RoomMeet: Internals

Brian Chen, Dorothy Chen, Michael Danielczuk, Leonidas Tolias, and Campbell Weaver

RoomMeet is a web application that was developed using Django–a high-level Python web framework that can be found at www.djangoproject.com–and PostgreSQL–an open source database that can be found at www.postgresql.org–and is hosted on Heroku–a cloud platform as a service that supports Python. To communicate between Django and the PostgreSQL database, Psycopg2–a PostgreSQL adapter for the Python programming language that can be found at initd.org/psycopg/–is used. All front-end work in designing the website is done using Twitter Bootstrap and basic HTML and CSS.

The basic file structure that defines the application runs through the Django structure–urls are defined in the urls.py file and each url points to a function in the views.py file. Each function in the views.py file renders an HTML file located within the templates folder with a certain context using Django's templating system, and the webpage is displayed. Along the way, the HTML files themselves are enhanced with the use of CSS and Twitter Bootstrap, and the database is consulted and changed within the views.py file, according to the objects defined in the models.py file. The settings.py file allows the different applications involved to interact, as well as linking the database to the main roommeet application.

A large part of the appeal in the website relies on its automatic updating of the various house and friend lists that are linked to each user, as well as the map background itself, without reloading the page. In order to maintain these lists and the map in real time, AJAX POST requests are sent when actions such as clicking on the map and its icons, a filter, or buttons within the list or person popups (all located in the gmap.js file). These requests are sent to urls specified in the urls.py file (such as /get_marks/ or /meet_person/) and Django calls the appropriate function in views.py to determine what action should be taken at this address. Inside each of these views.py functions, the POST request information is interpreted, then the template tags in each html file corresponding to the different lists are filled with the appropriate updated context, so the updated information can be displayed.

This updating process shows the true power of Django's template system: not only does one base html file serve as template for other html files within a site, but also html files can be changed dynamically through the use of template tags. We made wide use of these tags, as the {% for %} tag allows for dynamic list expansion, used in the cases of friend and house lists above, when a new Python list or tuple is passed into the template. These updates are handled in the views.py file using the get_template() and Context() methods from the Django template library, which locate and fill the template tags within the html file, respectively.

Using Django also provides easy interface with the PostgreSQL database that we were using–

instead of writing any SQL, we were able to allow Django to do all of the work through the models.py file. Here, we define different "objects"–in our case, these were Person and House–and their attributes, such as name, working dates, gender, friend and house lists, among others, for a Person or contact name, available dates, contact email, and latitude and longitude points, among others, for a House. Each of these attributes are specified as a type of field–such as a CharField, EmailField, or ManyToManyField and Django does the rest of the work, allowing each of these objects to be saved with all of their attributes with a simple save() command in views.py, which behind the scenes generates the necessary SQL code to add to the database.

Similarly, using the filter() method, the database entries are easily sorted and selected, again without using any SQL. This comes in handy when finding out if a user has already added another user or house, or when providing a list of the user's houses for use in the talk or housing lists. For modularity, each of these objects is defined as its own application, and so each has its own models.py file located within the respective application folder. The main roommeet application is able to recognize these applications based on their inclusion in the settings.py file.

Django also provides an easy way to create and use forms within the web page–linking the views.py file to the models.py files through the forms.py file. Inside the forms.py file, forms for the profile page and for the add housing page exist and each field of the form is defined by a field similar to that in the models.py file, such as CharField, DateField, or DecimalField. However, several other options are present to facilitate the models translation to a form, such as ChoiceField and RegexField. Within these fields of the form, several options are present, such as max or min length and widgets that can define the size, shape, or function of the form field. Once these forms are defined, they can be used in the views.py file to render an html file (such as addhouse.html) with the correct fields (Django takes care of all of the html code based on the type and options given in the forms.py file). The input that is received from a POST request when the submit button is clicked on the page can then be cleaned and accessed as a Python dictionary, with the form fields corresponding to the values. Additonally, Django checks each of the fields to be sure that the input is valid (again according to the type and options specified for the particular field in the forms.py file), storing the errors from each of the fields in the form in a form.errors variable so that they can easily be displayed on the page.

The generous use of Javascript and jQuery within the RoomMeet site allows for the clean and efficient user interaction with the page and quick response time to a user event on the page. Connecting on-click events to each of the buttons on the page that link to Javascript functions allow for the different tables and forms to slide into and out of the main Meet page without the page refreshing, and doing the same for the submit and cancel buttons on the form allow the AJAX POST requests to be sent instead of loading a new page. To allow the different tables and forms to slide in and out of the main page, the different pages are moved on and off of the main page using the animate() function.

Additionally, the Javascript usage allows interaction with the Google Maps API, which is heavily featured within the site (taking up the Meet page and remaining in the background throughout all of the other pages). Basic resizing, zooming, and other map manipulation

functions are all done using Javascript and are located within the gmap.js file. Using the addPersonMarker() and addHouseMarker() functions in gmap.js, we can asynchronously add markers to the map while the aforementioned house and friend lists are simultaneously updated. We can also filter the markers shown by finding only the markers that fit a certain category and displaying those. The heavy lifting for all of the filtering is done in the getMarks function, which sends a POST request that is interpreted in the views.py file. According to the different filters set, the views.py get_marks function compiles a list of matching people and houses and then sends it back to the getMarks Javascript function, which adds the appropriate markers and resizes the map view to fit the markers.