|  |
| --- |
| University of Nottingham |
| **Democratic Conferencing Tool** |
| **G52GRP Final Group Report** |
|  |
| gp09-sdb |
| May 2010 |

**Group:** gp09-sdb

**Supervisor:** Dr. Steven Benford

**Group Members:** Robert Golding (rjg08u),

William (Billy) Redrup (wrr08u)**,**

Tammie Seo (tls08u)**,**

Christopher (Kit) Lensvelt (cxl08u)**,**

Henry James (hej08u)**,**

Zhongda (Carl) Zhu (zxz09u)

Table of Contents

[Table of Contents 2](#_Toc257201659)

[1 Executive Summary 4](#_Toc257201660)

[2 Introduction 4](#_Toc257201661)

[3 Research 4](#_Toc257201662)

[3.1 Summary of Completed Research 4](#_Toc257201663)

[4 Functional Specification 6](#_Toc257201664)

[4.1 Overview 6](#_Toc257201665)

[4.2 User Stories/Scenarios 6](#_Toc257201666)

[4.2.1 Scenario 1: Jimmy 6](#_Toc257201667)

[4.3 Flowchart 7](#_Toc257201668)

[4.4 Screen-by-Screen 7](#_Toc257201669)

[4.4.1 Home Page 7](#_Toc257201670)

[4.4.2 Login 7](#_Toc257201671)

[4.4.3 Register 7](#_Toc257201672)

[4.4.4 Debates List 8](#_Toc257201673)

[4.4.5 Debate Screen 8](#_Toc257201674)

[4.4.6 Voting 8](#_Toc257201675)

[4.4.7 Creating a new Debate 8](#_Toc257201676)

[4.4.8 Users List 8](#_Toc257201677)

[4.4.9 User Profile 9](#_Toc257201678)

[4.4.10 Search 9](#_Toc257201679)

[4.4.11 FAQ 9](#_Toc257201680)

[4.5 Requirements 9](#_Toc257201681)

[4.5.1 Functional Requirements 9](#_Toc257201682)

[4.5.2 Non-Functional Requirements 10](#_Toc257201683)

[5 User-Interface Specification 11](#_Toc257201684)

[6 Implementation 12](#_Toc257201685)

[6.1 Database Design 12](#_Toc257201686)

[6.1.1 Room 12](#_Toc257201687)

[6.1.2 Question 12](#_Toc257201688)

[6.1.3 Poll 13](#_Toc257201689)

[6.1.4 Choice 13](#_Toc257201690)

[6.1.5 Vote 13](#_Toc257201691)

[6.1.6 Justification 13](#_Toc257201692)

[6.2 API Design 13](#_Toc257201693)

[6.2.1 JSON 13](#_Toc257201694)

[6.2.2 API Methods 14](#_Toc257201695)

[6.2.3 jQuery 14](#_Toc257201696)

[6.3 Django Design 15](#_Toc257201697)

[6.3.1 Rooms Application 15](#_Toc257201698)

[6.3.2 Polling Application 15](#_Toc257201699)

[6.3.3 API Application 15](#_Toc257201700)

[6.4 Third-Party Applications 17](#_Toc257201701)

[6.5 Testing 18](#_Toc257201702)

[7 User Interface 18](#_Toc257201703)

[7.1 Overall Design 18](#_Toc257201704)

[7.2 Design Layout 20](#_Toc257201705)

[7.3 Inspiration 21](#_Toc257201706)

[8 Testing 21](#_Toc257201707)

[9 System Walkthrough 21](#_Toc257201708)

[10 Project Summary & Reflection 21](#_Toc257201709)

[11 Time Plan 22](#_Toc257201710)

# Executive Summary

**[To be completed after the report content is finalised]**

# Introduction

The purpose of this report is to describe and explain in detail how our group completed our project; the Democratic Conferencing Tool, and all of the stages involved. It will include relevant content from the earlier interim report, while any information that is deemed to be unnecessary will be removed. All documentation completed throughout the project will be present in this report or references made to where it is included within the interim report.

The structure of this report will follow that of the interim report, with the initial section focusing on the problem we hoped to solve as a group, and then move towards the research and planning stage; with sub-headings for the appropriate research sections. The research section will feature condensed material from the interim report, with additions made to account for both changes and any new ideas we have gathered from other sources. After the research is finalised, the focus will shift to individual specifications, with particular interest paid to the technical aspects of the system as that has become the most important part of the project in recent weeks.

**[Introduction to be completed when other sections are written as I’m not sure of the layout yet]**

Research

## Summary of Completed Research

The majority of our research is contained within the interim report, as most of the early focus was upon researching ideas for our system from similar products. This focus has shifted to a more technical aspect as of late; therefore this section will be brief in comparison.

