EDPC Mentoring Database Documentation

Release 0.1.0

EDPC

Jul 20, 2017

Contents

CHAPTER 1

Getting started

This section is intended for developers new to the database who want to get up and running with a test instance quickly.

Before you start

You should have done the following things before starting to work on the database:

1. Get a machine running Unix-like OS or Mac OS X. (It makes life easier in the long run when dealing with Python.)
2. Install Python and virtualenv.
3. Work through the Django tutorial.

Running a test instance with fake data

Clone the repository and `cd` to it. The remaining instructions assume the repository root is the current working directory.

Create a virtualenv:

```
$ virtualenv -p $(which python2.7) env
$ echo "env" >>.git/info/exclude # To avoid any accidental commits
$ . env/bin/activate
```

Install the requirements:

```
$ pip install -r requirements.txt -r dev-requirements.txt
```

Note: The `requirements.txt` file lists the Python packages required for deployment. The `dev-requirements.txt` file lists the Python packages used only by developers. For example, the "ipython" package, which is used to improve the `shell_plus` command, is in the `dev-requirements.txt` file since it is not required when deploying the database.

Perform the initial database migration and populate the database with some fake data:

```
$ ./edpcmentoring/manage.py migrate
$ ./edpcmentoring/manage.py runscript loadtestfixtures
```

Start a local development server:

```
$ ./edpcmentoring/manage.py runserver_plus
```

Open http://127.0.0.1:8000 in your web browser. The admin interface is at http://127.0.0.1:8000/admin.

Installing on UIS MWS3 (https://panel.mws3.csx.cam.ac.uk/)

You will need:

- To register and create an MWS3 server (see link above)
- The root password for MySQL server (available once you MWS3 has been setup)

ssh onto your MWS3 server (via putty, or a linux/unix console), then:

```
$ cd /var/www/default/admindir
$ git clone https://github.com/cuedpc/edpcmentoring.git
$ python edpcmentoring/deploy/setup_mws.py
```

You will be asked for

- The server's MySQL root password.
- A short name which will be prefixed by '**pc_**' and used as the database name
- A password the Django applicatoin will use to connect to the MySQL server
- A unique passphrase / secret key for your application
- Whether you wish to load the sample test data into the application

Once complete you should be able to visit the mws3's host name and, if you have loaded the test data, login as test000X.

Note: The local config is held in edpcmentoring/edpcmentoring/edpcmentoring/settings_local.py, and DEBUG is set to True - not advised for production systems.

Notes on the test database

- There is one superuser: `test0001`.
- The users `test0001` and `test0002` can log into the admin interface.
- Users `test0001` to `test0099` are members of CUED but not all are *active*.
- Users `test0100` to `test0199` exist in the database but are not CUED members.

Development

This section contains some important information if you're thinking of developing a feature for the database.

Tests

The test suite for the mentoring database is run via the `tox` test-runner. If you're intending to develop a feature for the database, it is important that you write tests. By default, `tox` will run tests using whichever Python version correspond to the installed `python` and `python3` binaries.

Install `tox` via pip:

```
$ pip install --user tox
```

You can now run the tests via the `tox` command:

```
$ tox
```

Any positional arguments are passed to the underlying invocation of `manage.py test` and so you can specify a particular application to test by giving it's directory. For example:

```
$ tox edpcmentoring/cuedmembers
```

Code coverage

The tests are run under the `coverage` code-coverage utility and files which do not have 100% test coverage are printed out after the tests are run. Additionally, a HTML report is generated in `htmlcov/` which is useful for determining which lines are untested.

Although 100% code coverage is probably infeasible in general, we aim for as close as possible in the database. Pull requests which increase test code coverage are welcome.

CHAPTER 2

Mentoring relationships

The *mentoring* application provides the core of the mentorship management support.

Relationships

class `mentoring.models.`**`Relationship`**`(`**args*, ***kwargs*`)`

The mentoring database is structured around the concept of a "mentoring relationship" between two users. One user will be the mentor and one will be the mentee.

Note: This model is not yet completely set in stone. In particular, it's unclear whether the ending of the relationship should be recorded in the model or in a related model.

`mentor`

The user who is the mentor in this relationship.

`mentee`

The user who is the mentee in this relationship.

`is_active`

True iff this relationship is currently active.

class `mentoring.models.`**`RelationshipManager`**

`active``()`

A queryset of active relationships.

`inactive``()`

A queryset of inactive relationships.

`mentees_for_user``(`*user*`)`

A queryset returning all users who are the passed user's mentees. Only active relationships are considered.

