**CS673S15 Software Engineering**

**Group Project - Communication Tool**

**Software Design Document**

|  |  |  |  |
| --- | --- | --- | --- |
| Team Member | Role(s) | Signature | Date |
| Oscar Ingham | Implementation / Environment & Configuration Leader | *Oscar Ingham* | 04/16/15 |
| William Wilson | Team Leader | *William Wilson* | 04/16/15 |
| Yuyan Zhang | QA Leader | *Yuyan Zhang* | 04/16/15 |
| Shishuang Shu | Requirement Leader | *Shishuang Shu* | 04/16/15 |
| Xinzhu Xu | Testing | *Xinzhu Xu* | 04/16/15 |

**Revision history**

|  |  |  |
| --- | --- | --- |
| **Version** | **Author** | **Date** |
| 1.0 | Oscar Ingham | 04/16/15 |
| 1.0 | William Wilson | 04/16/15 |

[Introduction](#h.87t9hln2vjz0)

[Software Architecture](#h.buttcq9i221r)

[Design Patterns](#h.x18fj36s1121)

[Key Algorithms](#h.mtfbusfb0eq3)

[Classes and Methods](#h.7ucksmkf6rzx)

[References](#h.15tmymhipvdv)

[Glossary](#h.8n34lvocupub)

# Introduction

This document outlines the software architecture of the communication component of our project management tool. This component is designed to allow multiple users to share messages separated by topic channels in near real-time. Our application shows the online availability for all users, allows any user to create new named channels for conversation, and allows users to search for previously sent messages. Additionally, we enable the users to upload and share files and use a set of emoticons in messages in any of these channels.

Our principal design goal was to keep the application responsive for optimal user-experience. We wished for messages to be broadcasted to other users with as minimal delay as possible. Additionally, we wanted all features to be logically organized with an attractive interface so that the user could easily use all features on any device with minimal-to-no training. As our communication application is just one component of a larger project management application, we also sought to seamlessly integrate our component into the main project.

# Software Architecture

During the early design phases of our application it became apparent that pure Django was not an ideal fit for a real-time communication application. Django is built around a typical request-response design wherein the user issues an HTTP request to the server, which return more-or-less static web-pages.

A pure Django approach would require us to constantly refresh the page to retrieve new messages. This is not ideal for users, as they would be forced to deal with constant page refreshes and any unsent messages that were partially entered would be lost. This would also introduce a significant latency before any user can see messages that are sent. We explored using JavaScript to constantly poll the server for new messages via AJAX, but this introduced the same latency concerns and would introduce excessive stress on the server, which would need to constantly re-render content, and would severely hinder application scalability. The frequency of new messages would also prevent the possibility of alleviating this stress via caching.

Because chat is an event-driven use case, we began exploration of alternative, asynchronous architectures. We quickly discovered SocketIO, a mature library which allows users to create persistent connections to the server via WebSockets. SocketIO also allows users to seamlessly fall back to alternative protocols (such as long-polling) in the event that their client does not support WebSockets. The client-side JavaScript could be easily extended and used in our application. However, SocketIO also requires a server-side component which requires an asychronous, event-driven architecture.

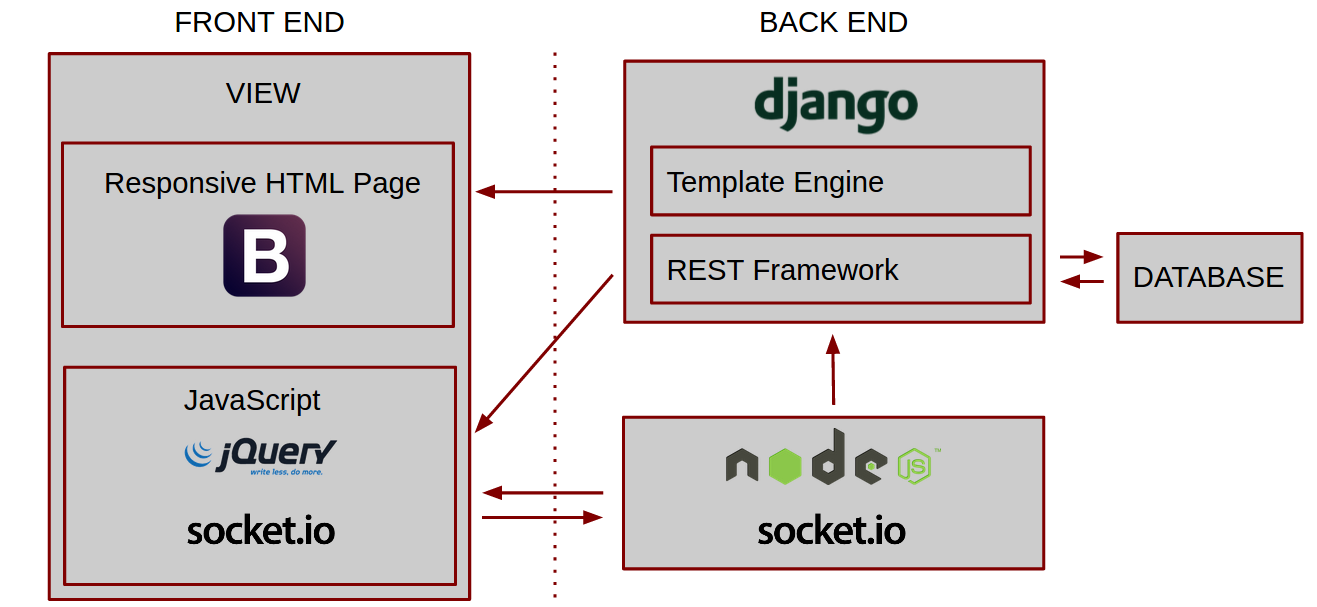
We initially looked at the possibility of extending Django to support WebSockets (which it does not do natively), but found only poorly-documented and out-of-date solutions. As the remainder of the group would also be using Django for their projects, extending the main framework could potentially introduce subtle bugs in downstream functionality. We decided to broker our all requests to a separate server-side process.