We started with market research into similar web-chat software, in the hope that we could gather ideas that would be suitable for what we wanted to achieve with our system. We looked at a wide range of products, but Google Wave and Effusia Business IM were the most relevant to our plan. We gathered information about how to portray a simple user-friendly interface from these products, Effusia especially, as well as incorporating the idea of leaving short messages to inform other users of your change in status if you were to leave the computer for a certain period of time.

After gathering a small, albeit useful, amount of information from these products, the research shifted onto the technical side of the system, and we began to look into ideas for frameworks that we could utilise as a base for our program. A discussion began about web-frameworks and desktop-based platforms, and a comparison was made in the interim report along with the advantages and disadvantages of each. We then branched further into discussing particular software within each platform and researched a number of products that are both web-based and desktop-based. Having quickly made the decision that we would be using a web-based platform for our software, we had a good idea of which direction we would be heading in regards to the completion of this project.

To allow for ease of communication between group members, we set up on Basecamp; a project management and collaboration tool, which then allowed us to keep in touch with each other far more easily than if we were relying solely on e-mail. At Basecamp, we would all be able to view our progress at the same location, which made things far easier for everyone, as well as having a function for the creation of a ‘To-Do’ list, which allowed us to set tasks for ourselves and one another and keep track of these with the use of deadlines. Once completed, a task can be crossed off the list and you can move onto something else.

Along with the use of this project-management software, we decided to utilise Subversion software (SVN) to create a repository where all of our work could be stored and shared with one another without fear of files conflicting with one another when saved at different locations. This made it possible to keep completely up-to-date with the project on our home computers, as you can access the files at a separate location to where you were working.

**[Summary of Completed Research to be completed along with additional research (JQuery, etc)]**

Functional Specification

## Overview

myDebate is a service that allows users to participate in a democratic debate or discussion - to achieve a goal of agreement on a particular subject.

The system forces users to vote on a goal at the end of each time period (which is **30 minutes** by default). The goal (an issue that is to be debated) is specified when the debate is created. If all (or a specified proportion) of the participants agree when the vote is cast, then the goal has been achieved and the debate can be ended. Otherwise, the discussion continues for another period.

## User Stories/Scenarios

The user stories or scenarios specified in this specification will help to define how users interact with the system.

### Scenario 1: Jimmy

Jimmy is an 18-year-old college student in London, studying the arts, and has a keen interest in computers. Jimmy is a participant in a number of online communities, where he often has heated discussions, supporting his views on his choice of operating system – Mac OS X. Jimmy enjoys making fun of people that use Microsoft Windows or Linux, and often gets himself into arguments in forums or message boards over the subject.

While browsing his usual online forums, Jimmy came across a thread that simply stated: “Which operating system is better, Windows or OSX? Debate now on myDebate!”. He was intrigued, and clicked the link to the myDebate website.

In order to take part in the debate, Jimmy was required to create an account on the site, which he did. He clicked the link that was sent in an email to his account, and logged in immediately. He then joined the debate about OSX and Windows.

He found that there was a timer counting down in the corner of the screen, and people were putting forward points in an attempt to win the vote of the other participants. When the counter reached zero, a screen appeared asking Jimmy to vote – he picked OSX straight away, and then saw a pie chart that changed as the votes from the other participants arrived. He was impressed at the responsiveness of the system, which seemed to react almost instantly to his clicks and keypresses.

After two periods, the vote was starting to show a majority towards OSX, so Jimmy remained in the debate and put forward his own arguments as to why he likes OSX so much. The next time a poll was taken, 68% of the participants voted in favour of OSX, and the debate was declared finished with a majority.

Jimmy enjoyed putting forward his points on myDebate, and bookmarked the list of debates in his browser so that he could keep checking the latest ones.

## Flowchart

This flow chart gives a high-level overview of how the user interacts with the system, and vice-versa.



## Screen-by-Screen

### Home Page

The home page should offer an introduction to the user, and briefly explain what the application is about. From here, the user can access each main area of the application using a tab-like menu in the header of the page. The header should be present on every page in the application, to retain a consistent user experience throughout.

The header (and therefore the home page) should also contain a button that allows the user to login. Also, search should be easy to access, with a search input field directly accessible inside the header (ready for the user to input a search query).

### Login

The login page should be simple and to-the-point. An input field for the username and password is all that is required. The password should be obscured using asterisks so that it is not visible when typed.

