Michael Inglis

API

Introduction

To link both the hardware and applications of the app controlled lock, an API had to be built which stored and served all the user data. It was also required to apply the logic for the hardware to function as expected across the range of user applications and allow for social features such as friends and granting friends access to locks.

My contribution was primarily the API as a whole. Due to the agile nature of the project and the close collaboration of elements to give us a strong system integration however, contributions towards the API were given by both Sam and Ayrton. While they will I'm sure do it themselves, I will also make attempt to acknowledge both of their contributions in an effort to assume I did the rest. This will also allow me to provide a complete report on the API in terms of both design and implementation.

Design

As mentioned in the introduction, there was strong collaboration, not only in developing the API, but in the planning of the API. As all of the team members would at one point be using the API, it was imperative that the functional requirements matched their own functional requirements on both the app side and the hardware side of the integration.

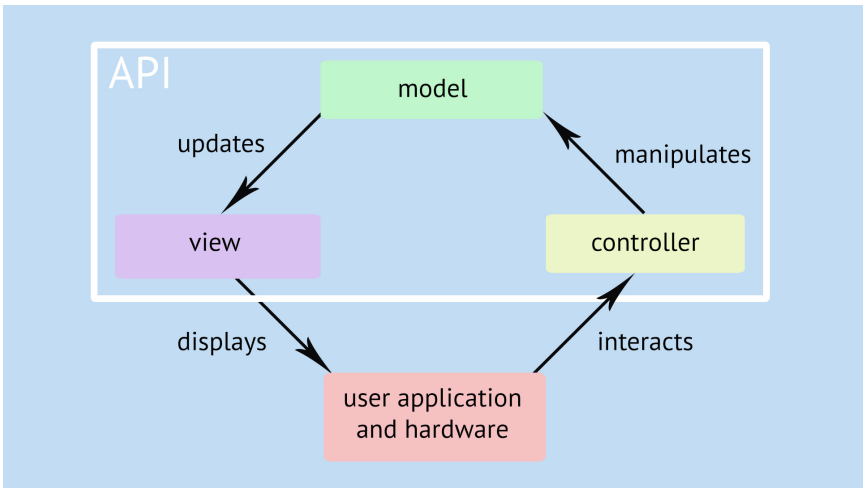
Version control repositories were set up on Github[1] and a uniform committing notation was decided upon in a better effort to ease collaboration. The repository of the API and each respective user's commits[2] may be seen on Github.

Due to the iterative development process undertaken in this project, the API design wasn't finalised before development started. The presentation of the project progress each week forced team reflection as regards to integration which further refined or altered the functional requirements of the API, impacting on the implementation the following week. The API I will be documenting here in the Design section will be the final version of the system that is available to see and interact with online[3]. I will give a detailed analysis as to how we arrived at the system described below in the Implementation section. The endpoints may be seen in the table below.

API Endpoint	**Functionality**
`/user`	• GET: Returns list of all users • POST: Registers user
`/user/<user_id>`	• GET: Returns all information associated with user_id
`/me`	• GET: Returns all information associated with logged in user

`/lock`	• GET: Returns list of logged in user's locks • POST: Registers a lock with a user
`/lock/<lock_id>`	• GET: Returns information associated with lock_id
`/open/<lock_id>`	• PUT: requests lock of lock_id to open
`/close/<lock_id>`	• PUT: requests lock of lock_id to close
`/im-open/<lock_id>`	• GET: lock authenticates with server it's open state
`/im-closed/<lock_id>`	• GET: lock authenticates with server it's closed state
`/friend`	• GET: Return list of friends with access to user's locks • POST: Register a friend • DELETE: Delete a friend
`/friend-lock`	• POST: Enable a friend to access lock • DELETE: Revoke friend's access to lock
`/hello`	• GET: Endpoint for unit testing
`/protected-resource`	• GET: Endpoint for unit testing

The System was designed in Flask, a Python Framework following the MVC (Model-View-Controller) design pattern[4]. To explain each element of the system, I will break it down into three sections, one about the Model, the Controller and the View respectively.



Models

The Model in the system is a collection of Databases on the server, the schemas of which can be seen in the models.py file on the repository. The structure of these may be seen in the tables below. Note that some entries appearing in models.py have been excluded due to irrelevance.

