Project 6

DropBucket

Thomas Alder
Matthew Donovan
Ryan Loi

Final State

The final state of our project is composed of a QT front-end and a RESTful Django API back-end that handles interactions with a GCP storage bucket. In the submitted version of Dropbucket we implemented the following features:

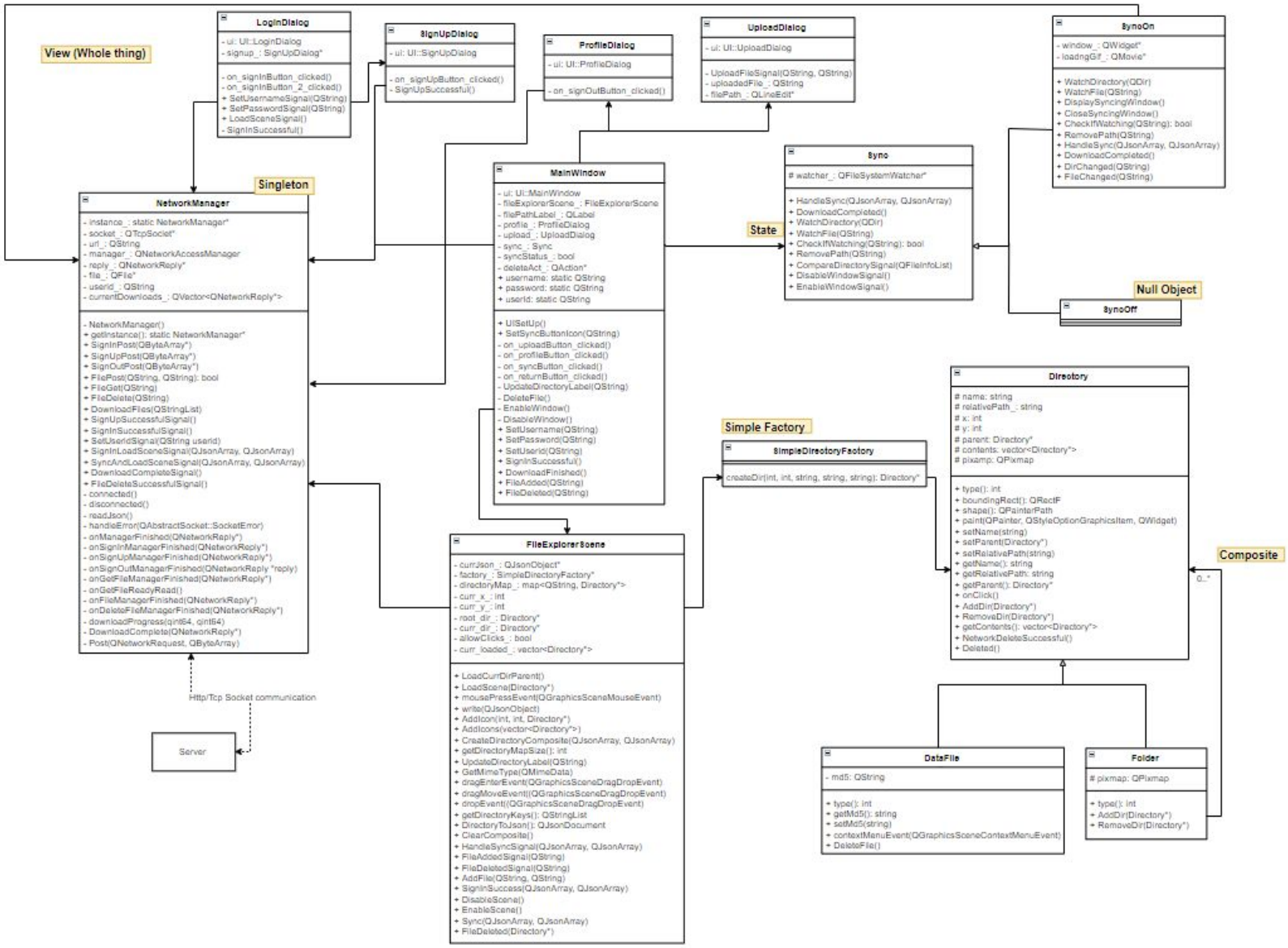
- Sign up/log in
- File uploading, downloading, and deleting
- Syncing with the remote storage across multiple devices

