TREC Evaluator Application

Description: The purpose of this application is to provide a tool that lets researchers evaluate how good their search system is and how it compares to other systems (already submitted).

Specifications:

Researchers need to be able to:

- create an account and profile
- edit, update and maintain their profile
- submit / upload runs to be evaluated
- obtain performance data for each run submitted to a particular track
- obtain comparison data against all other runs submitted to a particular track

Visitors need to be able to:

- view the list of tracks
- view the table of runs for a particular track
- view and inspect the descriptions for each run
- sort and rank runs based on a particular measure
- filter runs by particular type or description keyword
- view a graph of the results/runs submitted over time

Administrators need to be able to:

- create tracks
    - upload initial set of runs associated with the track

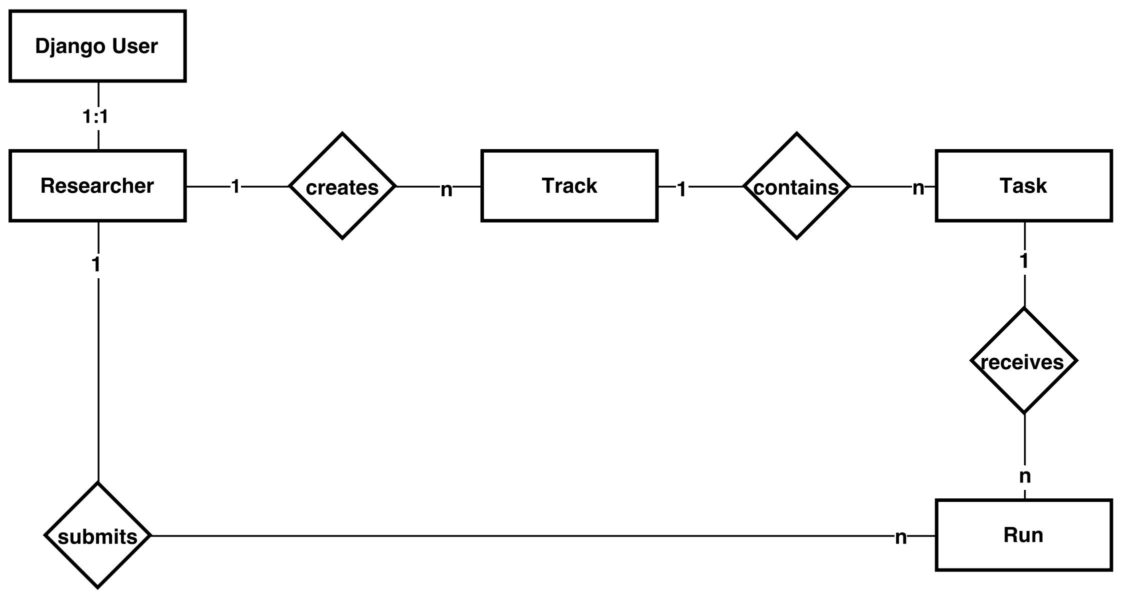
The application needs to engage the service of an external programme, called trec_eval: http://trec.nist.gov/trec_eval/

A TREC track/task consists of a title, url to track, year, description of the track, task type, a judgements file (qrels), a list of official measures, and notes.

Example tracks:
http://trec.nist.gov/data/robust.html
http://trec.nist.gov/data/webmain.html

One track has many runs associated with it. A run consists of a file in a TREC format (see example file), a description of the run, a name, and an associated type (automatic, manual), a query type (title, title+description, description, all, other), and feedback type (none, pseudo, relevance, other).

Each run is evaluated the list of performance measures associated with the run are also stored as results. The trec_eval program provides the results given the qrel file and the run file. We want to keep a record of: map, P10 and P20
A user has a profile (url, affiliation) and they can submit a number of runs to a track.



Researcher (user, profile_picture, website, display_name, organization)

username, password, email, are store in the django user object.