USER
id (Primary Key)
email

first_name
last_name
password

The USER table allows us to keep a database of users and their information. This personalises the app in that it can give each user their own environment with their own locks, friends etc. It also adds a level of security with a hashed password which will be explored within the controller section. The USER.id seen within this table iterates automatically based on the amount of users registered with the system to ensure that each user has a unique ID avoiding any conflicts which may otherwise occur if the email had been set as such.

LOCK
id (Primary Key)
name
requested_open
actually_open

USERLOCK
user_id (Primary Key, Foreign Key referencing USER.id)
lock_id (Primary Key, Foreign Key referencing LOCK.id)
is_owner

The LOCK and the USERLOCK tables are set up to keep track of the locks registered with the system and the many-to-many users-to-locks relationship. The LOCK.id field is functionally identical to the USER.id key mentioned above. LOCK.name is simply for allowing the user to keep track of their locks within the the user application. LOCK.requested_open and LOCK.actually_open refer to states the lock can be in, something that will be will be explored within the Controller section. When a lock is registered, it is added to the LOCK table and the USERLOCK table with USERLOCK.is_owner set to true with the USERLOCK.user_id set to the user. Any subsequent additions to the USERLOCK table with the same USERLOCK.lock_id have USERLOCK.is_owner set to false. This would be seen in scenarios of users adding their friends to their owned locks. This also acts as a check to ensure users can only enable lock control to their friends if they own (were the first to register) the lock.

FRIEND
id (Primary Key, Foreign Key referencing USER.id)
friend_id (Primary Key, Foreign Key referencing USER.id)

The FRIEND table was set up simply using two user IDs. This relationship allowed users to then assign their "friend" to control one of their locks.

Controller

The Controllers for each of the models can be found in the .py files with the "api" prefix. They carry out the relevant functionality we would expect of manipulating the data within the models and updating it for the View. The API was

broken up into 6 files with such a prefix to add modularity and thus increase readability. We stuck to a uniform class naming convention devised by Sam in an effort to better handle scalability, naming the classes DATABASE_NAME+Detail to deal with registering and receiving information generally associated with one instance of the whatever was being pulled from the database and DATABASE_NAME+List to handle multiple instances of the database objects.

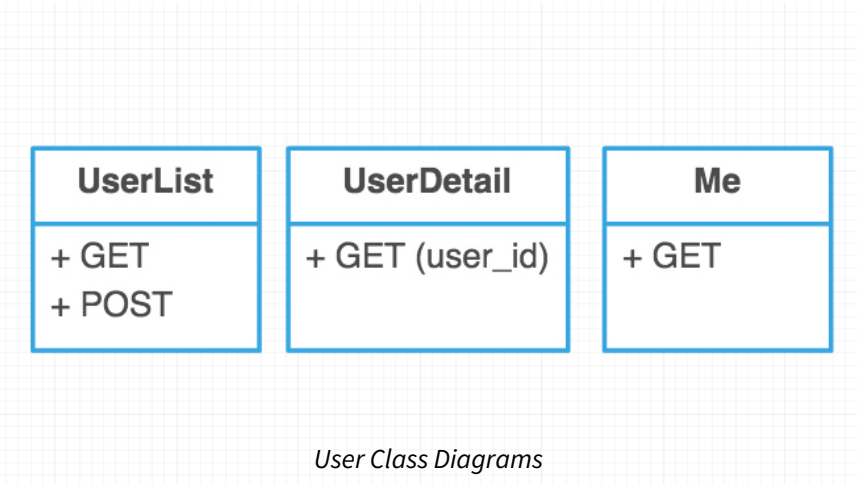
API Controller

File: api.py[12]
Endpoints: /hello, /protected-resource

This file acts as an umbrella file, storing the carrying out the highest level functions such as assigning the classes to endpoints and holding the most basic of endpoints used simply to aid in testing responses from the API. The ProtectedResource class was written by Sam.