`mentors_for_user`(*user*)
A queryset returning all users who are the passed user's mentors. Only active relationships are considered.

Meetings

class `mentoring.models.`**`Meeting`**(**args*, ***kwargs*)
Meetings are recorded for a particular relationship.

Note: The in-database duration is likely to have a ludicrous resolution (maybe microsecond) but using a DurationField in this model has the advantage that it is exposed as a standard Python timedelta object.

Note: It is likely that a "happened" field will need to be added to this model at some point to allow users to specify that a meeting was mistakenly recorded.

`relationship`
The *`Relationship`* this meeting is associated with.

`held_on`
The date this meeting was held.

`approximate_duration`
The approximate duration of the meeting.

class `mentoring.forms.`**`ReportMentorMeetingForm`**(**args*, ***kwargs*)
A form which allows a user to record a meeting where they were a mentor. Much like a `ModelForm` this object provides a single *`save()`* method which can be used to save the cleaned data to the database.

Parameters **`mentor`** (*`django.contrib.auth.models.User`*) – A **required** argument which specifies the user who is the mentor.

`save`(**args*, ***kwargs*)
Save this meeting to the DB.

CHAPTER 3

Matchmaking

The *matching* application adds support for so-called "matchmaking" of mentors and mentees. This includes invitations to form matchmaking relationships as represented by the *mentoring.models.Relationship* model.

Models

class `matching.models.`**`Invitation`**(**args*, ***kwargs*)

An invitation to form a mentoring relationship.

Invites are the mechanism where mentor/mentee relationships are created. The *clean()* method of this model is overridden to allow invites to be created by anyone but, in that case, they need to be the mentor or mentee of the invite. Users with the "add_invitation" permission can invite any two users to form a mentor/mentee relationship.

Each invitation records who the mentor and mentee are to be and the user who created the invite. (This is useful to determine which of the mentors and mentees should actually be notified.) The creation date is also stored to allow for some form of automatic expiry.

An "active" invite is one which is not expired and is still awaiting a response from the mentor or mentee. An invitation which is declined by either mentor or mentee should be marked as inactive even if the other party has not responded.

The *deactivated_on* date records when this invite became inactive either through being declined by one party, accepted by both or manually deactivated.

Should the invite result in a new relationship, this is recorded in the *created_relationship* field.

RESPONSES = (('A', 'Accept'), ('D', 'Decline'))

The possible responses to an invitation.

clean()

Extra validation for invitations.

Creating invitations when the creator is not the mentor or mentee requires that the creator have the "add_invitation" permission.

created_by
The User who created this invitation

created_on = None
The date this invitation was created

created_relationship
If this invite lead to a mentoring relationship, it is recorded here

deactivate()
Deactivate this invite without creating a relationship.

Does nothing if the invite is already deactivated.

deactivated_on = None
If inactive, when did this invite become so

is_accepted()
Returns *True* iff both the mentee and mentor have accepted the invite.

is_active()
Returns *True* iff the invite is active (i.e. the "deactivated_on" date is blank).

mentee
The proposed mentee for this relationship

mentee_response = None
The response from the mentee to the invite

mentor
The proposed mentor for this relationship

mentor_response = None
The response from the mentor to the invite

respond(*user*, *accepted*)
Set the response of the specified user. If the user is neither the mentor or mentee then a PermissionDenied exception is raised.

class matching.models.**InvitationManager**
Model manager for *Invitation* model.

active()
Return a query set giving only the active invitations.

class matching.models.**Preferences**(**args*, ***kwargs*)
Records the mentorship opinions of a User.

matching.models.**invitation_create_relationships**(*instance*, ***_*)
A pre-save hook for Invitation instances which creates a mentoring relationship if:

- The invitation is accepted
- The invitation is active
- There is no current relationship

Forms

class `matching.forms.`**`InvitationResponseForm`**(*data=None*, *files=None*, *auto_id=u'id_%s'*, *prefix=None*, *initial=None*, *error_class=<class 'django.forms.utils.ErrorList'>*, *label_suffix=None*, *empty_permitted=False*, *instance=None*)

A form for responding to an invitation.

Validates that the user responding to the invitation is one of the mentor or the mentee and that the invitation is active.

Example:

```
invitation = # ... Retrieve Invitation instance
f = InvitationResponseForm(
    data={'user': request.user.id, 'response': Invitation.ACCEPT},
    instance=invitation
)
f.save()
```

`response` **= None**
The response. One of `Invitation.ACCEPT` or `Invitation.DECLINE`

`user` **= None**
The database primary key for the user responding to the invitation.

