Phase 3 Report

Overall, the software architecture plan we proposed in Phase 1 remained mostly untouched; that's more a product of it being a general layout for a modern web server than us having excellent foresight. As a team, we decided upon one major point of design revision, our data model. We had originally planned to store the course event data in our webserver's database. However, as soon as we got our integration with Google accounts out of the way, we found ourselves asking, "What's the point of reinventing something that's already solved? Google already stores most of the details we would have wanted in our Calendar and Event models. Persisting our own version of the data server-side, then having to sync with Google just introduces an extra step of processing. As our discussion of the problem carried on, we also discovered a number of subtle benefits to using the user's Google account as our remote "database". Some of the gains we get for leveraging the service this way is:

- Excellent security, no sensitive data gets persisted anywhere on our machines
- Built-in Android integration, events generated by our app automatically get reminders on your Android device because they're on your Google Account!
- The option to share events/schedules generated in our app to non-users like you would any Google calendar
- Speed and efficiency. Prior to this change, after receiving the events from the Google API we had to query all of our models and merge the information based on the id assigned by Google. Now we pre-process the data without querying our database when getting events.

After carefully reviewing the options, the decision was clear.

What this change involved was having our data models serialize to a network call instead of a DB relation (and vice-versa for deserialization). So instead of having an ORM map our object contents to a DB, we ‘persist’ data by serializing the object to JSON and sending it to Google’s API in a RESTful POST (retrieval is the reverse with GETs). We had to also introduce the concept of ‘metadata’; information about Events that we wish to store that doesn’t exist in Googles representation (this includes voting data, comments, school classes, etc). We treat all the metadata in our database as correct and send that along with every post/put to a google event. This allows multiple users to edit the same event concurrently, what the second user sees on the screen will not overwrite what the first one entered. Of course, there had to be some clever solution gets this metadata to Google’s side, and we’re still deciding on how this information should look outside our application, but we have something for the time being. Right now, we store a JSON serialized version of our metadata in the description field of a Google Event, and prevent users from editing that detail of our Events outside of the application. This semi-structured data can be formatted however we like in the future.

Unfortunately, from an architectural point of view, these changes moved some of the responsibility of our Models.py classes into our GoogleAPIWrapper (actual name: google_api_interface.py), which isn’t perfectly in tune with SOLID principles. A code refactoring might be in order down the line, but there are too many other features to implement before the end of the term, so this remains a low priority. However, as it stands, this class is still a great example of the adaptor pattern in action; it translates data in two directions between two otherwise incompatible, and completely separate, systems. Other extremely common patterns we’ve used are:

- state and chained responsibility during authentication
- mediation of multi-step flows using the WSGI middleware stack
- MVC in both Django and AngularJs

When it comes to the frameworks themselves, there are many moving parts in software of this scale, but some of the more obvious patterns they use are: publisher/subscriber pattern, templating, and observers.

As far as problems we have faced as a team, there are three that stand out more than the rest. The first was coming up with the above changes to our models in order to create a reasonably robust MVP. We solved that through discussions as a group. The second was our differing schedules, as some group members are contributing more than others. There is no perfect solution; members that are less free are given the less essential tasks while they wrap their other work up. The last problem is that some members are more skilled in the tools, languages and frameworks we're using. We've held group tutorials, followed online tutorials and fostered an atmosphere of rigorous code reviews and welcoming questions.

On a more pressing note, we're approaching the last feature iteration of our plan and moving towards polishing what we've implemented. The focus is going to be a balance between presentation and code-quality. We will try to set aside time for important tasks such as setting up regression, and other automated testing, refactoring, and rethinking decisions. Nonetheless, I think I speak for all of us when I say, we just want this to look good for the demo, then never to think about it again.