We were not able to implement the following features:

- Sign out
    - For sign out, while developing we wanted to concentrate more on signing in/up as well as the basic networking for the server. Due to this we simply neglected the sign out since we concentrated on the more complex interactions the user would accomplish whilst signed in.
- Bucket metadata information
    - We found that the parsing of the metadata and passing to the user added a functionality that was critical for basic user interaction with the client and server. This would be a cool add-on and wouldn't necessarily take long to implement but we chose to concentrate on the more complex
- Syncing when sync is turned on from off
    - When we came to developing turning the sync on we had already reached the point where the final project was close to being due. We would've needed to add an extra endpoint and handle it being hit and we simply ran out of time to implement this without the risk of accidentally messing up the entire server.
- Diffs between files so we don't reupload full files
    - This was the most ambitious of the features we wanted to implement. In the end, we didn't have the knowledge and couldn't come up with a methodology to accomplish this as well as the other features we wanted to implement. We concentrated on mainly networking in which we didn't have the most experience with so the learning curve took longer than anticipated. We look to have this feature implemented at some point but would most likely many more sprints to get this working effectively.

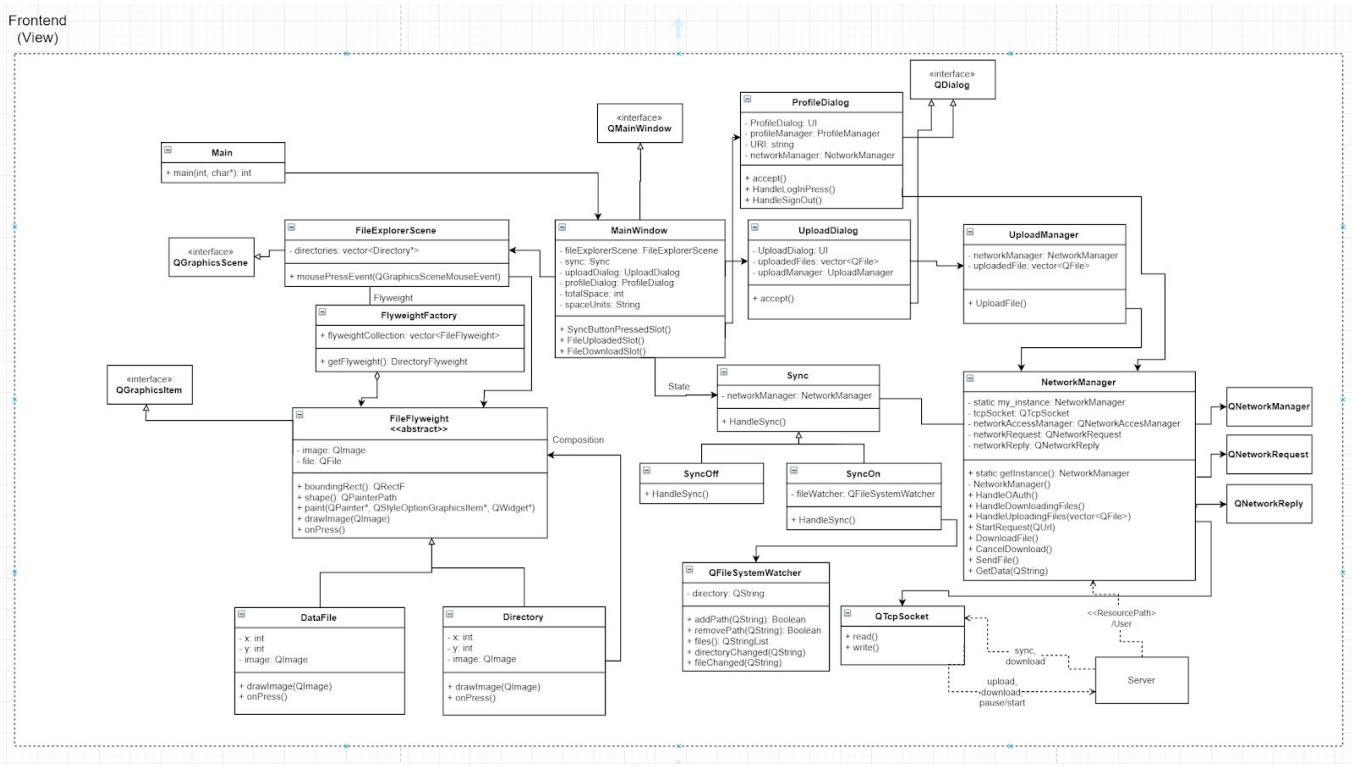
Frontend Diagram

Final class diagram



Apologies for the fields/methods being difficult to see, the full diagram can be seen here with a colorado.edu account.

Initial class diagram (from project 4)



Changes between diagrams

For the front end, the first major change between the two diagrams was the Flyweight pattern included for the DataFile and Directory. Initially, we had thought that this would be a good fit as each of these classes represent the icons rendered in the FileExplorerScene as the files or directories. As these icons did not turn out to be as graphically or memory intensive as first though we chose to eliminate usage of the Flyweight pattern and just have a Composite structure instead. In addition, each DataFile and Directory has a unique structure and data so reuse wasn't feasible when it came to deletion of a DataFile or Directory as this would cause a deletion of the entire system.

In addition, we removed a UploadManager from the diagram and instead had the NetworkManager handle the uploads since these came in the form of Http POST requests which the NetworkManger has already handled POSTs for sign in/up and UploadManager only increased the number of necessary classes.