Track (title, track_url, description, genre)

Task (track, title, task_url, description, year, judgement_file)

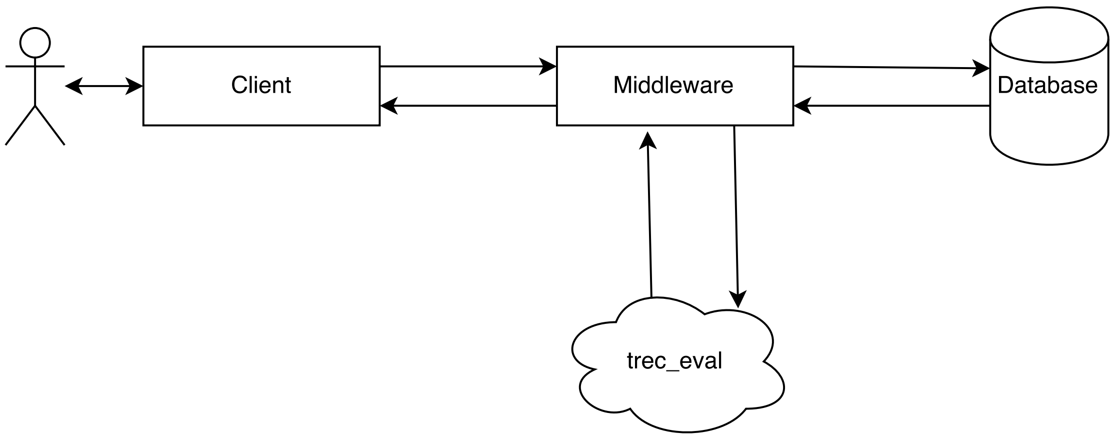
Run (researcher, task, name, description, result_file, run_type, query_type, feedback_type, map, p10, p20)

What if later on we want to score lots of different measures? How would we cater for this change?

Ignore the <creates> entity between Researcher and Track (not needed now).

Enumerated Types:
Run_type (automatic, manual), a query_type (title, title+description, description, all, other), and feedback_type (none, pseudo, relevance, other).

We can also set genre to be an enumerated type as well. See github for example data.



trec_eval can be considered an external service as we are invoking code that is external to our application.

Systematic Review Application

Description: The purpose of this application is to enable researchers to perform a systematic review using PubMed. A systematic review consists of a number of stages, which need to be recorded: (1) query construction, (2) Abstract Pool Evaluation and (3) Document Pool Evaluation.

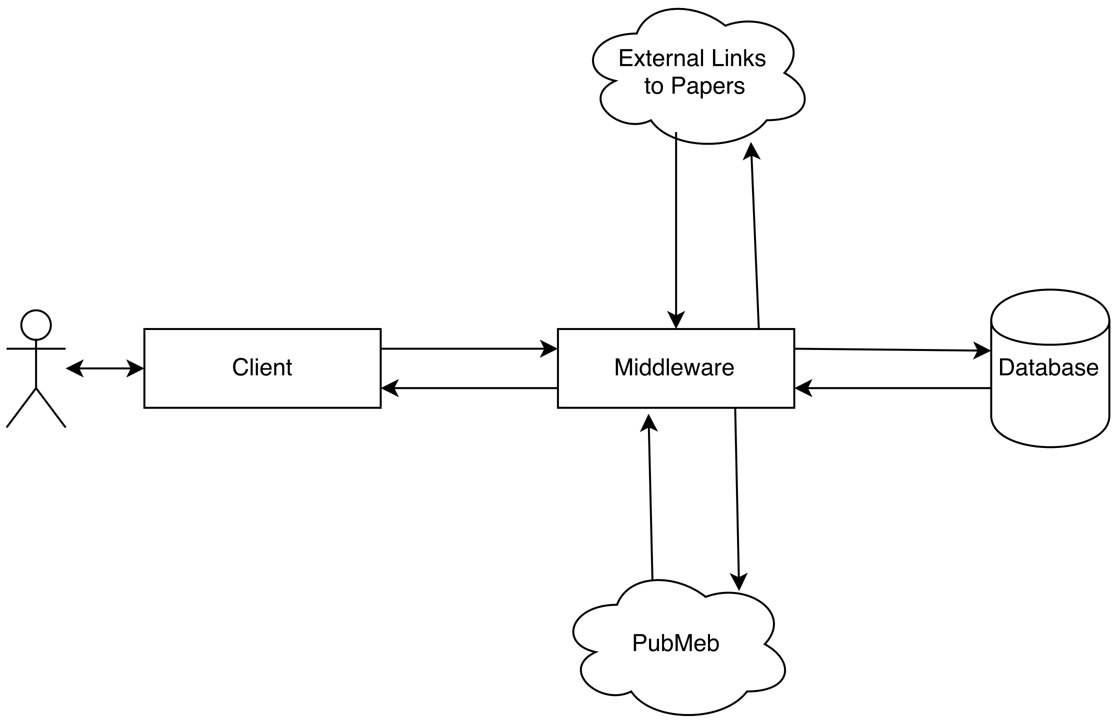
Specifications:
Researchers need to be able to:

