ASN3 Technical Report

CS 373: Software Engineering
Fall 2010

Shanky Balani, Brian Burns, Jonathan Grimes, Ben Kornfuehrer, Sang Yun

1 Introduction

1.1 What is the problem?

The objective of this project is to create an anonymous social network that allows users to add information about certain items, such as classes, books, restaurants, etc. and rate and comment on them based on their experiences, and to view the aggregated scores of other users.

1.2 Why is it interesting and important?

This is an interesting problem because it demonstrates the different views and opinions students may have about a certain item. For example, student A believes that a book is very well written and organized and therefore gives it a 90/100 ranking. Student B, however, is not does not believe this same book is interesting at all, and does not like the way the author describes the settings, and therefore gives the book a 40/100 ranking. The application calculates an average rating and it is interesting to see a book that gets very high ratings by book reviewers get such low ratings by students, or vice versa.

This is important because it gives people a perspective on a certain item without ever having experienced it themselves. It's just like before buying a car or a television. Before you invest a lot of money, you look up reviews for that item and check the overall rating. Only once you see what other people have to say then will you make a decision whether to buy it or choose something else. In this scenario, it helps students select their courses, places to live, what internship to go for, etc. that serves them well in their best interest.

1.3 Why is it hard? (E.g., why do naive approaches fail?)

One of the main reasons websites like this are difficult to make is the issue of scalability. A site needs to be able to cope with varying levels of demand - for this site, access might increase around registration time as students look for classes to add to their schedule. That is one of the benefits of hosting a website on the cloud, like on Google AppEngine, because the

infrastructure is able to detect extra demand and replicate the application on more servers automatically. This is a much simpler and cheaper solution than providing your own server capacity to cope with the maximum anticipated demand.

Access to the application is limited based on the type of user one can be. They can either be an administrator, which will allow them to import/export an XML instance, run unit tests, completely remove a student from the database, or reset the whole system; a student, which will allow them to add classes, books, restaurants, recreational places, study places, residences, papers, internships, and games; or a browser, which will just allow them to view all the information that was entered by the students.

Trying to separate the users from each other was tricky. As mentioned, there are certain features enabled for administrators but not for students or browsers, and certain features enabled for students but not for browsers. So we had to make sure we had the correct features enabled depending on the type of user logged in.

Another reason why this problem was hard was because we had to make sure a student user did not have the authority to delete another student's entry. They are however able to edit the entry such as the rating and add their own review.

Now that we know a student can add classes, books, games, etc., we have to make sure that information entered into the database is real, legitimate information. To ensure this, we have to validate the form. This way, we are guaranteed that we are entering true data into the database. For example, the isbn for a book can be either 10 or 13 digits. The application will throw an error if letters or a character string is entered. On the same note, when recording the semester where a student lived, for example, the syntax should be a "Fall", "Spring", or "Summer" followed by the year.

The simple naive approaches wouldn't have worked just because we have to keep track of the session we are in. Everything works in a line. From the session, we can access the user. Accessing the user will grant us access to the student. From the association between the student and the attributes, we are able to access classes, books, etc. Just trying to get to the attributes directly will fail because there is never any way to link the user to those attributes.

1.4 What are the key components of our approach and results? Also include any specific limitations.

Google AppEngine encourages a Model-View-Controller pattern - the model is the datastore, the view is the Django HTML templates, and the controller is the handler classes. The model stores the data and provides methods for manipulating it, adding new items, deleting items, etc. The view handles the user interface - in this case, HTML templates which get filled with values through server-side script embedded in the HTML. The controller is the go-between between the view and the model, allowing the view to edit or add data, for instance.

1.5 Which team member is most responsible for each of the major components of the project?

Site design, CSS and JQuery - Jonathan Grimes
Validation - Sang Yun
Aggregation (average grades and ratings) - Brian Burns
Session management and Authentication - Jonathan Grimes
Accordion Sidebar - Ben Kornfuehrer, Shanky Balani
Profile Page - Ben Kornfuehrer, Shanky Balani
Graphics - Sang Yun
Data model - Brian Burns and Jonathan Grimes
Import - Ben, Shanky and Sang
Export - Jonathan Grimes and Brian Burns

2 Related Work

2.1 How does our solution compare to others?

Review each of the other **10** group GAE apps and identify the following:

- three things they did better
- three things you did better
- three bugs
- a rating **(1-10)** of the other group's app

2.1.1 Group 11-01

http://cs373-asn2-nobleteam.appspot.com/

Things we did better:

- Our website took advantage of CSS and is styled a little more cleanly
- Getting to the details of a class/book/paper is somewhat clunky for them. You have to select a checkbox then hit a button for details whereas we simply have links that go to the details section
- At the time of this writing it was difficult to tell whether I had successfully logged in or not (this may be a bug). Our site at least changes the links in the main navigation column to include a Profile link and a Sign Out link

Things they did better:

- You are able to filter out results and get what you actually would like to see. Currently our site just returns all records when listing classes/books/papers etc.
- You are able to view more than one object at a time, could be useful when comparing classes
- You are also able to comment and rate multiple objects at a time, this could also be

handy

Three bugs:

- When you get a new password, if you try and login instead of using the button, it fails and continues to keep you in a browser role
- You are able to add ratings when not signed
- You are not able to delete entities

Overall:

A solid effort, and good functionality, albeit implemented in a less than optimum way.