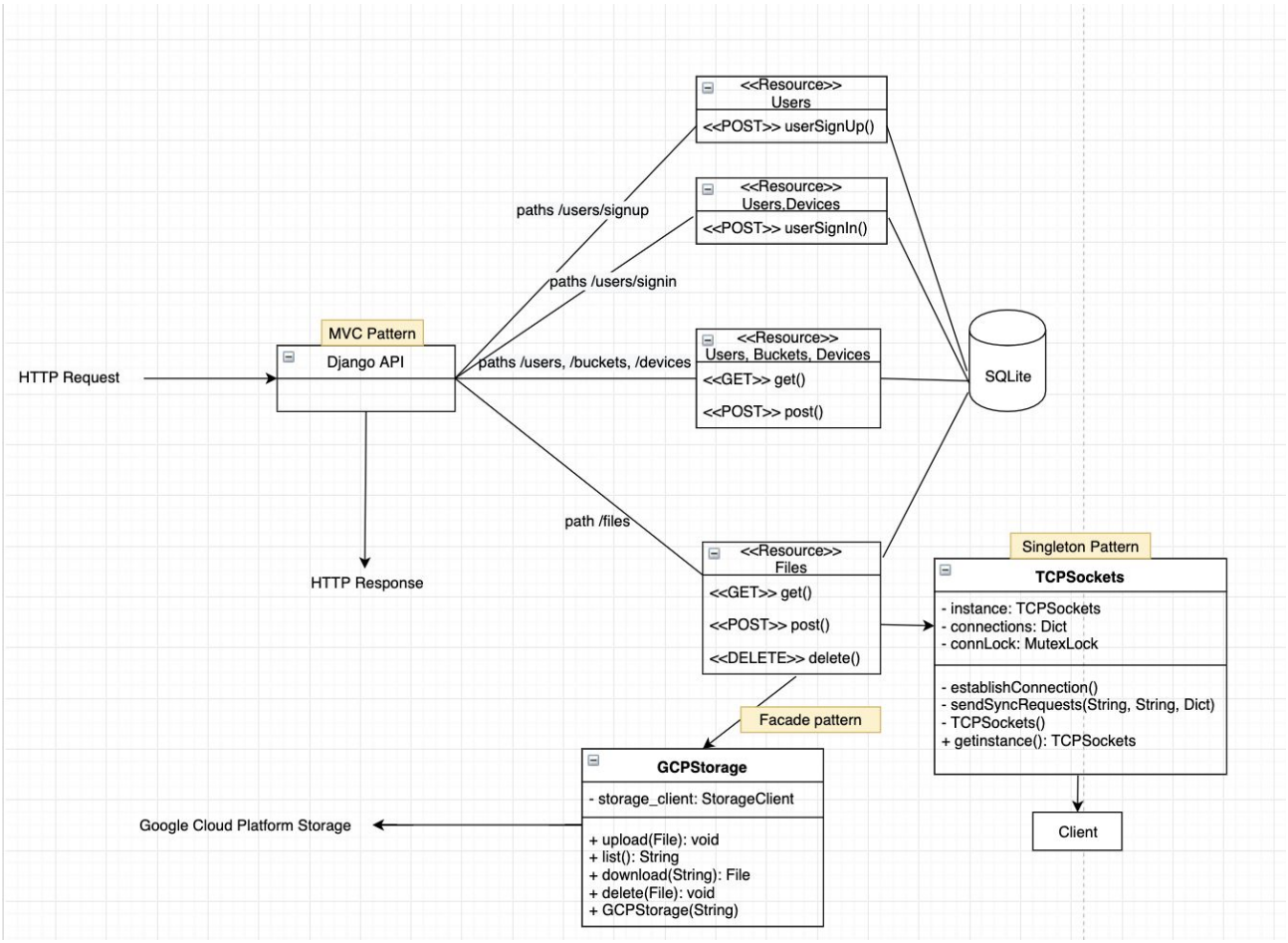
Finally, we also broke the ProfileDialog into three separate dialogs, the LoginDialog, SignUpDialog, and ProfileDialog. We found during development that Qt doesn't allow for dynamic UI changing with dialogs so we were forced to have three separate ones for logging in, signing up, and general profile stuff including signing out. This added more classes but decoupled the potential profile dialog from handling all three tasks.

Besides those major changes, the final class diagram followed close to what the final class diagram ended up as. The only other changes came in the form of methods and fields that were added during realization of need while developing.