CHAPTER 4

The CUED People database

The *cuedmembers* application contains models and logic to maintain a shadow copy of a list of CUED members. It augments the builtin `django.contrib.auth.User` model with information about a member of CUED. This includes:

- The full list of "first names";
- Their research group and division (if any); and
- Whether they are a current member of CUED.

Members

class `cuedmembers.models.`**`Member`**`(`**args*, ***kwargs*`)`

An extension of the standard Django User to indicate that a particular user is a member of the Department.

There is a one-to-one mapping of Users to Members however not every User is necessarily a Member.

Note: While there is nothing stopping you adding a foreign-key to a Member in models, its better to add a foreign-key to a User. That way you app's models are decoupled from relying on the CUED membership database and may be of wider user.

The "Surname" and "Preferred name" fields from the Department are mapped through to the associated User's `last_name` and `first_name`. The "First names" provided by the Department are stored in this model.

An "active" member is currently present at CUED.

Information in this model is expected to be provided by the Department. See http://www-itsd.eng.cam.ac.uk/datadownloads/support/div_people.html for some discussion of what the fields mean.

Note that is_active is the primary means by which one should judge if a Member is currently a member of the Department.

This model does not include role/course, host/supervisor, room number or phone number. The "arrived" flag is folded into the is_active field.

user
The `django.contrib.auth.models.User` associated with this Member.

first_names
A string containing the first names for the member supplied by the Department. Although it is tempting to use a space as a separator for these, that way danger lies!

research_group
The *`ResearchGroup`* which this member is a part of.

is_active
Members are not usually deleted from the database. Instead they become "active" or "inactive". This is to allow for the same person to be considered *as* the same person if they leave CUED and then subsequently return.

crsid
This member's CRSid. The CRSid is the username of the associated user.

This property merely returns the `username` of the associated User.

class `cuedmembers.models.`**MemberManager**
A specialised Manager for *`Member`* instances.

active()
A query-set of active users.

inactive()
A query-set of inactive users.

update_or_create_by_crsid(*crsid*, *defaults=None*)
Retrieve or create a new member from a crsid. If a corresponding user does not exist, it is created. The newly created user has set_unusable_password() called on it and is added to the database.

The first_name, last_name and email entries in defaults are set on the corresponding user and any other values are set on the member.

See the update_or_create() documentation for discussion of the defaults parameter.

`cuedmembers.`**get_member_group**()
CUED members who are active are a member of a group. Membership of this group is automatic for those members who have `is_active` set to `True` when imported from CSV via `csv.read_members_from_csv()`.

Note: The group membership is *not* automatically updated when `save()` is called on the *`models.Member`* model. This is because only advanced users who know what they're doing should be fiddling with the database model directly!

By default this group is called "CUED Members" but the name may be overridden by setting the `CUED_MEMBERS_GROUP` setting.

Departmental structure

The Department is structure into Divisions which comprise separate Research Groups. The `cuedmembers` app ships with a fixture which is automatically loaded into the database at migrate-time which contains the current Divisions and Research Groups.

class `cuedmembers.models.`**Division**(**args*, ***kwargs*)
A division within CUED. The primary key for a division is actually its letter, A-F.

letter
The Divisional ID letter, A-F.

name
A human-readable name for the Division.

class `cuedmembers.models.`**ResearchGroup**(**args*, ***kwargs*)
A research group in CUED.

division
The `Division` which this research group is a part of.

name
A human-readable name for the Research Group.

Settings

CUED_MEMBERS_GROUP String giving the name of the group created or returned by `get_member_group()`.

Management commands

Synchronising membership data via CSV files

The `importcuedmembers` management command is used to synchronise the membership database with an authoritative source.

The Department provide CSV dumps of CUED membership. See http://www-itsd.eng.cam.ac.uk/datadownloads/support/div_people.html for more details. This command allows the ingestion of a CSV file in the format outlined at that page into the database.

Members listed in the CSV file are created if they don't exist. Their personal details, such as first name, surname, etc. are updated from the CSV. A previously active member who does not appear in the CSV file is marked inactive. Similarly, a previously inactive member who appears in the CSV file is marked active.

By default, an email address of `<crsid>@cam.ac.uk` is used for each member. This can be configured through the `--email-domain` argument.

This command can take either a path to a CSV file on the local system or a http or https URL to a CSV file located on a remote server.

CHAPTER 5

Notifying and reminding users

The *autonag* application takes care of notifying users of events they should be aware of and reminding them of tasks they have yet to complete.

Python Module Index

Index