- create an account and profile
- edit, update and maintain their profile
- create/edit/delete systematic review topics
- construct boolean based queries
    - for experts (a standard query box)
    - for novices (a structured / visual querying interface)
- view the total number of hits that a query would return
    - ~~(and how much overlap with the abstract pool)~~
- save/edit/delete queries
    - saved query results are added to the abstract pool
- add/remove results from a query to/from the abstract pool
- update the abstract pool based on the set of queries for that systematic review
    - this needs to check if documents that have been assessed will be removed, and warn the user.
- View the abstract pool of documents
- Judge abstracts in the abstract pool
    - If considered relevant, the document is added to the document pool
- View the document pool
- View the document in the document pool (perhaps through an external link)

- Judge the document in the document pool and mark as relevant or not-relevant
- View final set of relevant documents

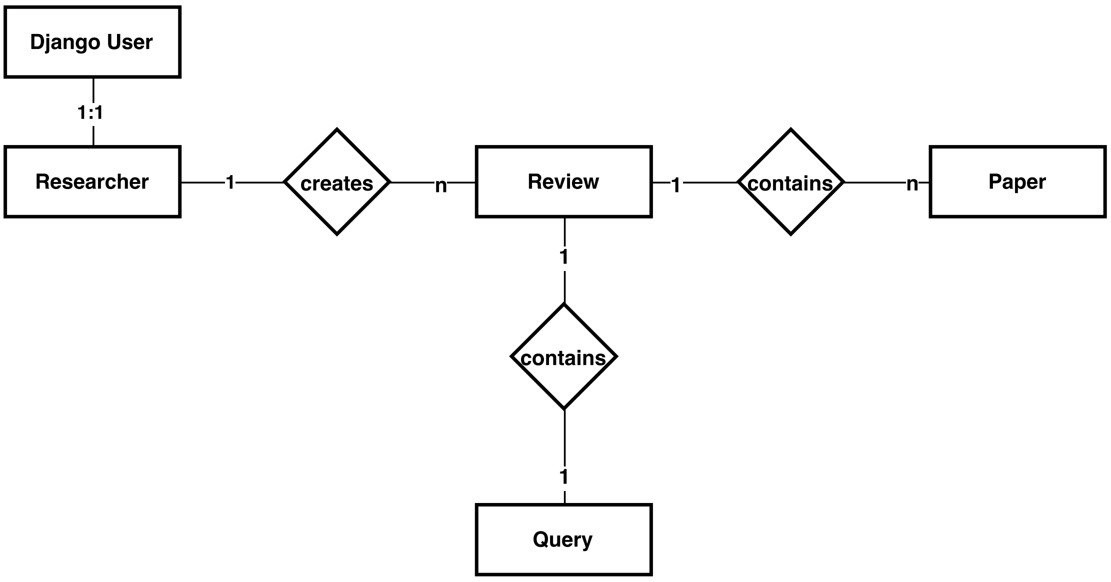
One researcher can create many systematic reviews, one systematic review contains a number of queries, each query retrieves a number of documents, which are added to the abstract pool. An abstract can be judged relevant and is added to the document pool. A document in the document pool can be judged relevant and added to the final pool.

The API to use is provided by E-Utilities which gives access to the PubMed api: http://www.ncbi.nlm.nih.gov/home/api.shtml



PubMeb is an external API (must have), and the External Links (may or may not be necessary) denotes that we could be contacting other sites for information about the a paper).

Actually the external links cloud should connect to the client – not the middle ware.