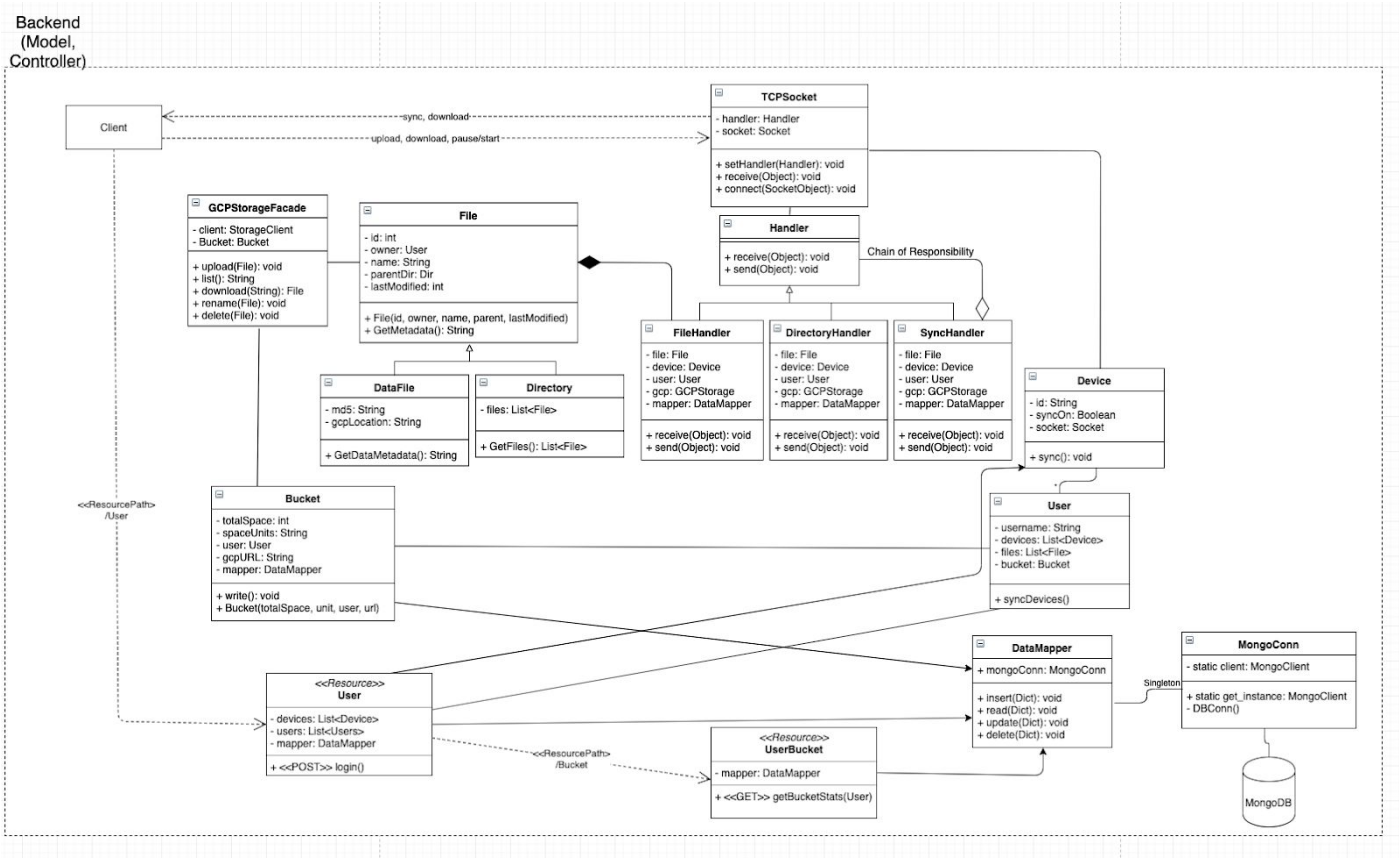
What else is missing from the new class diagram is the inheritance from Qt specific classes, these were left off for the sake of space as all objects inherited from a Qt specific class and would only complicate the diagram further.

Backend Diagram

Final class diagram



Initial class diagram (from project 4)



Changes between diagrams

So a lot of changes were made between the proposed class diagram in project 4 and the final class diagram. A decent amount of these changes were made because we were not very familiar with a decent amount of the technologies we wanted to use (we used this as a chance to learn new stuff).

Key changes:

To start off, we initially wanted to use MongoDB as our database, but found out that Django is well built for relational databases because of its built in ORM capabilities. This led us to change our database to be Sqlite3. So we removed the DataMapper pattern and used Django's ORM instead. This also required us to change some of our endpoints for django since we will be using tables to represent entities now instead of storing our "User" objects in Mongo Documents.

So now we have User, Device, and File objects that reflect the Django models and these objects are created within each Django endpoint (pretty much Django pulls a row from the database and creates and object out of it).

Another change we made was to the socket logic. Originally we wanted it to handle all file operations, but this was too complicated and we decided to move most of the logic out to endpoints instead (/file endpoint). The socket now only sends "sync" requests to clients. Thus,

we removed all the Chain of Responsibility pattern stuff because with endpoints, we know exactly what the request is for.

Lastly, we initially wanted to use OAuth as a our authentication method, but this ended up being a difficult task with QT (client-side stuff), so we ended up making our own authentication process. Again, this changed some of our endpoints and how other routes dealt with “authenticated requests.”

