Minh Nguyen

CS 496: 'Take Home' Final Exam

Overview and Performance

My database is designed for teacher's to log in and add classes they teach into their accounts, and add students in and out of those classes. On the front end, user's are allowed to add classes and students to these classes to their accounts. However, there is no administrative attribute in the front end for user's to add classes and students to the database as an entity. There will be cURL calls from the administration side to add classes and students into the database. Classes have a relationship with students by having a student array property. User's have a relationship with classes by having a class-teach array property in the their entity. Setting up relationships in this fashion greatly enhances performance of the data stores over relational database queries. Queries only occur when students are added to a class by first checking if that student exists, likewise with classes to users. Since my app is very small and only returns JSON responses, it makes minimal trips to the data store. There is no need for image caching, or any expensive computation over the server. There are only simple queries of the 4 verbs, with GET being the most verbose one as it queries through the entire store for a given item.

My implementation of authorization and authentication was quite simple. The user entity has a username, password, and a classes array. When creating a new user, a username and password is needed to create a new user. I implement this on the front end by throwing an error if any of those fields are empty during creation. On the back-end, there is no requirement for it being non-empty which means an administrator can go in and enter in an empty name. When a user is created, a new User entity is then added to the database. In order to log in, a user must provide their username and the matching password that goes with this entity. When a user types in a username and password, the button takes the username and sends it to the database with a GET method with this name. For instance, a user named John would send this GET request to the data store http://final496minh.appspot.com/user/John. The front end will retrieve the response from the server and parses the information to see if the username and password that the user supplied matches the JSON response from the server since the server shoots back a name and password as well. If this is a match, then authentication is complete and users may log in. Once logged in, the username variable is passed into the next intent (page) of the android app, and this is their key for authorization. Using this

username to make more PUT and DELETE requests by building URI strings necessary to add classes to their accounts.

Portability and Reliability

How reliable the API is depends on how reliable the hardware is at Google. Since I'm running my application on Google App Engine, my API is pretty reliable. Using the NDB data store, my back end can have better scalability and increased reliability due to the replication of data across data centers. I get around with the eventual consistency problem of the data store by enforcing strong consistency using parent keys and creating ancestors of these parent keys when creating new entities. When a new ancestor entity is created, it is inputted immediately into the database and stores it under it's parent entity. Since my application is very small, there is not much reliability issues as it requires very little memory/data which makes for a lesser likelihood of failure.

My API is for the most part, a REST APi. RESTful API's are very portable since nearly any platform can connect to it and expect a response. So far in this class I've tested my API using a browser, cURL, and my native android application. They all work extremely well and that is due to the fact that an HTTP request is all that is needed to talk to my API. From this assignment, I understand now the true power of RESTful API's and why many developers out there go with REST API designs. The extreme portability allows for access across many different platforms that exist today. Portability is a very big factor when deciding to change code or refactor a part of the program. From my application, I can change entities and properties at will and it will ultimately not affect how front end users use my API. Translating this idea into larger API's where there are a thousand times more data, it is extremely crucial that any changes in the back-end will not force the front-end to be changed, or else this would cause a very big problems for front-end developers and users.

Security

Security in my app is close to none. Since I implemented authentication and authorization by hand using entities and properties of the NDB data store, it is easily accessible to the public. However, I do provide a positive security model in the sense that users cannot do anything to their accounts until they are logged in. This prevents users accessing other users accounts. I keep security simple by allowing only one screen for logging in, and the other screen for displaying information. For input

validation, I did not implement any sort of string checker to make sure that characters inputted are valid. When creating a new account on the mobile device, users must enter in a username , a password, and password again to confirm. Both passwords are not hidden for demonstration purposes so this also takes away from security. If a field is empty, then submitting the data will fail. This is also done on the front-end with java code. I also check the 2 password inputs to make sure that they are the same values. Once username, password, and confirm password are filled in properly, then the enter data button will successfully execute the request.

When attempting to log in, both fields of username and password must be provided. The app checks the input from the user and compares it to the data store for a match. This feature is not that safe since anyone can view the data on the cloud data store. No passwords are ever stored on the source code itself however. Since my API is quite open to the public, anyone can make POST and GET requests to the database as long as they have the URL's necessary for it.

Security was not my primary concern when creating my application. Since this project is for learning the cloud and to manipulate it with a mobile front-end, something I've never worked with before, I did not enforce very much security which allows me to focus more on developing the other areas of my project. However, if I were to be working on a project or a job that requires security, I would definitely not do it the way that I've done for this assignment.

Another layer of security I provide is that users cannot do much more other than create an account and add classes to their account. Users do not have the power to add new classes to the data base, nor can they add students to the database. This limit of authorization is a security feature for users to not be able to manipulate data in the database that they have no need for.

Interoperability

My cloud back-end is very interoperable in the sense that it can communicate with many other platforms by it's nature of being a REST API. It responds with JSON and as long as the receiving end knows how to parse JSON data, it is very easy to talk to my back-end. With the back-end having high interoperability, I can create front-ends as a mobile application, a web-browser, or even cURL commands on the command line.

In my application, I developed the mobile app on android. To communicate with the back-end system on android, there are HTTP request libraries allowing for access to the internet and URL's. This is how my application makes a connection to the back-end and receive responses. Using HTTP client libraries, a response is captured after execution of the http request. This response is in JSON data and has a complex parsing sequence. Every platform has their own way to deal with JSON data, but the most important thing is that it can parse the JSON data. This is a method to make my mobile application interoperable and how it consumes data from the server.

Another key factor in android applications is the use of intents. My main intent, which is the first page that opens up is the login page. After login credentials are correctly supplied, the login button activates another intent which is the display of information page. The program must be interoperable with itself and intents take care of this by allowing the passing of variables from one activity to another. It is passed using a key-value pair. One intent will pass a key and a value, say key:username, value:John. The new intent can then create a new variable called user and the value will be set equal to the value of the key username from the previous intent. The use of intents allow for many different applications and systems to come together in android. Intents can open up a web browser, a game, email, a phonecall... etc. This allows for very high interoperability within the system to talk to itself.

Usability

My mobile application is highly usable. Since it's a very small demonstration of a mobile app, there are only 2 pages, the login and the display/insert information pages. Buttons are labeled clearly for users to see what each button will do. And it all fits inside the screen so users can understand the app clearly at a glance.

Although atheistically lacking, my intention for this web application was more of a way for me to demonstrate the communication between the front and back end. This could have been addressed by making custom buttons more nice or a better background. However, I do introduce the native mobile feature of sound clicks to my buttons that sound differently when the button is clicked depending on which operation is being done. This pertains to mostly a successful operation or a failing operation. A successful operation will have a positive sound to it that lets the user know that the request was successful. A negative sound will let the user know that the request was unsuccessful and must retry it again.

Another feature I added is the Toast feature in android. A toast is a quick message that appears for a short duration. In my application, nearly every button click has a toast associated with it. This combined with the button sound feedback helps improve usability. User's will hear a negative sound as well as a toast that says their request failed and why it failed. This allows users to change their inputs in order to successfully complete their action. When a request does succeed, a positive Toast appears saying that the request or action was done successfully, giving feedback to the user.

Since the size of my application is quite small, it is easy for users to quickly pick up and learn how to use it. This was my main goal when creating this application as I'd like for it to be easy for instructors to follow along with the demo and see how all of my requests work. It's clear and concise with hardly any overhead so that it does not take away from the main purpose of the assignment.