8/10

2.1.2 Group 11-02

http://cs373-asn.appspot.com/

Things we did better:

- The student ID's are a little ugly as they consist of letters numbers and symbols; while random they can still be difficult to remember. Our simply consist of numbers and letters
- When browsing it is difficult to tell if you are logged in
- Inconsistent interface when adding a rating to an entity, once I add a rating it moves me away from the browing interface and to the add interface; our interface is fairly fluid for all users, displaying add/edit/delete links only when applicable

Things they did better:

- Very simple, cleanly organized site, twitter-like feed of latest added entities
- Green and red rating bars make it easy to get an overall feel for the ratings of things very quickly
- You are not able to add ratings unless you are signed in

Three bugs:

- Received a crash when trying to enter in an internship with valid information; appears to have a problem in authorization check
- Major problems adding any type of entity, I am supposedly signed in but having no luck in adding any type of entity
- I'll revisit this site later; difficult to tell if they have bugs cause not much works

Overall:

Initially reviewed this site and encountered some errors when adding some things. Went back a week later and still ran into the same issues. Good design idea and UI but poor execution.

6/10

2.1.3 Group 11-03

http://cs373-asn2-anarseyf.appspot.com/

Things we did better:

- We make it easier for users to rate or comment on existing items, while browsing them. Their site makes you enter in all the information manually.

- We show the average grade for courses, which was part of the original spec.
- We let the user sort and filter the table data, which makes it easier to find an item you're looking for.

Things they did better:

- They use Ajax to get reviews of items clicked on, displayed below the main table - works nicely.
- When you delete your review of an item, it replaces the review with a block saying deleted, then animates the removal of that block. Pretty cool.
- They have an animation when paging through a list, with the table sweeping out to the left and a new one coming in from the right.

Three bugs:

- Site was crashing when clicking on Browse, Login, or Signup from the Home page: BadKeyError: Invalid string key ahNjczM3My1hc24yLWFuYXJzZXlmciMLEhtfQXB wRW5naW5lVXRpbGl0aWVzX1Nlc3Npb24Y. Notified the group through Blackboard and they fixed it.
- Login to site, goto Update, logout, then hit Back, goto Browse, and site crashes - BadKeyError: Invalid string key ahNjczM3My1hc24yLWFuYXJzZXlmciMLEhtfQXBwRW5naW5lVXRpbGl0aWVzX1Nlc3 Npb24Y. Then can't access site at all - need to clear cookies.
- Not quite a bug, but they don't list the place type in the table of places, which makes it hard to find something you're looking for.

Overview: Some nice Ajax work, though a little slow for such a small amount of data. And what is up with that creepy troll face?

Overall grade: 9

2.1.4 Group 11-04

http://cs373-asn-asmith89.appspot.com/

Things we did better:

- Got average rating working for books and classes (and average grades). I tried added a class with exact values of an existing class, just changing the rating, but it didn't get an average rating for the class - just stored and displayed them as separate records.
- Our site is faster - if you click on the Class tab while browsing, it sometimes takes a long time for it to display, so long that you might think there is no data in their database.
- We have paging, sort, and search for our tables, thanks to the DataTables plugin.
- We had a lot more unit tests - they only have 9, we have 47.

Things they did better:

- They have some nice javascript to display information in a popup window - eg if you click on a class, it shows an hourglass window, grays out the main window, and then shows a small information window.
- I like the styling on their site and their tables better - the grayscale is nice, and their tables have a nice hover effect of highlighting the row the mouse is over.
- They have nice instant validation on properties as you enter data, with visual feedback, along with hints on what valid information should be.

- They have an awesome splash page image of Glenn Downing as Superman

Three bugs:

- My papers didn't get imported - maybe the giant comment I had in a book (>512 chars) crashed the import, since papers come after books. The two books before this huge comment made it in though. My games didn't get imported either, since games come after books in the xml file.
- Says "Philip K. Dick" is invalid for author because only letters are allowed - wouldn't let you put in a period.
- Added a book 'foobar', said it was successfully added, but then it showed up as 'Introduction to Probability'. Oh, it was using the ISBN to decide what book I was adding, and the ISBN 1234567890 was already in the database (several times), and the first one was the probability book. So not quite a bug.

Overview: A good effort, though often very slow, and not as polished looking and acting as some other sites.

