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 want in a Calendar and Event. We've come to realize that our current processes are not benefiting for storing our version of the data on our server and joining that with the information google does store. That being said, as our discussion of the problem carried on, we realised there were a number of subtle benefits to using the user's Google account instead as our remote DB. Some things we get for free this way are:

- 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 wepre-process the data without querying our database when getting events.

By carefully reviewing the options, the benefits were apparent, and so we decided to turn our Data models into a method to protect against data loss without sacrificing performance. We treat all the metadata in our database as correct and send that along with every post/put to a google event. This allows users to edit the same event at the same time and what the second user sees on the screen will not overwrite the first. Of course, there had to be some clever solutions to store 'metadata' (data like votes, comments, school classes; the stuff not stored in Google Calendar), and we're still deciding on how this information should show up in from outside our application, but we have a solution that works for the time being. That solution is store a JSON serialized version as the event description, which is not directly editable by the end-user. When doing this, we also realized some minor issues with the structure of some of the objects in the database, which we adopted to work with our much more performant plan.

Unfortunately, from an architectural point of view, this change 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. 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. One direction takes our JSON representation of events, converts them to a RESTful HTTP call at the appropriate URL, then wait for the response and formats it in a way our system understands.

Other extremely common patterns we've used are state and chained responsibility, during authentication; mediation of such multi-step flows using the WSGI middleware stack, and of course everything used to build up Django's MVC framework. The frontend code, built using AngularJs is also MVC based and makes heavy use of the publisher/subscriber pattern. There are so many moving parts in software of this scale, it might be easier to list unused patterns. From templating in the Django, to the observers maintaining consistency in Data between separate layers examples are easy to find. In fact, in the latter case, I could be talking about data between Angular to Django, or across Relational Tables to Django Objects, or any number of other cases.

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 important 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.