A popular Python solution is the Tornado framework, but we found greater documentation and examples using a server-side JavaScript framework called NodeJS. NodeJS is ideal for this sort of application because it is designed around a core event-loop and uses function callbacks to handle all components asychronously. Consequently, IO-bound tasks such as chat are able to scale to a much greater extent than Django could handle itself.

We also made the decision to serve our main webpage as a single-page-application using native Django. Single-page applications are attractive because they only load the page at first, then all updates contain only the necessary data. However, this design would require alternative means to access data. Instead of rendering each page completely, we decided to use a REST API to store and access archived messages, store available chat rooms, and access the current list of users. We decided to use the AJAX features built into the JQuery framework to access the data from the server in JSON format.

We decided to use the Django REST Framework to establish our REST API. This framework builds on standard Django models, defines allowed HTTP methods and data serialization techniques and exposes them as REST API endpoints in either XML or JSON format. It also provides built-in support for Cross-Origin request handling, Cross-Site-Request-Forgery prevention and a convenient user interface for development.

**PROJECT ARCHITECTURE**



# Design Patterns & Principles

* Model-View-Template (MVT)
  + The Django web framework is built on a Model-View-Controller variant called Model-View-Template. As with MVC, each portion of the application is separated into distinct functions, however the terminology and use-cases are slightly different. All data storage and access is handled by the Model layer, which utilizes Django’s ORM to abstract data persistent to the database. In Django, the archetypical MVC “controller” is handled by a set of functions known as the “View.” These functions coordinate incoming HTTP requests and redirects to dynamically-generated HTML pages based on the context of the request. The “templates” (or View in pure MVC) in Django are a mixture of standard HTML and Django’s pythonic templating language. Here, the page content is rendered based on the request context.
* Publish-Subscribe (Observer-like)
  + Our NodeJS server manages collections of WebSockets used for many of our site functions. On the server, sockets are grouped by function into individually addressable namespaces. Each client creates a sockets for each room that is available and subscribes to new messages on that channel. Each socket persists for the duration of the user session. When a client enters a message, it is sent to the server on that socket and the server brokers the same message to all users connected in that namespace. When a new message is received, the message is added to the appropriate page using client-side JavaScript.
  + Each client subscribes to a “global” namespace on page load, which allows us to send general instructions to all connected users. These events include new rooms being created and new users connecting to the site. This allows us to make new rooms available and display the online status of users.
* Mobile-First
  + As more users consume the internet via mobile devices, it is necessary to adapt websites to smaller screens for optimal user experience. This often requires a dramatic re-work of UI workflow. In our case, we have used the Bootstrap CSS framework to create a fully-responsive experience, which guarantees a usable web-site on any sized device.
  + By minimizing the amount of data that is transferred when a message is sent, mobile users are able to see new messages much more quickly over 4G wireless connections.
* REST
  + REST is an architectural style that abstracts data management to HTTP requests. Data access, creation, updates, and delete are all available via the corresponding HTTP verb. This enabled our application to still use the Django modeling system, while allowing other components of the application access to the same data. The HTTP status code of the response allowed us to ensure that data was being managed properly.
  + All data is accessed using AJAX requests from JavaScript.
* Single Page Application
  + Single-page applications load the main page data only at application start-up and then dynamically updates the view with just the data that is necessary. This minimizes unnecessary data transfer and leads to very responsive applications that are well-suited for mobile devices.