Overall grade: 8.5

2.1.5 Group 11-05

http://cs373-asn2-pros.appspot.com/

Things we did better:

- When adding a class, it redirects right back to the add form, which can be confusing. There is a flash for a succesfully created class, but our site upon creation of an object redirects you to the view page for that object, where you are able to rate the object.
- When creating a new login ID for the site, it does not automatically log you in and you must write down the ID and password and then go to the login form. Our site, automatically logs you in upon creation of a new ID.
- Each object in the database has only one rating and one comment. Our data model allows you to add mutliple ratings and comments to each object.

Things they did better:

- They made good use of AJAX calls to make a lot of the UI seamless, most notably, they have client side validation going on in the add page for objects which means that the user is aware that a field is incorrect before submitting the form.
- Slickly designed, easy on the eyes and the site degrades gracefully from using a big screen to a small one.

Three bugs:

- As with a lot of previous sites, if you logout and then use the back button of the browser, you are able to modify objects while being logged out. This is a common, but egregious security flaw.
- When adding a location, you can't use punctuation when filling in the location field so "Austin, TX", a common address idiom, is not acceptable input. Additionally, numbers are not permitted either, so the string "5th St and Congress" is not valid input when it should be.
- When adding a fun place, I tried to add a large comment and wass receiving no error messages that the comment was too long, even though the validation was stating that

everything was valid.

Overall:

I felt like they should have done some of the user interface better and while their use of AJAX was good, some of it felt like it was just tacked on for the sake of using AJAX instead of using it to solve a real problem.

7/10

2.1.6 Group 11-06

http://asn2-bubu.appspot.com/

Things we did better:

- While their site is pleasantly designed, they suffer from a couple UI flaws, such as when you are signed out it is easy to find the browse link to browse classes, but when you are signed you are only able to view objects relevant to you and not any other users information.
- The initial load time for things such as browsing through classes, is a good 5 - 6 seconds of load time, which while not too bad, it is noticeable enough to annoy the user.
- Again, from a UI perspective, there are some weird idiosyncrasies that are confusing to user. When adding a class you have the choice to either add a completely new class or add a rating to an existing class. But once you rate the class, you only see your rating and comment, not any of the other users information. Addtionally, the form in which you add this information is poorly designed, with essentially two forms, but not a clear delineation of what both of them do.

Things they did better:

- Not much

Three bugs:

- The add links from the list pages while signed in do not work. You can reach them from the main page, just not from separate objects listing pages.
- Their site also seems to have failed to import our book with the huge comment.

Overall grade: 8

2.1.7 Group 1-01

http://cs373-asn2-timmy.appspot.com/

Things we did better:

- Our view page for browsing through each element is much easier to look at since we allow the user to change how many items per page he/she would like to see at a time. This app prints out all the items in one page, making the user to scroll down a lot and makes it easier for him/her to get lost.
- We provide an option to sort the items according to user's specified preference, while this app does not seem to have any sorting mechanism.
- If an invalid input was given to add to the database, our app tells the user what they have done wrong in a little more detail than just "This field is invalid" like this app. Our app tells the user if he/she left a required field blank, or entered information in an invalid

form (also provides the correct format and examples).

Things they did better:

- Looks very clean, compact sized interface for easier focus (excluding the list of items).

Three bugs:

- This app does not tell the user what went wrong when log in fails but just returns to the log in page without any error message.
- If you log in to this app, you cannot browse through items other than yours. In which you would have to log out to go back to browse mode. Our app allows the user to also have all the access that normal browser would have.

Overall grade: 9

2.1.8 Group 1-03

http://cs373-asn2-a-team.appspot.com/

Things we did better:

- Our app's data retrieval speed is much more faster than this team's app. This app take pretty good deal amount of time to list a large information (i.e.: classes).
- This app does not notify the users when they have not field out required field. For example, when all fields were left blank while adding class it just returns back to the add page without any error messages.
- While this app is supposed to be anonymous, it asks for user's email address to register.

Things they did better:

- This app prevents the user from adding a duplicate item to the database.
- This app tells the user which fields are required at the add page, while our app tells the user as an error message while they are adding.

Three bugs:

- When invalid semester is entered, it does not provide the user what was wrong with it. All it says in red is "semester" whereas other attributes tells the user what went wrong. (Same thing happens for "Unique")
- All fields are cleared when an error occurs while adding, even the correctly formatted ones.

Overall grade: 9

2.1.9 Group 1-04

http://cs373-asn2-p6.appspot.com/

Things we did better:

- We show the data in a much more concise way. I feel that their pages are entirely too long.
- Logging in is a bit clunky. Because the id and password are random, I feel it is difficult to type them back in.
- Their site does not return you to a view of the category you just added after you add. You are instead stuck in a successfully added page forcing you to click back to where you want to go.

