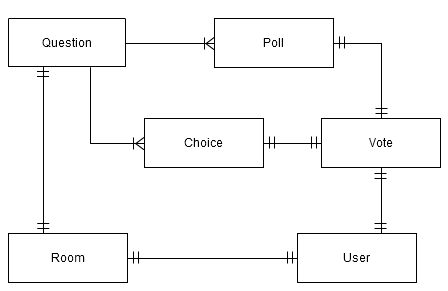
# Database Design

When designing any application, the first step is usually to define the way that data will be stored. This is most easily done on paper, using rough entity-relationship diagrams to map out the way that different database objects (entities) will interact. In Django, the database layout is known as the *models*.

Our group spent some time discussing the best way to design the database, with most of the effort spent deciding on the polling structure. The way that polls and votes are to be stored is the most complicated part of the application, as we decided that historical data would need to be kept for post-processing and analysis once the conference/debate is finished.

After some revisions (which are included in the appendix in photographic form), the final entity-relationship diagram was decided on as follows:



### Room

Central to the application is the **Room** entity. This represents a conference room/debate. The **Room** entity has an owner, which is represented internally in Django using the **User** model.

### Question

The room also has one (and only one) **Question** object. This represents the topic that the room is based around, and stores the question that members vote upon every time a period ends.

### Poll

The question then references many **Poll** objects. These are the “instantiations” of the question for a particular period. This allows each **Poll** to store the votes associated with it, whilst not being deleted once the period has ended.

### Choice

The question object also references many **Choice** objects. These represent the choices that are available for participants to vote on.

### Vote

Once a participant casts a vote, a new **Vote** object is created. This object references both the **Choice** and the **Poll** that was taking place at that particular moment. It also references the user that cast the vote.

## Justification

The above implementation was chosen over a number of alternatives. These alternatives include using an “archive” version of the poll, choice and vote objects in order to keep historical data.

The given ERD was decided upon as being the neatest (and simplest) way of solving the problem, and hinges on the fact that a **Vote** objects stores a reference to both the choice *and* the poll upon which it was cast.

# API Design

The messaging aspect of the application, in particular, relies heavily on Javascript. Without this, messages would not be delivered to the user until the browser page is refreshed. This was deemed early on to be wholly unacceptable.

Therefore, an easily-consumable API was required, to allow simple acquisition of data, and to also allow data to be posted asynchronously (in the case of sending a message or casting a vote, for example).

## JSON

The group decided to use JSON (Javascript Object Notation) to serialise the data when communicating between client and server. This is because it is such a simple, human-readable protocol. Also, it is extremely easy to serialise and de-serialise data using built-in Python and Javascript libraries.

The alternatives to representing the data in JSON, would be to use a format such as XML, which carries it's own advantages and disadvantages. For example, XML is less human-readable (and thus it is harder to debug an error in the application), though it is possible to give the data meaning, which is not an option with JSON – the application designer simply has to know which fields represent what. Fortunately, that is an advantage that we have with this project, so JSON offers a much easier format to work with.

## API Methods

To achieve full functionality on the front-end of the system using Javascript, the following methods were implemented in the API:

### rooms/get\_info

Returns relevant information pertaining to a particular room/debate. Includes the following data:

* number of members
* list of members
* messages (all or only unread)
* current room mode (conferencing or voting)
* time before next poll

### rooms/send\_message

Posts a message to a given room/debate. Returns the following:

* list of unread messages for the sending user (**including** the message posted)

### rooms/reset

Resets a given room/debate, deleting all current poll data and resetting the countdown to the original value (length of the period). Only the creator of the room or an administrator is able to perform this action.

### rooms/end

Ends a given room/debate, thus removing it from the main list of debates. Again, only the creator of the room or an administrator is able to perform this action.

### polling/get\_info

Retrieves the information for the poll that is currently in use for a given room. Returns the following information:

* number of votes
* whether the current user has voted on the current poll or not
* poll results so far (the number of votes for each possible choice)

### polling/cast\_vote

Casts a vote for a given choice in the current poll for a given room.

## jQuery

In order to implement the complex front-end user interactions, such as asynchronous message receipt and vote casting, a lot of javascript was required. Therefore, it made sense to make use of a library in order to make the heavy-lifting much simpler. The library that seemed best suited to this task was **jQuery**, which is free and open source (and licensed under either the MIT license or the GPL).

There are alternatives to jQuery, such as mooTools, Prototype, and others. Rob has experience with jQuery however, and there are no obvious disadvantages to using it over one of the alternatives. In fact, it is the most popular Javascript framework on the internet[[1]](#footnote-2).

# Django Design

When designing a Django project, the generally accepted method for development is to split the functionality into separate, loosely-coupled *applications*, each of which could feasibly be installed into another Django project, and function correctly (given any dependencies that the app may have are satisfied). For our system, we divided the project up in the following way:

## Rooms Application

The *rooms* application is where the core functionality of the system lies. Rooms contains the Room, Message and Membership models, and handles all aspects of the low-level chat-based system. The *rooms* app has the following views:

* conference\_room
  + Displays the initial HTML for the conference room/debate screen, at which point the Javascript takes over to provide the end-user experience (as the page never refreshes)
* leave
  + Allows a user to leave a conference room
  + Removes user from the “current members” of a room instantly
* create\_room
  + Allows a user to create a new conference room/debate
  + Presents the HTML form for creating a new room if the request method is GET
  + Parses the form data and returns errors/success redirect if the request method is POST

The *rooms* application depends on the *polling* application.

## Polling Application

The *polling* application handles all aspects of the polling system, including casting votes and getting the results for a given room's poll. Therefore, the *polling* application depends on the *rooms* application. This app contains the *Question*, *Poll*, *Choice* and *Vote* models.

The *polling* application has no views in it's app folder. Instead, all functionality is exposed via. the API, which is explained in the next section.

## API Application

The project uses a third application to wrap up all the functionality of the JSON API into one neat code structure. This application is laid out slightly differently to the other two, with a folder for the views instead of a single file. This is because the API application is made up **solely** of views, so separation made more sense.

There are two important files inside the views folder – one for rooms and one for polling. The third file, **\_\_init\_\_.py**, is simply an empty file that tells Python that the directory is to be treated as a python module, instead of just a normal folder. This allows the rest of the project to access files inside the views folder.

Also, the views themselves in the API application are written differently. All the views extend from a common class, **APIView**. There is also a class called **APIAuthView**, which acts as a convenience class that does not allow the API view to be used unless the user is logged in.

The API methods for both of the main apps are listed in the API Design section above. This section focusses however, on the way that the applications are laid out.

The APIView class has two methods that are available for overriding: **get()** and **post()**. These are called when the respective request method is made (a GET request or a POST request). The parent class also has a method named \_\_call\_\_. This is used in Python to execute when the object is “called”. This means that an instantiation of the APIView class acts just like a function, which is how a view should act.

The \_\_call\_\_ function then calls the correct request method, and then returns the serialised JSON data to the browser, with the correct mimetype. This also allows the \_\_call\_\_ method to catch any exceptions that may occur in the processing, and return them as an error string in the JSON data, leaving the success variable as False, which signifies to the client that something went wrong – without causing an error 500 (internal server error). This also means that errors that occur deep within the backend system can be propogated to the user, instead of using complex error codes (or ambiguous “general” errors).

1. http://trends.builtwith.com/javascript/JQuery [↑](#footnote-ref-2)