### Register

The registration page allows a user to sign up for an account on the system. The following information is required for an account:

* First and last names (i.e. the full name of the user)
* A username (that is not already taken)
* A password (must be entered twice to make sure no mistakes were made)
* Email address

The email address must be verified by sensing an email to the address containing a link. When the user clicks the link in their email, their new account is activated and they can login. Before this point, the user cannot login (as their account is not yet active).

Again, the passwords should be obscured using asterisks so that they are not visible when typed.

### Debates List

The list of debates should show each debate that is currently open, with the number of users in that debate clearly shown on the left. Also on the left, the list should show whether the room is in join mode or lock mode. Finally, each debate should show who created it, and when (in relative time format, e.g. 3 hours ago).

### Debate Screen

Users can only be present in one debate at a time, so if they attempt to join a debate when they are already in another, they should receive an error that tells them which other debate they are already present in.

When the user is in a debate, the largest portion of the screen should be taken up by the messages that are being exchanged in that debate. On the right, the list of users currently present in the room should be shown, with the countdown timer at the bottom of this panel.

Underneath the messages list, the user should have a box to enter their message, which has a submit button to the right of it. The user should also be able to activate this button with the enter key.

When the user arrives on the debate screen, the message input field should be automatically in focus, so that they can begin typing without having to click in the field.

### Voting

When a debate reaches the end of a period, the voting window should appear in front of the debating window (and the background should be greyed out, as it is unusable). Once the user has cast their vote, the voting window should change to a graph showing the results of the vote in pie-chart form. This graph should automatically update as the votes from the other users are cast.

Once all the users have voted, the person who is in control of the debate has the choice over whether to go to another period, or end the debate.

The person who is in control of the debate is either the person who created it, or the person who has been in the room longest if the creator has left. If the creator re-joins the room again, they ***do not*** become in control again, unless everyone that has been in the room longer than them leaves.

### Creating a new Debate

Users should be able to create new debates as they wish. To create a new debate, the user is required to enter the **question** (i.e. the issue that will be debated) and **at least two** choices (that users can vote on in response to the question). They can then also alter the length of the period and the join threshold (the length of time remaining before the room is locked to new members).

### Users List

Users should be able to browse through the other users on the system, in a list that looks similar to the list of debates. This list of users should show a thumbnail of the user's avatar on the left, and their name should be a link to their profile.

### User Profile

A user's profile should contain their avatar (in larger format) on the top right of the page, with the information they have entered into their profile in the center.

If the user is looking at his/her **own** profile, there should be an option to edit it, so they can enter, edit or remove information from their own page.

### Search

The search page should contain a prominent search query input field at the top, with the search results listed below. Search should only query the names of debates (and not return anything other than debates), so the list should look the same as the debates list screen.

If no debates are found for a given search query, the results list should not be present, and instead should display the text “No results found”.

### FAQ

The Frequently Asked Questions page should be a one-page section with questions specified in headers, and the answers beneath them. This page should also include images to make the descriptions easier to understand.

## Requirements

### Functional Requirements

1. The system should have a web-accessible interface
2. The system should allow users to login and logout
3. The system should present users with a list of available debates
4. The list of debates should show how many users are currently participating in each
5. The system should allow users to enter a debate, and chat with others in that room in near-real time. This requires the user to be logged-in
6. The system should allow users to see who else is in the debate
7. The system should allow users to leave a debate, and return to the list of available debates
8. The system should use an asynchronous method of communication within the debate (i.e. the page should not refresh when a new message is available.
9. The system should allow users to sign-up for a user account.
   1. Users must specify their full name
   2. Users must choose a username and password
   3. Users must enter their email address
   4. The system should send the user an email, which includes a link that the user must click in order to “activate” their account
   5. Before the user has clicked the link in their email, the account should be inactive - so the user cannot login
10. Users should have a profile, containing their personal information shared with other users.
11. Users should be able to edit the information in their profile.
12. Users shoud be able to change their password.
13. Users should be able to create debates.
14. Users should be required to input a “poll” when creating a debate. This should represent the “objective” of the debate - all members should agree on the issue named in the poll. The poll also requires a number of choices, upon which the users can vote.
15. Users should be able to choose the length of the “period” when creating a debate (a default value of 30 minutes should be provided).
16. Users should be able to choose the time at which a room “locks” - so that no new members can join, before a poll is due. This means that new users cannot “hijack” a debate a few seconds before a vote.
17. The system should switch to a vote at the end of each specified time period.
18. When in the voting mode, users should not be able to enter messages in the conference. Everyone is required to vote on the poll specified at creation time.
    1. One the poll has been completed, the person who is unofficially in charge of the room has the choice over whether to end the debate or go to another period
    2. The person who is in “charge” is decided by who has been in the room the longest
19. The system should allow users to search for debates quickly and easily, so that sorting through a large amount of debates is not necessary.
20. The system should have a Frequently Asked Questions section.