Things they did better:

- I like their drop down menus for browsing and adding in the upper right corner.
- They have a visual rating bar. It is a little more aesthetically pleasing to the eye when browsing
- I feel adding is a bit easier because you can go straight to the add book page from anywhere whether than viewing the list of books first.

Three bugs:

- If you hit logout and then hit back, when you try to add something you get an index out of range error because the student object is gone but you are still trying to fetch said student.
- You can add javascript from the comments
- When you logout and go back, the bar at the top still looks as though you are logged in.

Overall grade: 8

2.1.10 Group 1-05

http://cs373-asn2.appspot.com/

Things we did better:

- Our splash page had was a bit more dynamic and appealing. Their site did not have much to it without the tables to display.
- We allowed for multiple comments on one instance where as they had everything stored individually
- Our tables were shown in a less cluttered manner without a bunch of clickable buttons everywhere.

Things they did better:

- Allowed editing of multiple items at once
- Allowed multiple deletes at once
- Switching between categories was extremely fast with the links right above the table.

Three bugs:

- When adding a place, it was parsing the values incorrectly resulting in two location values in both location and semester. The rest of the table was then pushed to the right and extended beyond the table headers
- It seems that the database was reset for each new user. I created three accounts and added classes with each and they did not show up in the other's view.
- When you change you password you can no longer browse categories.

Overall grade: 8

2.2 Other websites

Another existing website with similar functionality is myedu.com (which used to be pickaprof.com). It also lets you rate classes and professors, but also displays information in graphs and charts. Considering that we developed this site in a few weeks, ours is a pretty good start at a similar commercial website, or at least a prototype for one.

3 Design

3.1 What were our design decisions?

In developing the data model, we had a choice to make a relatively flat model, reflecting the XML schema, or a more relational model, with association classes between students and the items they were rating. The former choice would have been simpler to develop in phase 1, and would have worked for importing and exporting the items, but we knew that in later phases we would want the association classes, so that students could rate existing items. So we used a more relational model.

To implement the average ratings for items, we could have either calculated them on the fly, each time an item was displayed, or store the average rating with the item. In a normalized relational database, you could use a GROUP BY clause in a SQL query to get the average rating. But for performance reasons, sometimes it is preferable to denormalize the database, and store the information more than once. Because we knew we would want the site to be scalable, we chose to store the average rating information in the item being rated, and update it on adding, editing, or deleting a rating for the item. That way, there was no need to do a join (manually in the case of AppEngine, since it doesn't have JOINs in the queries) to get this information, which would have slowed the site down.

One design decision we made in phase one was to use copy-paste to extend the code, instead of inheritance. At the time it seemed like the simplest thing to do, and we said that we would refactor it in phase 2. However, that never happened, and it never happened in phase 3 either. As a result, making changes to basic functionality required making the same change in 5 other places (there were a total of 6 basic types: Classes, Books, Papers, Internships, Places, and Games). This included 4 or 5 HTML template files for each type. This really slowed things down towards the end, and made it hard to be flexible. It would have been better, overall, to have used inheritance before ever copying and pasting, as once the copying had been done, it then became a barrier to implementing the inheritance.

3.2 What special features did we add?

We added a section on the right side of the web page which displays the 5 most recently added/ edited information. This information remains throughout the page and changes based on the information added.

For the import, we added a semi-fuzzy search in order to find existing items and attach the

ratings and comments to those items, instead of always creating new ones. For classes, this does an exact match on the course number, and a partial match on the instructor (eg ‘Downing’ will match ‘Glen Downing’). For books, it does an exact match on title only, ignoring the other properties. This was a bit of a compromise made in order to have some aggregation done on the XML data from the other teams. As it turned out, some people had course numbers like ‘C S 373’ and ‘CS373’, which we hadn’t anticipated. Anyway, if this were a real app, we would have modified the XML schema to follow our data model more closely, and allow ratings to attach to the correct items through unique identifiers. But in phase 4, we could use an extension of the fuzzy search to prevent users from adding duplicate items.

Another feature we added to the lists of items is the ability to sort, search and page through tables. This means, if the list of items is too large, we can display the list with a certain number of items per page. The search box also acts as an instant, case-insensitive filter on all properties in a table.

We’ve also added the feature to allow the user to change their password if they decided to. This allows for better security or in case a user is just not happy with their password.

3.3 Roles: admins, students, browsers

We had to consider the different types of users mainly because each user has different levels of access. The administrator can add/edit/delete a student and all their information. They are also allowed to import/export XML instances which will automatically fill the database. The administrator also has the right to delete the database entirely and start all over again.

A student can add different classes, internships, games, papers, etc. Students are also allowed to edit their information, however they are only limited to editing the rating and comments of other student’s entries. This prevents from one student completely changing another students entry.

A browser is only allowed to view the database. They are not able to add/edit/delete anything.