# Key Algorithms

* + **Search:** The program user provides a text string. It is sent to the Django server using the Django REST API. The Django search function looks for a match of the user provided string in the text fields of the Message table. All message objects that contain a match are returned as a json object through the REST API.
  + **Socket management:** Sockets represent a bidirectional communication channel between a single client and the server. They are grouped into logical namespaces based on communication channel and dynamically generated when new users connect. At application start, the NodeJS server establishes a global namespace and retrieves a list of all available chat-rooms from the REST API. Each room is given a separate namespace to broker messages from other clients. The global namespace is reserved for other, generic functions such as announcing that a new room is available, or that a new user has connected.
  + **File upload:** Because file uploads via HTTP are chunked into many separate requests, we needed a library which would help manage these asynchronous requests and merge them after all of the data is available. The ‘multer’ library in NodeJS was able to serve this feature for us. All files are uploaded to a common directory at first, but then moved into a new directory defined by a unique hexadecimal SHA1 hash. This allows us to retain the original filename with no worries of clashes if the filename already exists. The files are served from the NGINX static directory, alleviating the strain of serving these files from Django or NodeJS.

# Classes and Methods

Classes:

User

This class is used to store information about the communication tool users. This class is imported from the Django library.

Fields:

name: The name of the user

Room

This class is used to store information about each group.

Fields:

name: The name of the room (group)

Type: Charfield

description: A description of the room

Type: Charfield

public: Set to true if the room is public

Type: Boolean

Message

This class is used to store information about each message. Two of the fields, user and room, are references to other classes.

Fields:

text: The text of the message

Type: TextField

time: The time of the message

Type: DateTimeField

room: The room where the message belongs. This field is a Foreign Key.

Type: Room

user: The user who sent the message. This field is a Foreign Key.

Type: User

at\_message: Set to true for an “at\_message”

Type: Boolean

UserRoom

This class is used to track the relationship between Users and Rooms. It takes a many to many relationship (users to rooms) and makes it a one to one relationship.

This class is necessary because we need to serialize objects within an object. That does not seem to be possible with a many-to-many relationship.

Each UserRoom object must be unique. There cannot be two objects where the user and room are the same.

Fields:

user: A user who is a member of the room. This field is a Foreign Key.

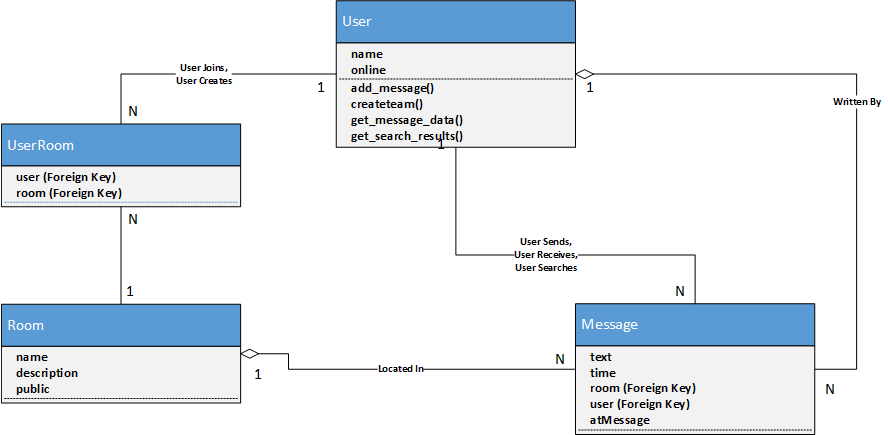
Type: User

room: A room where the user is a member. This field is a Foreign Key.

Type: Room

Class Diagram

The UserRoom class could have been avoided with a many-to-many relationship between users and rooms (this relationship records the rooms where a user is a member). Many-to-many relationships did not seem to work well with the REST API. We could not find a way to serialize objects within an object when there was a many to many relationship. We wanted to serialize objects within an object because it reduced the number of API calls necessary to get all of the data that we needed. For example, if we wanted to retrieve a message, and we did not serialize the objects within the message object, then we would not get the room and user data, we would only get references to the user and room data. Then we would need to make more API calls to get the room data and user data.



createteam(): This function uses the REST-API PUT method to add a room to the Room table.

add\_message(): This function uses the REST-API PUT method to add a message to the Message table.

get\_message\_data(): This function uses the REST-API GET method to to get all of the messages associated with a Room.

get\_search\_results(): This function uses the REST-API to search the text fields of all the message objects in the database. Message objects with matches are returned.

# References

Bootstrap: <http://getbootstrap.com/>

SocketIO: <http://socket.io/>

NodeJS: <http://nodejs.org/>

Django: [http://www.djangoproject.com/](https://www.djangoproject.com/)

Django Rest Framework: <http://www.django-rest-framework.org/>