ASN Specification and Design

Specification

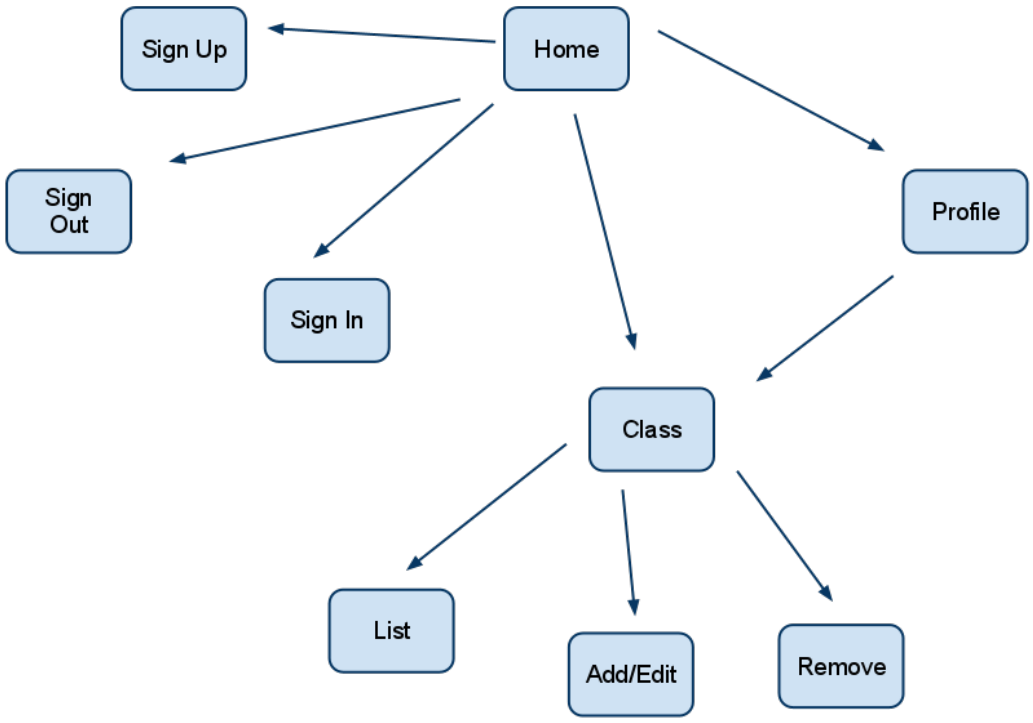
ASN is an anonymous social network that tracks a student's academic interests and displays the academic interests of others. Users will be able to create an account, log in once they've created their credentials, change password, add/edit/remove classes, books, papers, internships, places (eat, live, have fun, etc.) and games. Non-logged in users can see list of courses, books, papers, internships, places, and games that have been posted by students, and with each item, they can also see the number of ratings and average rating. There

Scenarios
Ashley is a student and instead of studying for her Chemistry final the night before, she goes down to Sixth Street, has series of humorous experiences involving a polar bear, ten pounds of rock salt and some rope, that culminate in her staying up till 6 AM. She takes her final at 8 AM and fails it, but in the spirit of her generation, blames her failure on the professor. She wants to vent her frustrations about the professor in an anonymous way, but cannot do so through social outlets such as Twitter or Facebook. She goes to ASN, creates a profile and posts the class and rated the instructor with an "F" and writes a heinous comment. Ashley is vindicated.

Bob is an ASN administrator who thinks Ashley is a little rambunctious with her words. Being a man of Mormon sensibilities, he quickly sets out to delete the comment. He goes to the site and logs in and thanks to his super admin powers is able to navigate to the comment and delete it. Bob hails himself as a paragon of virtue and buys himself a Dr. Pepper. Now on a sugar high he decides to smite the lowly Ashley's virtual representation and deletes her entirely from the system. Huzzah!

Tim is Bob's son, and is watching the exchange, with an eager eye to try the site out for himself. He does so, but on the off chance that his father might catch him logged in, he does not create an account and simply decides to browse. He takes a look at all the ratings for the different categories paying attention to the game ratings and not understanding why the "Human Sexuality" class gets such a high rating.

Site Flowchart



Note: Class branch will be duplicated for every entity

UI Wireframes

In ASN2 Reference

Design

-**Profile Page**

We already know that a student can be associated with a class, a type of place (eat place, live place, fun place, etc), Internship and game. One feature we implemented is a profile page where users can see what classes or internships, for example, they have added. This feature can only be accessed if a user is logged in. No information will show up on this page if a browser is accessing this application or if another user is logged in and the only information in the datastore is information input by another user.

To assure we had the information display for the user that input them, we had to use our association classes to link the user to their classes, books, places, etc. When a user creates

login credentials, a random number is created and used as their id. We can access a user via that id or their e-mail address. Once we get the user via their e-mail, we are able to get all their other information therefore allowing us to display their information in the profile page.

-Recently added/edited Item

The other thing we decided to have show on all pages of the web page is a list of most top five most recently added/edited attributes of a student (i.e., Classes, internships, places, games, etc.) This piece of information is viewable by all types of users and is changing based on any new changes. This is a similar feature Yahoo has on their home page, with the top 10 searches.