To maintain this distinction, we remember the type of user in the session. By default, when someone visits the application without logging in or signing up they are a browser. So immediately add/edit/delete functionalities are kept hidden. Once they sign up, they are recorded as a student, because there is no reason an administrator should have to request credentials to access their own webpage. And if they have already signed up, that information is maintained in the database, and therefore only has to go through that process one time.

3.4 UI

We wanted to make this application as user friendly as possible. It took us a while to figure out

exactly what we wanted the layout to be. We started out with having some links on the left side of the page which provided some basic functionalities, such as signing up for credentials for a first time user; login for an existing user; import/export XML instances; view added items such as books, internships, restaurants, etc; and finally the ability to delete a user.

As we made more progress, we realized we weren't really separating the features of different users. For example, we had allowed the import/import XML for a student when it should only be for an administrator. After adding that restriction, the application made a lot more sense. Another major change that we made was allowing whoever the user is: admin, student or browser to be able to access the list of added items regardless of the page they were currently browsing, by adding a menu bar across the top of every page.

Our UI slightly mimics that of Facebook with the header and title of the webpage. We have three menus that are constant throughout all the pages of the website. One of them is a menu that has the following options: Home, Login, Sign Up, About, and Help. When someone logs in, the Log In option is changed to Log out. The reason why we have this menu constant throughout the page is because people may need help through out the way, or they would like to see information about the creators of the page. Therefore we decided to keep this information available and visible wherever they are on the page.

Another menu that we keep constant through out the pages is links to our items. So on the top right of the page, we have the following links: Classes, Books, Papers, Internships, Places, Games. When clicking anyone of these links, any user will be able to view the whole list of whatever item was clicked, and depending on they type of user, they will be able to edit/delete and entry or just view it.

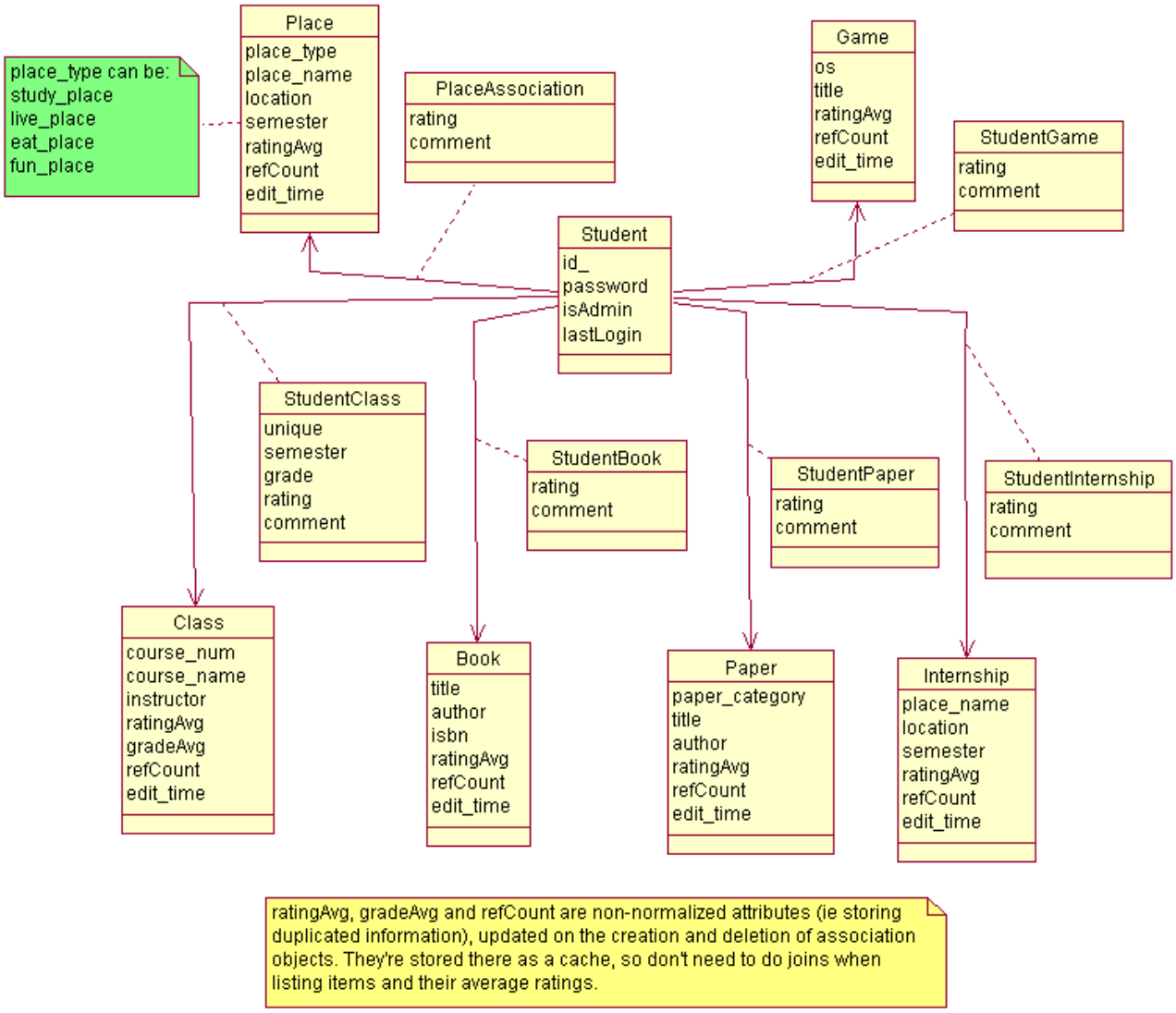
One last thing we decided to leave constant throughout the webpage is our list of most recently added/edited items (Classes, Internships, etc.) The reason why we even decided to have this feature is so that other users can see the most recent places somebody ate, or where somebody lived, just to give them a feel that people still eat/live there.