Third party code

- Django REST Framework - statement
    - https://www.django-rest-framework.org/
        - Django Rest Framework has built-in features that make building a REST API in Django easier. It was used in path creation, object serialization, and HTTP response building.
- GCP SDK: This was used to handle communication with our GCP Project. Mainly authenticating our service account (private key created for this service) so it can interact with the GCP Storage buckets.
    - https://cloud.google.com/storage/docs/
- Qt Creator/Qt 5.12.5: Qt was used for the front end development. It allows cross-platform development so we could have an executable on Windows and Mac for the client.
    - https://www.qt.io/
- Examples followed during development
    - Front end
        - Tcp socket example for Qt: https://www.bogotobogo.com/cplusplus/sockets_server_client_QT.php
            - This was mainly used to know how to read from a QTcpSocket as well as which signals/slots to connect with the object. The connected and disconnected functions follow this example though the connection to hosts as well as what we write and handle the read is original.
        - Http networking example for Qt: https://code.qt.io/cgit/qt/qtbase.git/tree/examples/network/http?h=5.13
            - This example was used to understand how to perform Http requests in Qt. Most if not all of our code for Http requests is original, but the example showed the type of data and an example of how the data was prepared for the request.
        - Download manager example for Qt: https://doc.qt.io/qt-5/qtnetwork-downloadmanager-example.html
            - This was used to know how to download directly from an Http get. The functions DownloadFinished and the queue for QNetworkReply downloads follow closely to the examples here though what we did post download is original to our needs.
        - Http multipart form request example and boundary code for Qt: https://stackoverflow.com/questions/39549211/qhttpmultipart-generates-different-boundary
            - This example allowed us to understand how to perform a mutli-form Http request with Qt. From this example, we used the code to create a random boundary necessary to send the form as a hard coded boundary was not working as well as the headers. How we set up the mutli-form for our needs is original.
        - Drag and Drop example in Qt: https://wiki.qt.io/Drag_and_Drop_of_files

            - This example was used to know how to drag and drop files into the FileExplorerScene for uploads. The dragEnterEvent and the dragMoveEvent is lifted from this example and third-party but the dropEvent is original for our needs.
    - Backend
        - Singleton in python example: https://python-3-patterns-idioms-test.readthedocs.io/en/latest/Singleton.html
            - Used this to see how to implement a singleton pattern in Python
        - GCP SDK create bucket example: https://cloud.google.com/storage/docs/creating-buckets#storage-create-bucket-code_samples
            - Used this to help understand what API calls I needed to make and the flow I had to go through for creating a bucket
        - GCP SDK upload example: https://cloud.google.com/storage/docs/uploading-objects#storage-upload-object-code-sample
            - Used this to help understand what API calls I needed to make and the flow I had to go through for uploading files
        - GCP SDK download example: https://cloud.google.com/storage/docs/downloading-objects#storage-download-object-python
            - Used this to help understand what API calls I needed to make and the flow I had to go through for downloading files on a bucket
        - GCP SDK list example: https://cloud.google.com/storage/docs/listing-objects
            - Used this to help understand what API calls I needed to make and the flow I had to go through for listing the files stored on a specific bucket
        - GCP SDK delete example: https://cloud.google.com/storage/docs/deleting-objects
            - Used this to help understand what API calls I needed to make and the flow I had to go through for deleting a file stored on the bucket

Statement on the OOAD process for your overall Semester Project

1. Attempting to shoehorn patterns that didn't fit well in the project. Our initial design made extensive use of patterns and technologies we had no prior exposure to. Luckily we were able to pivot and change scope with plenty of time left to implement useful patterns with the right tools.
2. We made sure our classes were always single-purpose, but sometimes underestimated the size of individual classes. Some classes became very large because the sole purpose they were created to fulfill required many methods. Example- the NetworkManager class for the client which was used for all networking while we could of broke it out into HttpManager and TcpManager.
3. Code was made for re-use so we didn't repeat ourselves. This allowed us to easily make changes to frontend and backend code as we progressed.
4. It was easy to communicate the system via pattern and entities that we defined in our class diagrams.