# OIT

In order to make our web application, we used Django as our web framework. Django uses python and so we all coded in python on our local machines using a text editor. Along with the Django included libraries, we included several other non-native python libraries. We used pytz, a python library which simplifies time zone integration, and a Google Calendar python API which provided methods to access and parse Google Calendar events. Pytz can be found by using the provided pip installation tool, and the google API can be found on Google’s website for download.

Django also provides an easy way to deploy the web application for development. In order to test our local changes, we deployed the web applications on our local machines by typing “python manage.py runserver” in the project directory. This would create a local version of the application, which could be accessed through localhost:8000. Then we all used Github to facilitate code sharing and collaboration. Additionally, Github provided version control and allowed us easily pull our code on the server.

Localhost was helpful in testing our changes by ourselves, but it would not be sufficient when deploying to the public for alpha and beta testing, and eventual release to the public. So, we rented a server from the company Linode. We chose Linode because they were at HackPrinceton and offered $50 credit for renting their servers, or roughly half a year of service. Then, we registered for a .tk domain address because they were free, and this domain would connect to the IP address of the cloud server. Finally, we ssh-ed into the server, cloned our Github repository on the server. For deployment, we configured Apache on the server and deployed the Django application through Apache and mod\_wsgi. The latter processes Django applications so they can be hosted by Apache. In order to do this, we copied the Github files to a folder called “server” in the /var/www/ directory. Then we created a configuration file called “skedg.conf” which told the Apache server where to find the source files for deployment, and told mod\_wsgi where to find the python wsgi.py file which acted the same as maange.py; pointing to the location of the settings and other necessary Django source files were. For the server, we turned off the Django debug setting, but enabled the setting which emailed us any python errors which resulted in 500 errors. We also deployed a separate instance on port 82 with the debug setting enabled by executing “python manage.py 0.0.0.0:82” in order to double check functionality before we released new features to the public. In order to assist in deployment, we created aliases named “partyServer” and “publicServer” which automated the process of pulling from Github and deploying to the port 82 development server, and the port 80 Apache production server respectively.

# Our Code

Our code uses the standard Django file structure, along with an extra file that handles all interaction with the Google Calendar server, cal.py. One file automatically generated when creating a Django app is url.py. This file handles all URL requests and appropriates the request to the correct python method to render a page that the user will see. URLs are matched through regex expressions and this feature allowed us to create a unique URL for each event by telling url.py to interpret any URL inputs as an event ID if the parameter input did not match any of the fixed URL patterns. Upon matching a URL pattern, url.py would direct Django to the appropriate request handler python method. These python methods are located within our views.py file. In this file, we have methods which are called whenever the user submits any forms on our page, which generate a POST request, or when the user accesses any links to our website, which generate a GET request. These methods will parse the request, alter the database accordingly depending on the context given by the request, and redirect the user to a success page, or back to the original page with errors. Request parsing was done to verify that the request was not tampered with, such as by changing the HTML or Javascript of the forms on the client side. For example, manageCreator was only supposed to be called if the request was through the POST method and the user was the creator of an event. A benefit of using Django was that Django included which user was logged in in the request, as well as CSRF tokens which made maliciously replicating our forms difficult, and masquerading as another user submitting forms difficult as well.

Then we use two other Django default files to assist with our application. These are forms.py and models.py. As the name suggest, forms.py includes parsing for any forms that we may have, but we only used it to parse our registration form. We double check that the email address provided is unique and several other checks on the registration form. Since Django doesn’t allow for logging in with an email through the default User object, we did a hack that set the user’s email as the username, which would guarantee that all emails were unique and allowed for authentication using email. One negative side effect of this was that the username character limit is 30 characters, as opposed to the default 75 for emails, but we decided that there were few people with emails longer than 30 characters and not to worry about it. With more time, we would have learned how to create a custom User model which we could customize to allow for a longer email.

The file models.py houses all of our objects used in our web applications. We created objects for all data structures that we needed. For instance, we created a model called UserProfile to house all of the user information not included in the default Django user object, such as which Google calendar they use and any unique registration key or forgot password key that they may have. For each object, Django allows us to specify one-to-one, many-to-one, many-to-many relationships between the objects. This allowed us to let models be variables and parameters to other models. And conveniently, Django allowed us to get the set of all models linked to a particular object, for instance we were able to get all models of type Invitee that had a particular event as one of their variables.

Finally we created cal.py. This module contains all methods that interact with Google Calendar. The tasks covered by this module include: getting access to the user’s Google Calendar, validating a stored refresh token for Google authentication, querying a user’s Google Calendar for his/her events between two times, and putting a newly created event into a user’s Google Calendar.

Another key method contained in this module is “findTimes”. Given a list of participants and their respective list of events, each with a start time and end time, this method will determine who can meet at what time intervals they can meet. The method follows two main steps. For the first step, the method uses a sweeping-line algorithm to get a list of distinct times and who can meet starting at that particular time. The algorithm puts the start times and end times into a priority queue and indicates whether it is a start or end time, as well as whose calendar the event belongs to. Then the algorithm goes through the priority queue and looks at the times in sorted order. It keeps track of who is free to meet at each time, by adding a participant to the list if they have no current events, and removing a user from the list if the sweeping line encounters the start of one of the user’s events. This runs in O(n log n) time.

After finding a list of distinct times, the method now finds the ideal times. The algorithm then considers all subsets of consecutive times from the list of distinct times. This will give a time interval and who will be able to stay for the whole interval. But, it is possible that the specific set of people who can make that certain time interval will actually be able to meet for a longer, extended time interval. So for each of these subsets of consecutive times, the algorithm checks if it can potentially extend the time interval to a larger time interval. Then if this larger time interval is longer than the event duration, the algorithm will put this time in to a list of possible time intervals, as well as who can make that particular time. This algorithm checks every possible subset of consecutive times, and for each subset, tries to extend the time interval. A brute force approach would take O(n3) time, but through clever manipulation, we were able to implement an O(n2) running time algorithm. Thus, getTimes runs total in O(n2) time, where n is the number of events.