Researcher (user)

username, password, email stored in django user table.. so research is not needed

Review (user, title, description, date_started, query_string, pool_size, abstracts_judged, document_judged)
Paper(review, title, authors, abstract, publish_date, paper_url, abstract_relevance (T/F), document_relevance (T/F), notes)

A **Query** class could also be created. Here I have it as a one to one, so I subsume it within the **Review** class, but it could be represented as a one to many, i.e. a review has many queries (or many query parts) that are issued.

In Review, I have included a couple of attributes regarding the size and number of pool and abstracts/documents judged. We could compute them each time, but this would require numerous calls to the database.

Example Queries:

Attention Deficit Disorder

```
1 Attention Deficit Disorder with Hyperactivity/
2 adhd
3 addh
4 adhs
5 hyperactiv$
6 hyperkin$
7 attention deficit$
8 brain dysfunction
9 or/1-8
10 Child/
```

```
11 Adolescent/
12 child$ or boy$ or girl$ or schoolchild$ or adolescen$ or
teen$ or "young person$" or "young people$" or youth$
13 or/10-12
14 acupuncture therapy/or acupuncture, ear/or
electroacupuncture/
15 accupunct$
16 or/14-15
17 9 and 13 and 16
```

Back Pain

1. randomized controlled trial
2. controlled clinical trial
3. randomized
4. placebo
5. drug therapy
6. randomly
7. trial
8. groups
9. or/1-8
10. (animals not (humans and animals))
11. 9 not 10
12. dorsalgia
13. exp Back Pain/
14. backache
15. (lumbar adj pain)
16. coccyx
17. coccydynia
18. sciatica
19. sciatic neuropathy/
20. spondylosis
21. lumbago
22. exp low back pain/
23. spondylosis.mp. or exp Spondylosis/
24. back pain.mp.
25. or/12-24
26. 11 and 25
27. Ultrasonics/
28. ultrasonic*
29. ultrasound
30. or/27-29
31. 26 and 30

Federated Health Search Application

Description: The purpose of this application is to help people find out about particular conditions and to save the information that they find into different folders. The application lets people search across two different medical sites (medline and healthfinder) and the general web (bing). People using the application would like to self-diagnose, i.e. given some symptoms find out what are the likely conditions. They would also like to find out information about particular conditions, treatments and medicines.

Specifications:
Searchers need to be able to:

- create an account and profile
- edit, update and maintain their profile
- create a category (where they can save pages that they found)
- save pages that they find during the process
- search for conditions, treatments, medicines.
- share a list of links (via a category)
    - links and categories should be private, unless shared.

The search interface should provide:

- a mash up of results from the three search verticals
- tabs to enable access to the individual search verticals
- results should be annotated in terms of source (which vertical)
- a list of user defined categories should be available to select and browse
- an easy mechanism to save results to categories
- inline query suggestions to help uses with tricky words and spelling

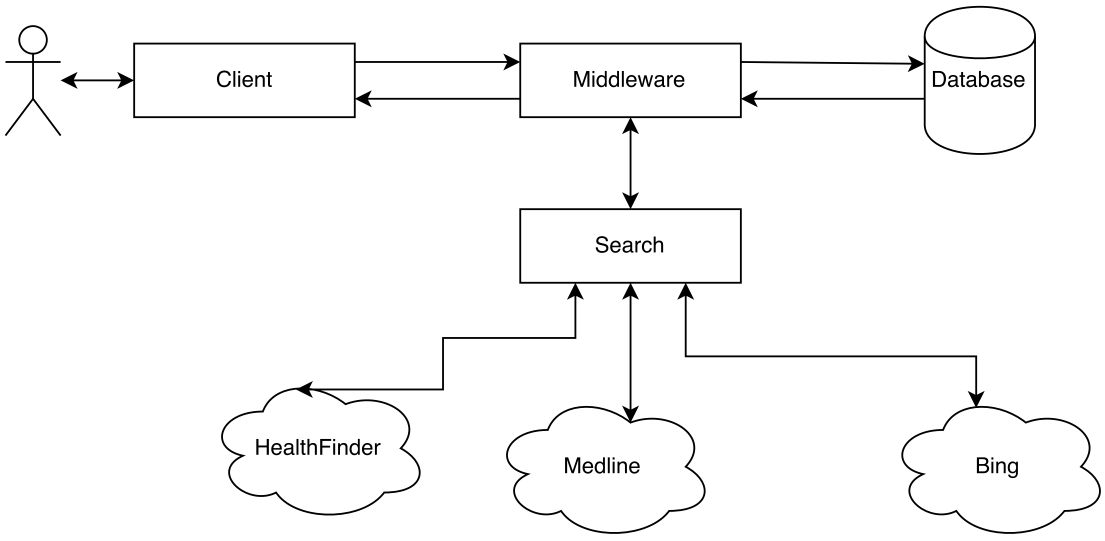
When results are saved, the page should be fetched and stored. The readability of the page should be calculated, using TextStat's Flesch Reading Ease Score. The polarity and subjectivity of the text should also be calculated, using TextBlob. When looking at saved pages, the searcher should be able to see the title of the page, the summary, the source, url, the reading score, polarity score and subjectivity score.

