Sunday, 29 September

This evening, Petr completed setting up Amazon Beanstalk with Django installed, database initiated and admin interface activated.

Admittedly, it took several hours to set up my local development environment due to the learning curves associated with Amazon Web Services. *In the end, I think the time spent was well-justified as this workflow with git deployment really sped things up and reinforced our original plan of continuous integration.*

In getting this to work, I inadvertently and inevitable committed and pushed our Amazon secrets to Github. To ensure the security of our project is not compromised, this must be removed from all historical commits as soon as possible.

Monday, 30 September.

The initial task was to get Amazon to host static files for Django and have it be configured to point to an Amazon S3 bucket. For this, I used boto and django-storages for a robust solution for separating media and static files.

Once this was complete I downloaded the latest Twitter Bootstrap to serve as our CSS/Javascript framework and created a basic HTML template to test that indeed our files stored on S3 were indeed being served correctly by Django. This was completed on

Tuesday, 1 October.

Once the static files hosting was resolved, Han and I worked on modelling our data with ERD diagrams. This was immediately implemented as a Django model and its corresponding table schemas synchronized in the database.

A problem I encountered is the the amount of time taken Amazon Beanstalk's app deployment. Since it had to pip install everything in requirements.txt, execute all custom commands, and update files on every deployment, the total overhead is absolutely massive if you're just tweaking a few lines of code or trial-and-erroring your syntax. This was simply not feasible so I updated the Django project settings to support local development on a SQLite DB and shared this configuration with the group.

This allowed me to have the Django management tool run a local server and watch for changes in the project and update accordingly. This meant I could make a change and see the result immediately.

Following this, I immediately registered our newly created models with the Django Admin interface and got the home view of our application to display poll questions.

One challenging aspect was allowing users to input rich content and media. I solved this by using markdown as user input and using the Pagedown widget that Stack Overflow uses. Using markdown, I override the save function of our poll question model and have the markdown version of the content convert to markup (HTML). We save a HTML version so that this does not have to be rendered when serving our pages. This widget as embedded in the admin interface and tested on the home view using the ‘safe’ Django template tag to render the HTML.

Additionally, I also installed a number of third-party Django apps to support common functionality such as object tagging, voting, etc. One of these was a bootstrap 3rd-party dajngo app which supported bootstrap forms (from my previous experience, it was quite difficult to use bootstrap tags in Django forms and required special modifications. Installing this should make things far easier).

Wednesday, 2 October.

Nelson, Han and I met up in Circular Quay to discuss the authentication aspect of the project. We went over our design and decided upon the type of authentication and security we needed to provide to our users. Further more, we decided upon how we would extend the native User model with our own attributes, etc.

Saturday, 5 October.

My local development workflow was working well, but I wanted to see if I can replicate our Amazon Beanstalk app instance’s environment in a Python virtualenv and connect directly to Amazon RDS and S3 locally. I updated the Amazon RDS security group to allow incoming traffic from all IP addresses as a temporary solution - this still needs to be addressed using SSL certificates instead. With this in place, I could work on the project and see ‘live’ data, instead of my local database. This is also a best practice (as suggested in “Two Scoops of Django”), that one should never use different database backends for development and production. In this instance I was using SQlite and MySQL.

After this I installed South for schema migrations. Django’s syncdb function does not support schema changes which meant that to alter a model, one had to supply raw SQL commands for altering the table and handling existing or missing data gracefully. South can take care of this for us automatically. I made schema migrations for our existing app and updated the Amazon Beanstalk custom commands to include applying migrations. Having done this, I will now be able to modify my models with reckless abandonment.

Monday, 8 October.

Followed “Two Scoops of Django” and adopted some recommended best design practices. I got rid of the non-versioned local configurations file and instead reorganized our project directly structure more coherently. Now settings are in its own directory and to be inherited from a base setting file for different environments. We are also no longer storing secrets in these configurations and instead using virtualenvwrapper's environment postactivate hooks to set these environment variables in order to further replicate the Beanstalk instance's environment. This meant that the setting file used locally is the same one used in the Beanstalk instance. One could then supply command line options to the Django management utility in order to use the local setting which used a local database and static folder. Our project file structure and dependencies became far more elegant and logical after this. I then added exception handling for our settings file and used another package to manage project directories, making our configurations even less error-prone and more readable, in accordance with the aforementioned book.

Wednesday, 9 October.

Updated models and schema by deleting a few irrelevant attributes and objects and replacing them with those provided by 3rd-party apps. After this, I was able to properly configure the Admin interface so that questions could be created properly, along with their associated response choices. As part of this, I deemed it necessary to add markdown contend for response as well and updated this in the interface too. Lastly, the home view was edited so that the questions were displayed along with their choices. Also incorporated the django-taggit app to support category tagging. The view, or template rather, was further updated to contain placeholders and functionality yet to be implemented such as voting, forms for providing poll answers, buttons for asking questions, and links to new views, etc. in sprint meeting to take place tomorrow, we shall delegate these respective tasks to scrum team members.

15 October.

Implementing voting of poll Questions using generic views with AJAX support from django-voting app. Essentially, what is required is a url pattern to created which will pattern match the object id and voting direction, and by passing these parameters along with other parameters that specifies the content type, we are able to create the vote objects and use the provided methods to perform aggregation on vote statistics. By passing the `allow_xmlhttprequest` , we can allow it to bypass a vote confirmation page. Essentially, a button click invokes the javascript function that performs the voting by passing the appropriate parameters and sending a POST request to the specified URL. The vote object is created by the view and the score of the object is returned so we can update it in the function bounded to the success event listener. In this function, we then update the controls and the appearance of the buttons to correctly reflect the score of the object and whether the current user has voted by beveling in the buttons accordingly. Though a simple and elegant third-party package, like most others we have used, its integration into our app requires us to invest much effort into getting it to work properly and harmoniously with other existing components, also requiring use of various technologies or standards (in this case, AJAX, javascript, CSS, etc.) In the end, most uses of these third-party packages were justified since it separated the logic between the crux of our application and supporting functionalities/features that were not as important and allowed us to focus on the core

functionality.
Also worked with Han to implement basic search functionality by looking up keywords in questions.