We used google app engine's property "db" to indicate the types of the values within our classes. We added another attribute that records the date and time when an attribute is created. Using that attribute, we were able to sort the top five most recent added/edited information and display it on the side. The difference between this feature and the profile page is that this information is visible to all users at all times and is getting updated as new attributes are being changed where as the information is the profile page is constant according to a specific user.

- Change Password

We've also created another feature that allows the user to change their password. This functionality is standard in many web applications that require a password authentication (i.e., E-mails, bank accounts, social networks, etc.) You never know when this feature may come in handy. The user may want to their password to make it harder to crack, or just easier to remember.

We also have validation when a user decides to change their password. The standard form for a password change requires the old password and two copies of the new password to make sure there is no typo. We are validating that the old password they provide matches the original password, when they signed up for the application.

Another validation we have checks the new password provided. Reason we have the user type in the new password twice is because they may mistype it in either the first or second box, and us as the developers in the back end, need to make sure the user knows what their new password is going to be.

- Average Rating

The user should be able to see the average rating for different items, presented in a list.

To implement this with a relational database, you would normally do a join query with a GROUP BY phrase, to collect the average rating, number of ratings, etc. But since Google App Engine is a key-value store, not a relational database, it makes more sense to do something different.

We chose to update the average rating whenever a new rating is submitted to the database,

and store the average in the item. That way, no join query is needed. The alternative would be to calculate it at the time of listing the items, which would be expensive - a manual join would need to be done, by following the association class links.

In effect, we have a denormalized database - this is something you can do in a relational database, for performance reasons. In a normalized database, there is no duplication of information. But sometimes, to avoid doing expensive joins in queries, you store information in more than one place, caching it where you can get at it quickly. This makes sense for a web application which needs to be scalable.

The same thing could be done to calculate the average grade for a class, though we wound up saving this for phase 3.

- Functional Testing

There should be a way to test the functionality of the website, automatically.

One approach is to use plain Python, opening a page using urllib2.urlopen and then parsing the resulting HTML. This works, but there are some tools that make doing this easier.

The one we chose to use is Twill - it lets you write scripts for interacting with a website, checking for any errors along the way. You can fill out forms and submit them, then check that the resulting page (HTML) has certain strings or regular expressions.

Twill is useful for websites that don't have a lot of Javascript, which applies to our site. For sites that have a lot of Javascript interaction, a tool like Selenium actually drives a web browser, but for our purposes that is not necessary.

So you can make a folder full of Twill scripts, then run them all with one command. If anything fails, Twill will alert you. So far, we only have one Twill script, to test some of the functionality for Books. For the next phase, we will add more functional testing.

Each Twill script should leave the database in the same state it started in, in order to avoid polluting the database. So for the Book test, it adds a book with a rating, checks that it got added, and then should delete it. Right now

Twill can be run as simple scripts (for which the syntax is much simpler), or through Python - we might need to switch to Python in order to exercise the website fully, though hopefully that won't be necessary.

- Administrator Accounts

Administrators have a higher access than the browsers and the students. Although students technically have a limited access to the database in the sense they are adding information to

the database when they add a class or a place, and the fact they can edit a place or class, they still can't delete information from the database. On top of that, administrators can delete a user from the database and all their information. Another feature administrators are allowed to do is import data via XML instances. The application will then parse the file accordingly, and fill in the database.

Since we are the administrators of this web application, we assigned the members of our group with administrator rights and manually added those credentials into the database. Any other user credentials that get entered into the database via the web application is guaranteed to be a student user.

- Deleting Items

Administrators should be able to delete items, and student users.

Deleting items from the database should be behind a form, rather than a plain link, so that robots that crawl the web don't inadvertently delete data. So we have delete handlers for each item, which have a form with a Yes button going to the delete action, and another form with a No button redirecting to another page.

Administrators can see a list of Students, and delete them individually.

- Validation

Data should be validated at time of entry - either through import or through form entry. This prevents erroneous data from getting into the database.

Google App Engine models allow you to define properties with built-in restrictions. For instance, you can define a property which is restricted to numbers with 10 or 13 digits, which is used for ISBN numbers. We do this by defining validation functions for each property that needs to be validated, then passing this function to the property constructor.

Initially we had the validation function checking for missing values, in order to match the XML schema, which has some properties required. But this caused problems with the existing import code, which constructs empty objects, then sets the properties. Simply creating an empty object would cause the validation function to fail, throwing an error. The proper way to build model objects through code is apparently to pass all the properties to the constructor. But we had so much code in existence already that we decided to modify how we validate required fields.

So rather than specify that a property is required, we overrode the put method for each object, and checked for missing property values there. If something is missing, it raises an error. This works for both import and form entry.

- Import / Export

Administrators should be able to import and export data, and clear the database.

If the user is an administrator, they see additional menu items in the left navigation bar, including Import, Export, and Clear Database. This functionality is the same as in phase 1 of the project.

- Forms

Forms for data entry should be provided for all the different types of data in the database.

Forms are built using Django templates, and Django form generators. Each model can have an associated Form class, which can generate the HTML for a form automatically, based on the model's properties. If some properties need to remain hidden, you can exclude them from the form. The HTML form is then included inline in the HTML template at the appropriate place - the submit button and other form HTML must be included in the template.