The API's for the federated search are:

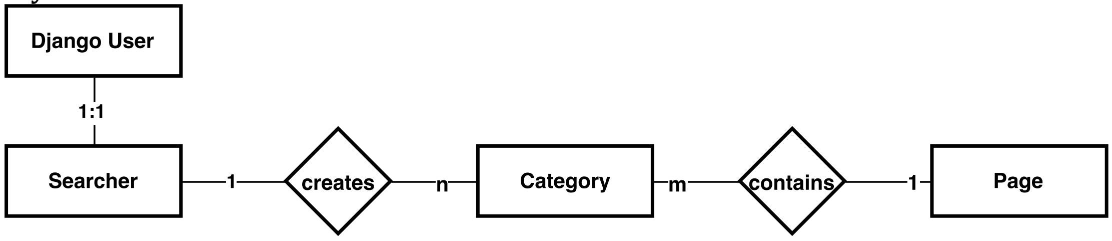
- http://healthfinder.gov/developer/How_to_Use.aspx
- https://www.nlm.nih.gov/medlineplus/webservices.html
- https://datamarket.azure.com/dataset/bing/search

Additional resources:

- https://pypi.python.org/pypi/textstat/0.1.4
- https://pypi.python.org/pypi/textblob/0.11.0



HealthFinder, Medline and Bing are all external APIs. To provide some clarity I have split out the Search component that will interface with these APIs. And then my middle ware will use this service.



Does this ERD make sense?

Searcher(username, password, email)
Category(user, name)
Page (category, title, summary, url, flesch_score, polarity_score, subjectivity_score)

Zombie Survival: A Game of Search

Description: The purpose of this web application is to provide the frontend interface for a game of search and survival. The web application needs to interface with a series of classes that create the world in which the player needs to survive.

The game is as follows:

A player starts on day one of the zombie apocalypse. Their goal is to survive as long as possible (i.e. number of days).

Their party size is 1. They have 3 units of food and 2 units of ammunition.

They enter a street. A street contains a number of houses. Each house has a number of rooms.

A room may contain other survivors, zombies, food and/or ammunition.

The player has a number of moves.

On the street:

(1) a player can choose which house they enter.

In a house:

(1) enter room
(2) move to next room
(3) leave house

In a room:

(1) search room
(2) leave room
(3) if zombies are present, they can fight or run.
  a. if they run, they leave the house and return to the street
  b. if they fight, they might lose the fight (lose party members), or win, in which case they claim whatever is in the room (survivors, food, ammo)
  c. the number of zombies killed is counted

The game mechanic for win/loss is based on the number of zombies, the amount of ammo and the size of the party.

During a day, the player has X units of time. Each action costs time. The day ends when they run out of time. At the end of the day, the game mechanic, decreases the amount of food depending on the size of the party. If there is not enough food, then party members might leave. Party size and food is updated, then they start the next day.

The web application needs to provide the following functionality.