The way we have our layout allows the user to see everything clearly and therefore not letting them spend more than 10 seconds looking for something. If necessary, we take multiple steps to accomplish a goal so that the user can see exactly what's happening. For example, when adding anything, let's say a book, the user will enter its corresponding information. After hitting submit, the user is then redirected to another form where they rate the book and leave any comments if any.

3.5 Data Model / UML

The data model is basically a list of Students, Classes, Books, Papers, etc, with association classes between Students and each item type. The association classes are necessary because the relationships store information - ratings and comments, and grades, in the case of Classes.

Each item also caches the average rating and the number of ratings, in order to be able to display that information quickly in a table. This information is updated when an association object is added, edited, or deleted.



3.6 XML

The XML instance and schema are unchanged from previous phases, except that we adjusted the ratings so that they were all on a scale from 0-100, and removed some duplicate items.

4 Implementation

4.1 What is our solution?

Our solution was to build a website using Google AppEngine, which provides automatic scalability and free server hosting, up to a certain amount of usage.

4.2 Frameworks and tools

Our implementation is based on the **Google AppEngine** platform and written in **Python**, relying heavily on the included libraries such as the **Django Forms**, **Django Templates**, and **Google Datastore**, but also some other third party libraries such as **jQuery** for user interface functionality.
The jQuery **DataTables** plugin was used to provide sort, filter, search, and paging functionality for the tables. [accordion, tab plugins]
GAEUnit was used as the basis for the unit tests, while **Twill** was used as the basis for functional tests.
xmllint was used for XML validation.
Google Code was used for issue tracking.
Git was used for our version control system.
Assembla was used for hosting our Git repository.
GNU make was used to store and run frequently used commands.

From an architecture perspective, we used the Model-View-Controller design pattern which guided us in both a design philosophy as well as helping us layout our project structure.

4.3 Authentication and session management

From the very outset, we realized we were going to need some sort of session management to maintain the state of whether a user is signed in. Since, the web is a stateless set of protocols, we had to have something that would work with the browsers cookie storage. To that end, we used the simple Session class that was found in the Programming Google AppEngine book written by Dan Sanderson.

This class is simply a collection of methods that allows us to keep a session state from request to request by storing the data in the Memcache facility inherent to Google AppEngine, and keeping a key to the data in a browser cookie. When the Session class is instantiated, it grabs the key from the cookie and accesses the data in Memcache. The class wraps the standard Python dictionary and simply provides a little more backend magic to it, but also has the same dictionary features such as looking up keys, and being able to add keys and values dynamically.

4.4 Datastore

The datastore is implemented in one file, models.py. It defines classes for each type in the UML diagram. Each class inherits from db.Model, and adds the relevant properties, eg:

```
class Book(db.Model):
    title = db.StringProperty()
    author = db.StringProperty()
    isbn = db.StringProperty(validator=validate_isbn)
    ratingAvg = db.IntegerProperty() # 0 to 100
    refCount = db.IntegerProperty()
    edit_time = db.DateTimeProperty(auto_now=True)
```

We override put() so we can catch any missing required fields - the validation framework can handle this also, but by the time we started to add validation, we already had a lot of code that created empty objects and then set properties on them, and creating an empty would throw a validation exception.

```
def put(self) :
    if not self.isbn:
        raise db.BadValueError("ISBN is a required field.")
    else:
        db.Model.put(self) # call the superclass
```

We add a static method to get the most recent 5 items for the type:

```
    @staticmethod
    def get_by_date(limit = 5):
        q = db.Query(Book).order("-edit_time")
        results = q.fetch(limit)
        return results
```

The association classes are similar:

```
class StudentBook(db.Model):
    student = db.ReferenceProperty(Student)
    book = db.ReferenceProperty(Book)
    rating = db.StringProperty(validator=validate_rating)
    comment = db.TextProperty()
```

