a display of an existing avatar.

class `fluxscoreboard.forms._fields.`**ButtonWidget**(**args*, ***kwargs*)

Todo

Document

---

class `fluxscoreboard.forms._fields.`**IntegerOrEvaluatedField**(*label=None*, *validators=None*, ***kwargs*)
A field that is basically an integer but with the added exception that, if the challenge is manual, it will contain the value `"evaulauted"` which is also valid. May also be empty.

class `fluxscoreboard.forms._fields.`**IntegerOrNoneField**(*label=None*, *validators=None*, ***kwargs*)

class `fluxscoreboard.forms._fields.`**BootstrapWidget**(*input_type=None*, *group_before=None*, *group_after=None*, *default_classes=None*)

`fluxscoreboard.forms._fields.`**team_size_field**()

class `fluxscoreboard.forms._fields.`**RecaptchaWidget**
RecaptchaValidator widget that displays HTML depending on security status.

class `fluxscoreboard.forms._fields.`**RecaptchaField**(*label=u''*, *validators=None*, *public_key=None*, *private_key=None*, *secure=False*, *http_proxy=None*, ***kwargs*)
Handles captcha field display and validation via reCaptcha

class `fluxscoreboard.forms._fields.`**TZDateTimeField**(*label=None*, *validators=None*, *format=u'%Y-%m-%d %H:%M:%S'*, ***kwargs*)

3.1.18 fluxscoreboard.util

Utility Functions

fluxscoreboard.util.**display_design**(*request*)

A function that returns `True` or `False` depending on whether the actual design should be displayed (`True`) or just a default one (`False`).

This is used to not disclose the real design until the CTF has started. The following logic is implemented:

- The admin backend has the default design (i.e. `False`).
- If it is a test-login from the backend, show the design.
- If the CTF has started, the real design is loaded for the frontpage (i.e. `True`).
- If no route is matched, the default design is loaded (`False`).
- Otherwise we fall back to not show the design (`False`).

The conditions are processed in order, i.e. the first match is returned.

fluxscoreboard.util.**is_admin_path**(*request*)

fluxscoreboard.util.**now**()

Return the current timestamp localized to UTC.

fluxscoreboard.util.**random_token**(*length=64*)

Generate a random token hex-string of `length` characters. Due to it being encoded hex, its entropy is only half, so if you require a 256 bit entropy string, passing in `64` as length will yield exactly `64 / 2 * 8 = 256` bits entropy.

fluxscoreboard.util.**nl2br**(*text*)

Translate newlines into HTML `<br />` tags.

Usage:

```
<% from fluxscoreboard.util import nl2br %>
${some_string | nl2br}
```

fluxscoreboard.util.**random_str**(*len_*, *choice='abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'*)

fluxscoreboard.util.**tz_str**(*timestamp*, *timezone=None*)

Create a localized timestring in a local `timezone` from the timezone-aware `datetime.datetime` object `timestamp`.

class fluxscoreboard.util.**not_logged_in**(*msg=None*)

Decorator for a view that should only be visible to users that are not logged in. They will be redirected to the frontpage and a message will be shown, that can be specified, but also has a sensible default.

Usage:

```
@view_config(...)
@not_logged_in()
def some_view(request):
    pass
```

3.1.19 fluxscoreboard.routes

Route Definition

All routes of the application in a list. Each item consists of the route name and its path in a tuple: `(route_name, path)`. These are then mapped to a view via the `pyramid.view.view_config` decorator. See *Routing and Views* for an explanation of how to create routes and work with them.

3.1.20 fluxscoreboard.install

Installation Utilites

Installation module that provides the application with mechanisms for installing and uninstalling the application. Also useful for testing.

`fluxscoreboard.install.`**`install`**(*settings*, *cfg_file*, *test_data=False*)
Installs the application. Only required to be called once per installation.

`fluxscoreboard.install.`**`uninstall`**(*settings*)
Remove those parts created by install

CHAPTER 4

ToDos

Fancy making a pull request?

4.1 ToDo

This is a list of things that still need to be done in this documentation:

Todo

Document.

(The *original entry* is located in /var/build/user_builds/fluxscoreboard/checkouts/stable/fluxscoreboard/forms/__init__.py:docstring of fluxscoreboard.forms.CSRFForm, line 1.)

Todo

Document

(The *original entry* is located in /var/build/user_builds/fluxscoreboard/checkouts/stable/fluxscoreboard/forms/_fields.py:docstring of fluxscoreboard.forms._fields.ButtonWidget, line 1.)

Todo

There is currently an ugly display of buttons on the Action listings as they don't have a 100% width. However setting it to 100% causes Overflow of larger text. Both is not nice :(

(The *original entry* is located in /var/build/user_builds/fluxscoreboard/checkouts/stable/docs/develop/forms.rst, line 129.)

Todo

Write chapter on how to unittest with py.test

(The *original entry* is located in /var/build/user_builds/fluxscoreboard/checkouts/stable/docs/develop/tests.rst, line 4.)

Todo

Configure gunicorn with proper logging.

(The *original entry* is located in /var/build/user_builds/fluxscoreboard/checkouts/stable/docs/user/getting_started.rst, line 95.)

Todo

Add some more details here.

(The *original entry* is located in /var/build/user_builds/fluxscoreboard/checkouts/stable/docs/user/getting_started.rst, line 139.)

CHAPTER 5

Indices and tables

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

f

Symbols

A

B

C

D

O

P

R

S

T

U

V