### Non-Functional Requirements

1. The web interface should be accessible from any platform, using either of **two** main browsers – Firefox and Google Chrome.
2. The system should be secure in that unauthenticated users cannot access debates
3. The system should be available
4. The system should respond to requests in a reasonable time period
5. The FAQ should be comprehensive, reflect relevant questions that users would ask, and aid the user in using the system as a whole.

User-Interface Specification

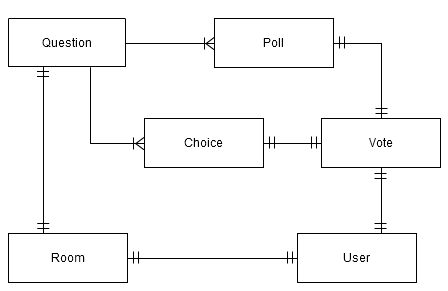
Implementation

## 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 person in control 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 person in control 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-1).

## 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. Due to the way the rooms application works, it depends on the polling application, which is described in detail below. 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.

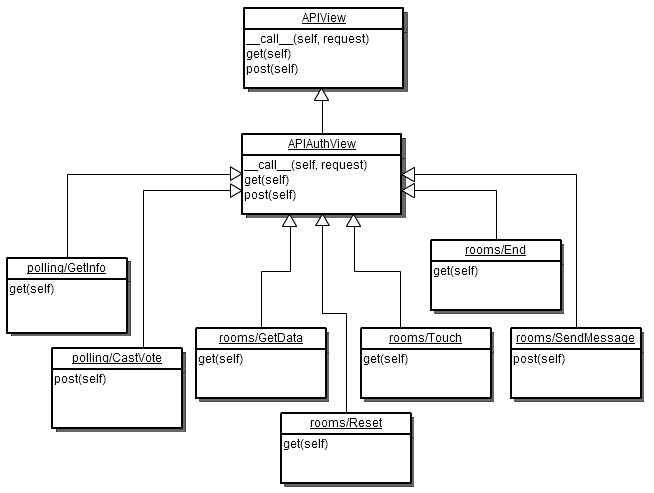
Also, the views themselves in the API application are written differently. All the views extend from a common class, **APIView**. There is also a class called **APIAuthView**, which acts as a convenience class that does not allow the API view to be used unless the user is logged in.

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).

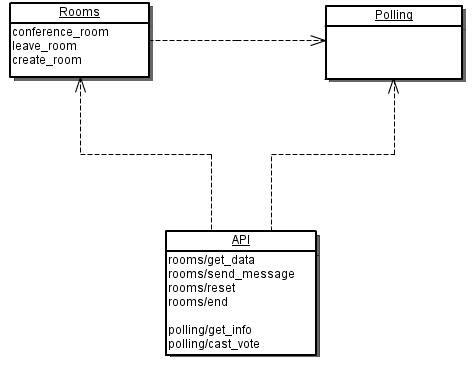
The structure of the API application's views can be represented by the following diagram:



It is clear from the diagram that all of the operations in the API inherit from the APIAuthView class, as all operations require the user to be logged in.

The option of combining the two base classes into one class was considered, but the above solution was deemed to be conceptually sound, and fit for future expansion with API operations that no not require authentication.

The Django design is most easily represented by a diagram. The basic UML diagram below shows the applications in the Django project, with dotted arrows representing dependencies (for example, the API application depends on both the Rooms application and the Polling application).



Alternatively, the Django project could have been designed by incorporating the API code into the relevant application to which it belongs. For example, the rooms API views could have been included inside the *rooms* application, and similarly for the *polls* application.

This, however, would mean that the code that is common to the API views (the APIView and APIAuthView base classes) would have to either be rewritten, or placed in a central location, resulting in a similar layout in the end (as rewriting code is obviously not optimal).

## Third-Party Applications

One of the major advantages of Django and the way it’s applications are designed, is the ability to download and use applications that others have written in other projects. In the group project, we used two third-party applications: **registration** and **haystack**.

Registration provides a reusable way of allowing users to register for the site, with e-mail validation. This was an important feature for our project, especially for the Open Day demonstration, which usually takes a long time to code.

Haystack is a front-end for the Python search framework called **whoosh**. It allows Django projects to index the content in their database, and provides an easy-to-use API for querying the search index. This allowed us to quickly implement search, which is also an important part the project.

There also exist applications for voting, but since this is such a core part of the project, the group decided to write the polling application from scratch, including all the functionality that was needed – and none that wasn’t. This gave us greater flexibility over the API that is exposed from the polling application itself, and as a result from the project as a whole.