We override put() and delete(), where we check for missing values, calculate the average rating and reference count, then store them in the associated object:

```
def put(self):
    if not self.rating:
        raise db.BadValueError("Rating is a required field.")
    db.Model.put(self) # call superclass
    self.getAvg() # update averages
def delete(self):
    db.Model.delete(self) # call superclass
    self.getAvg() # update averages
def getAvg(self):
    book = self.book
    links = book.studentbook_set
```

```
ratings = [int(link.rating) for link in links]
n = len(ratings)
ratingAvg = None if n==0 else sum(ratings) / n
book.ratingAvg = ratingAvg
book.refCount = n
book.put()
```

4.5 Forms

For the entirety of the site, we used Django forms to aid us in both writing the forms as well as providing a great deal of plumbing to facilitate our ability parse incoming data and get it validated. Essentially, every model class has a corresponding DjangoForm class that wraps around the model class and provides methods to display an HTML form for data entry. Additionally, as well as being easy to implement from a GET request perspective, the DjangoForm class also has great deal of functionality that allows us to parse incoming POST requests, run validation on the model, and provide error notifications if validations fail. Furthermore, because we did not implement manual HTML forms, if our data model should hypothetically change in the future, there is far less code to change as the class wraps all that functionality so effectively.

4.6 Validation

To validate the data for the database, we used a built in mechanism in the Google App Enigine. When we were declaring all the attributes and their properties of each entities, we passed in a corresponding validation function for each. Validation functions we wrote for each attributes takes in a data and compare it to a regular expression we specify. If the data being entered is invalid, it throws BadValueError exception which gets caught by the program and provides detailed message to the user.

When creating the new instance of an entity, we had some problems making certain attributes be required on the constructor level. So what we did was to override the put() method given by Google db to check if any required attributes are left blank during the insertion to the database rather than creation of an entity. This turned out to be very useful for us in that, when an error occurs during inserting a data to the database, we were now able to distinguish if the error was that the data is in a invalid form or it was left blank. This allowed us to give the user a detailed message rather than just outputing “Invalid input” to every error.

4.7 Import/Export

Import and Export are as described in phase 2 of this project. The only addition made was to having semi-fuzzy matching to find existing items to attach ratings to, rather than always creating new items, in order to accommodate the incoming XML data from other teams. This was accomplished by adding a static method to each type that looked like this (comments removed) -

```
@staticmethod
def findAdd(course_num, course_name='', instructor=''):
```

```
q = Class.all()
q.filter("course_num = ", course_num)
results = []
for c in q:
    if (c.instructor in instructor) or (instructor in c.instructor):
                results.append(c)
if results:
    c = results[0]
else:
    c = Class()
    c.course_num = course_num
    c.course_name = course_name
    c.instructor = instructor
    c.put()
return c
```

4.8 Files and directory structure

We tried to organize the files a bit more like Ruby on Rails, with models, views, and controllers in separate directories. We had trouble putting models into a separate folder though, so it wound up stuck in the main directory.

The directory structure:

handlers/	request handler classes, eg BookHandler.py
static/	css, images, jquery javascript, jquery plugins
test/	GAEUnit test files, eg testValidate.py
twill/	twill test scripts, eg testBook.twill
utils/	authenticate, session, dorender, import/export (Python files)
views/	Django HTML templates, eg profile.html, book/list.html
app.yaml	application definition, main url redirects
models.py	the data model classes

5 Evaluation

5.1 How did we test our solution?

We mostly used manual testing to test the user interface, though we attempted to get Twill working for some functional tests. For the backend, we have a number of unit tests.

5.1.1 Unit Tests

Unit Tests were written using the GAEUnit module. Import and Export tests were leftover from phase 1 of the project.

section	number of tests	notes

import	18	
export	11	
import+export cycle	1	does an import, then export, then imports the export and exports that, and compares the two exports for exact string matches - a good test of import, export, and the entire data model.
find/add	6	on import, does a semi-fuzzy search for pre-existing items
models	2	insufficiently tested (though the import+export cycle effectively tests all the models and properties)
validation	6	each test has several assertions
grades	4	each test has several assertions

5.1.2 Performance Tests

We wanted to make sure the site would be fast even with all of the data loaded, so we managed to get the huge XML instance from phase 1 cleaned up and importable. When running the site locally on an old computer (a 6 year old Pentium M 1.2GHz) with the full dataset, it seemed to really bog down, but running the site on Google's servers was much faster. As it is, our site seems to be faster than most of the other sites.

For a real site, you would want to do load testing, hitting the server with a bunch of requests and database operations at around the same time, to see how it handled it. We didn't do this for our site.

5.1.3 Groupware Tests

We didn't do any groupware tests, like trying to do something at the same time to the same record.

5.1.4 Functional / UI Tests

We used Twill experimentally to perform some functional testing of the website. With Twill you can interact with a site at the command line, following links, filling out forms and submitting them, checking the resulting webpage for the presence of strings or regular expressions. Then you can save your interactions to a script, and run the script automatically - Twill will then alert you to any failures.