16 October.

In our first presentation, it was noted that on firefox, the glyphicons provided by twitter bootstrap were not rendered properly and the layout was causing the user interface to behave erratically. It turned out to be an issue with the way Amazon S3 served files that for some bizarre reason only affected firefox. This was rectified by resorting to using the bootstrap hosted on a CDN. Worked with Nelson on integrating a user registration by using django-registration, which provides some basic generic views and more notably, email activation. To ensure user security, users must register and await a confirmation email from our site. A cryptographic hash is generated for that user as a activation key and included in the activation link in the email. Once the user clicks this, their account is activated and they may now use the site. If logged in, users can change their password and if they're not logged in and have forgotten their passwords, they can choose to change their password by first awaiting a password reset link in an email. This ensures only the user who registered with their own email may change their own password.

Also configured Amazon SES (simple email service) to be the SMTP server for delivering these emails. Also filed a request to have production access to this service as by default, it is only a sandbox service where you may only send emails to verified users. This is presumably to prevent spam and unsolicited emails. Since we meet the terms of service for production access and don't intend on spamming people with unsolicited emails, we should request production access.

18 October.

Updated poll answering functionality to use AJAX. The radio button click listener is registered to submit the form upon form click and the javascript function takes over (hijacks) the form to process the POST request. We must now update the view so that it returns JSON results of the Poll itself which can then be fed into the Google Chart Tools API.

This also ensures that in list view, when we answer a poll, we can stay in the same spot and have the result load without refreshing the entire page and having to find the poll again in order to view the results.

19 October.

Working on displaying results using Google Chart Tools API without AJAX for now since we have to make a few complex design decisions here. If the user has already answered and specifically wants to view results, we can populate the chart as soon as the document is ready whereas if the user hadn't answered, we populate the chart when he does answer. Here, we don't want to repeat code that essentially does the same thing. Essentially, the post request that registers the answer will respond with the JSON results. The explicit result view will do the same thing. So we must devise a way for those views to return same data. We could also have the answer creation view simply return a link to the result and have the javascript function retrieve the HTML from the result view and render this in the div. But this means we need some way also of invoking the function that draws the charts as these would have been “drawn” already without the relevant HTML elements present.

Experimenting with a django third-party package that renders charts was an exercise in futility since all this package really did was generate the code for the javascript function required to render the chart. This is okay for a single chart but for multiple charts, this will generate the same code for every single chart which is a lot like building 10 bathrooms on one floor when all you needed to do was place 10 cubicles in a single bathroom on that floor.

For a simple fallback in case we cannot take advantage of Google’s API, I have created a basic visual result display with Twitter Bootstrap’s progress bars.

To ensure a sees results for and only for polls he has answered, I implemented a template filter which simply takes a question and user and checks that the user has indeed answered that question. This is more of a UI constraint but we still need to add a lower level constraint the form of user permissions that prohibit the user from viewing results without having answered, amongst other things, like creating, answering polls, etc. without having logged in.

This also prevents users from answering a poll multiple times, but again, is only implemented on a UI level. Savvy users can always simulate GET or POST requests and add answers, etc. So again, we need some validation either on the model-level, form-level or view-level. Most likely will have to be view-level but more research is required.

20 October.

Finally got around to working on poll creation. Had been researching for a few days but hadn’t implemented it since it involved a number of complex components. First of all, when creating a question, one might expect to also create the set of associated choices at the same time (a choice has a foreign key point to the question). This had to involve the use of inline formsets which isn’t the most perfect feature provided by Django and is quite messy to work with. Essentially, we need to populate the inline formsets and feed it to the template and then recieve data (process the post request), we need to save the main form (the question) and then work with each subsequent inline form and valide/save those. Once this was done, we needed to add some JQuery code that allowed the user to dynamically add more choices if the default number of choices (two) wasn’t enough. This essentially clones the last form in the inline formset.

22 October.

Nelson and Han worked on pagination and fixing the search function which I think I inadvertently broke in migrating our views to generic views. Since the generic view supported pagination already, I modified their code to do it more efficiently and updated the templates to use bootstrap pagination components. Han started to work on the sorting of polls in the list view so users can order by popularity, etc.

23 October.

I cleaned up the codebase a bit and formatted the templates, etc. and modified the appearance of the question creation form.

To prepare for the presentation, I removed the placeholder navigation controls, fixed broken links, added basic views to support dead links that were supposed to have their associated views implemented, such as list of tags. Also got rid of the sidebar placeholder and added site-wide statistics for the list view and detailed statistics for a question's detail view.

24 October.

Started implementing some regression testing test cases. We liased with Yigong and he began writing a basic model-based test suite. We will also have most extensive test suites that simulate GET and POST requests to views and verify the correctness relevant server-end tasks. For the front-end, user interface and behaviour of the site itself, we must resort to something like Selenium which can get very tedious and might be overkill at this stage. We should at least get the two aforementioned tests suites implemented first, as the latter if more useful for sites driven by JQuery/AJAX, which our app only uses a bit of.

Started using django-extra-views to handle the creation and editing of our inline formsets.

25 October.