## Testing

One of the greatest advantages of using an established framework such as Django is that the code that our project is based on has already been extensively tested. This negates the need to test simple cases such as attempting to login with no password, or specifying an input that is too long.

Instead, we will focus on testing the *functional* aspects of the system, as it is perceived from a user’s perspective.

# User Interface

The first section of this documentation will

**[This section shall be formatted]**

## Overall Design

Why did we make the UI as such?

* Usability
  + Simplicity
  + User friendly
  + Neat and tidy looking
  + Not too complicated

Layout

* + Common tabs and links
* Home
* Debates
* Users
* Search
* Help/F.A.Q.
  + Layout and tabs are placed similarly to the initial prototype
  + Clear view of every system functionality

Forms (Button/Tabs/Links)

* + Login (Button/Link ?)
  + Logout (Button/Link ?)
  + Rooms, Profile (Tabs)

Inspiration

* + Various applications such as
  + iChat
  + Adium
  + MSN Messenger
  + Various websites where chatting / forums were present.
  + Facebook

The main designs are based on the functionality of the project. The usability was considered and the designs are made to be simple and based around the initial prototype design with all functions included. Efficiency and user-friendliness are all considered while designing these pages and it is made to be clear and easy to use.

There is a common header where some relevant information can be found such as the “help” page. So the help page will always be there when a user have difficulties in any part of the system. A search bar makes the searching of a room/title/topic easier.

The common footer consists of the SVN Revision number on the bottom right of the page and the developers name/group on the bottom left.

There is also a “Recent” news section. The developers can update users with the new functionality/features.

-Before Logging In/Unregister User

Users/The public will be directed to the Homepage when they first visit the page. There will be an option for register and log in. Unregistered users are not allowed to start a debate or a view the contents of the room. However, they could see the room’s names/titles. They can also see the list of the registered users, but they are not allowed to view the profile of the users without logging in.

-Registering

The most practical way for the registration of this website is the forms, very simple yet effective. Users are required to enter their details and also a valid e-mail address as an email conformation would be sent, right after they click the “Submit” (button/link).

-While logging in

The conventional way of logging in to a website, users are prompted for their (username/email) and a password. These will be in the form of forms. \*\*(There will be an option for forgotten password or username as well just in case the users forget their login details)\*\*

-After logging in

After logging in, the header will show that a user is logged on and it also shows the username. A logout option will be available so that a user can log out anytime he/she likes. A logged on user has the ability to access the Debate Rooms, User Profiles and he/she can start his/her own debate.

## Design Layout

Home

The “Home” section is just a page where the introduction of what our website does.

Nothing too fancy, just a brief explanation with the “Recent Updates” or “Changes” on the right side of the page.

Debates

The debates are displayed as a list. Each debate is wrapped in a box and it takes up a row each. There is a button for creating a new debate, above the list of rooms. The numbers of participants are shown in the left hand side of the name/title of the debate.

Users

In the “Users” page, the registered users are listed. They might have a picture next to their names depending on if they have an account on Gravatar. If they don’t, they would have a “Gravatar” default picture.

Search

---More to be added---

Frequent Asked Questions (FAQ)

The “FAQ” page will contain the questions and answers likely to be asked by a user. Therefore the FAQ will cover the whole system.

---More to be added---

## Inspiration

After doing much research on what design would be best. It came to a conclusion that to have a user-friendly, good looking website, it will be best to stick to the old fashioned simple and nice approach. Nothing too complicated and clear seemed to be the best one to be going for. Therefore, we now have a site whereby it is looking very simple with visible links and tabs for easy access and navigation around the site. The “Frequent Asked Questions” page is accessible by registered and non-registered users.

Before designing the final design, the group had decided that it would be best to keep it around the initial prototype design. Why change something that is already looking good? During the research, various websites and applications had given us the idea on how the debates would be like. There are some websites that have forums and chatting capability. For example, our very own computer science forum page; they have their discussions listed in a list and we(as users) can see the lists of discussions available.

The debates for our project take place in a chatroom with the basic message input field, message displaying field and box where the joined users are shown. The inspiration here comes from a few chatting and instant messaging system, such as Adium on a mac and MSN Instant Messenger on a Window machine. The basic input and display boxes are essential for communications.

On the university’s student union page, they have a polling that students can vote. The results are shown in a pie chart with percentage and different colours representing different opinions.

--More to be added--

# Testing

System Walkthrough

Project Summary & Reflection

Time Plan

**[Detailed time plan will be added upon completion of the other sections in the report]**

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