We ran into problems being able to delete items that were added in the Twill script. But you can also run Twill through Python, which would enable you to have more control over finding a certain link to follow (for instance, to delete an item you just added). This would let your functional tests leave the database in the same state that it started in.

A better approach though might be to use a test database, which could be possible by running the local appengine by pointing it to a new or existing test datastore file. But this would require restarting the server each time you wanted to run the functional tests. Or, perhaps you could have one console running the app on port 8080 with the regular datastore, for browser testing, and one console running the app on another port, for Twill testing.

6 Conclusion

6.1 Why is our solution the best?

- Speed - our site is relatively fast compared to some others, which implies that it would be scalable. Though there are some issues related to that in section 6.3.
- We decided to do the aggregation for classes on course+instructor, moving unique# and semester to the association class. This lets prospective students compare how two instructors might be at teaching a certain class.
- Ease of use
- Familiarity of interface (mimicking Facebook)
- Density of information (everything is presented fairly close together)

6.2 Outstanding issues

There are a number of issues that got pushed off to the mythical phase 4 of the project - there are 24 in our issue tracker. If we ever need to use this app for our resumes, we might end up working on phase 4. Some of the issues include the following:

- Profile page is too wide for computers with smaller monitor resolutions.
- Editing a rating or comment and leaving them blank will set their value to None, instead of an empty string.
- If a user clicks “back” right after logging out, the page still says “currently logged in as : <username> “. Even though we have it so that the user cannot modify anything in that stage, personal information might be viewed. This causes serious problem if the previously logged in user was an admin.
- Our tables have the ability to be dynamically sorted by each attributes, however how our table is presented does not tell the user that we have that option. We need to modify the interface so that we clearly tells the user about that ability (e.g: mouse-over cursor change, arrow marks next to each attribute on the table, etc)

6.3 Future improvements

The main improvements needed in the future would be for scalability, if this website were to actually be useful. And codewise, we would need to do some refactoring, to make it easier to maintain and extend.

- Need to implement a scheme to check for a user adding duplicate items - could do a fuzzy search on existing items and show them to the user with Ajax, before letting them add an item (like Stack Overflow does when you try to add a question).
- Make the accordion use a queue instead of performing queries with each page load (with sorts on edit_time field). ie as items are added to the datastore, add them to a queue of 5 recent items for each type. That way, no sorting need to be done by the backend.
- Use Ajax for DataTables plugin - it can pull data from server as needed instead of getting it all at once, and is quite fast. Right now it just pulls all the data, and the plugin chops it into pages, but the plugin can also use Ajax to pull just the data that it needs. See the example at http://www.datatables.net/examples/data_sources/server_side.html
- Refactor the copy-paste code, and use inheritance instead. Currently there is a lot of duplicate code, which is hard to maintain and extend.

6.4 What did we learn?

- jQuery is awesome - the DataTables plugin, in particular, is great - just a couple of lines in the HTML template and you have a paging, searchable, filterable, sortable table.
- I think we all finally figured out how to use Git properly, making local branches, etc. We were still having occasional problems with using it properly even into this phase of the project - if you don't know what you're doing, you can really mess things up.
- Using decorators for authentication - this was a nice practical application of the decorators we learned in class, and they came just at the right time. It saved having to copy and paste a lot of code.
- Copy-pasting code is bad - it's much better to do the abstraction up front. We kept saying we would refactor it, but we never did. In part, the very existence of all that copy-paste code formed a barrier to doing the abstraction. But it slowed us down when we most needed speed, in the day or two before phase 2 was due.
- Using Twill for website testing - with a bit of work, we might have been able to get this working. But as it was, our website functionality kept changing up until the last minute of phase 2, so having all these functional tests might have been a bit of a hindrance - they would have become obsolete anyway.
- Which leads to... do site design up front, using wireframes and flow diagrams. We did not do this, and instead just evolved the interface as we added code. As a result, the site is probably not as user-friendly as it could be.
- Who needs relational databases? Google AppEngine uses a key-value store for scalability, not a relational database, and it suited our needs just fine. GQL does not allow JOINs, but when absolutely necessary, you can do joins through code, by following references to other objects. This is almost as easy to do as building SQL queries

through code, anyway, and is perhaps conceptually clearer.

- You don't need to be tied to AppEngine - you can abstract all the data and UI aspects, and host the same app on Amazon or other Cloud services, or build your app using Rails, etc - see the AppScale and Active Cloud DB projects.
- Making websites is not so bad - five people made an almost functional website over a few weeks. If you have a good idea, you could start a business pretty easily, with no startup costs.

7 References

[AppScale] http://code.google.com/p/appscale/
[Active Cloud DB] http://code.google.com/p/appscale/wiki/Active_Cloud_DB