Specifications:
Players need to be able to:

- create an account and profile
- view, edit, update and maintain their profile
- start a game and return back to a game in progress.
- Play the game according to the mechanics described above
- View the leaderboard

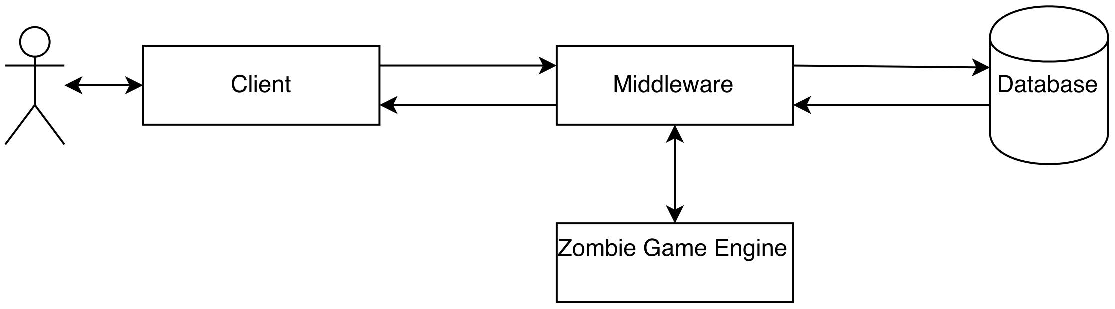
Visitors to the site:

- Need to be able to view the leaderboard
- Read the instructions of the game
- Be able to register

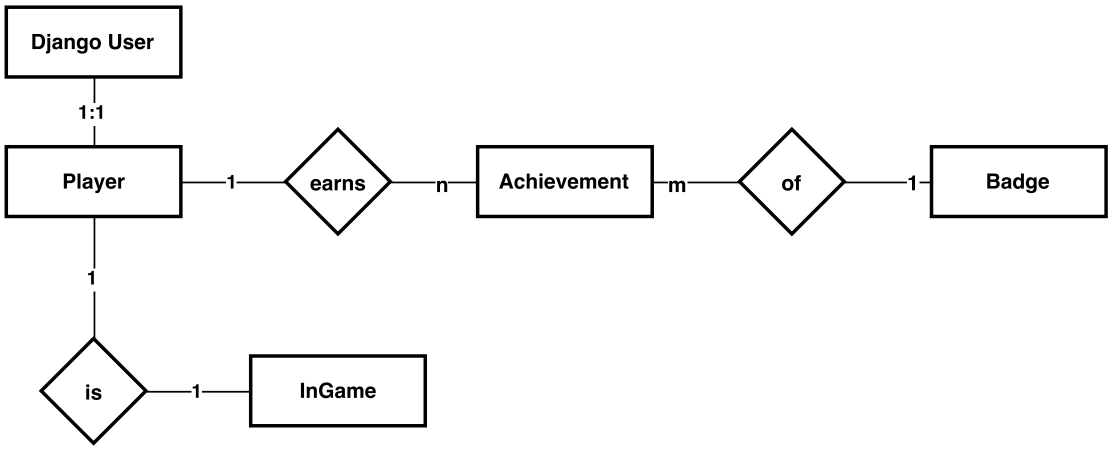
The profile contains the players handle/username and picture along with how many games they have played, their average and best number of days survived and zombie kills. The profile also lists the badges that the player has achieved.

Badges: there are 4 badges with different levels (bronze, silver, gold): (i) Survival: survived 5, 10 and 20 days. (ii) Killer: killed 10, 20 and 50 zombies in one game. (iii) Stamina: played 5, 10 and 20 games, (iv) Party: had a party size that contained 10, 20, 40 people.

Leaderboard: there are two leaderboards: (i) top 20 players based on days survived, (ii) top 20 players based on zombie kills.



As the application will use the Game Engine provide we have made this explicit on the high level system architecture diagram.



We assume that a Player can only play one game at one time. Statistics on pasts games are recorded in the Player model.

Since they can only be in one game at one time, we can store the game in the player model, too.

Player(user, profile_picture, games_played, most_days_survived, most_kills, most_people, current_game)

username, password, email, are stored in the django user model.

Current_game stored the current instantiated game (tip: use the python pickle package to persist the game object, and save it in the Player model).

How the **Badge** is represented depends a lot on the implementation. My suggestion is below, but other justifiable versions are acceptable, especially if they work in an elegant fashion.

Badge(name, description, criteria, badge_type, level, icon)

Where *bagde_type* (kills, people, days) and *criteria* is an integer specifying how many kills, people, days, etc have to be obtained in order to get the badge.

Icon is an image that represents the badge. The population script should ensure that the badges and their icons are all inserted into the database.

A class/handler is needed to determine if the *criteria* for a particular *badge_type* is met, or not.

Achievement (player, badge, date_awarded)