Users

File: api_users.py[13]
Endpoints: /user, /user/<int:user_id>, /me



User Class Diagrams

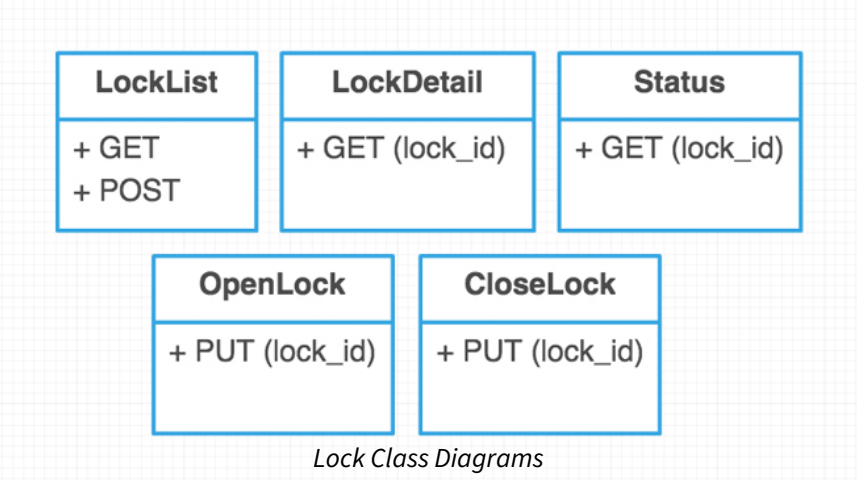
The classes in this file handle all the information regarding the USER table.

The UserList class features a GET method which returns all the users in the table. The POST method enables users to be added (or registered) to the database. This method includes an encrypt_password method (seen in api_helper_fuctions.py) which ensures that salted password hashing is used, the result of which is that the user's password is encrypted and thus isn't stored in our database as plain text.

The "UserDetail" class simply includes a GET method which takes in the parameter of a USER.id and returns the rest of the information associated with the user. The "Me" class does the same with the exclusion of the parameter, supplying the user information of the user calling the function. This simply allows for one less API call when getting the user's profile information and doesn't require the apps to store any user information locally.

Locks

File: api_locks.py[14]
Endpoints: /lock, /lock/<int:lock_id>, /open/<int:lock_id>, /close/<int:lock_id>



Lock Class Diagrams

The classes in this file handle all the information regarding the LOCK and USERLOCK table.

The "LockDetail" class and it's GET method returns the information from the LOCK table associated with the given LOCK.id.

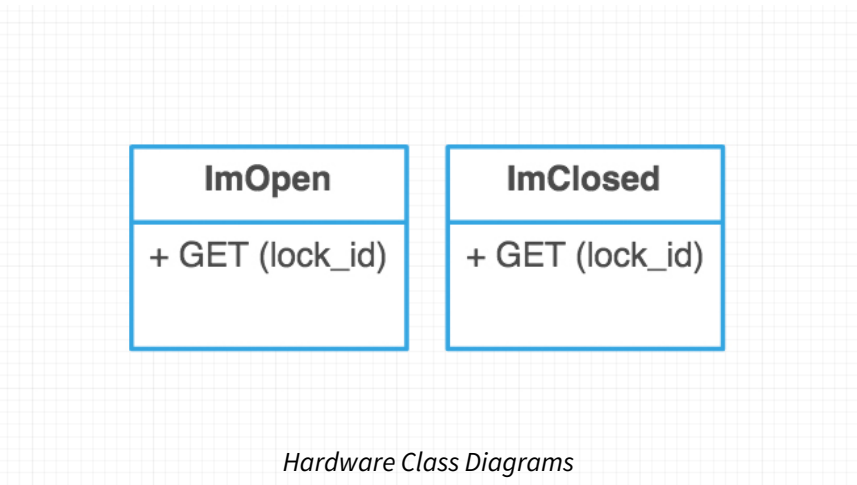
The "LockList" class has a GET method which returns a list of all the user's own locks. The POST method registers a user to a lock.

The "Status" class simply has a GET method which returns the LOCK.requested_open state. This is used purely for Human-Computer-Interaction reasons, giving the user feedback that their request to alter the state of the Lock has been recognised and the API is now awaiting feedback from the lock that their request has been fulfilled.

The "OpenLock" and "CloseLock" classes simply include PUT methods which check if the user has permission to change the state the lock before changing the LOCK.requested_open state. The outcome of this acted upon in the api_hardware.py classes which are called by the hardware lock.

Hardware

File: api_hardware.py[15]
Endpoints: /im-open/<int:lock_id>, /im-closed/<int:lock_id>



Hardware Class Diagrams

The classes in this file handle the physical opening and closing of the Hardware Lock. The final version of the logic was implemented by Ayrton, but the first couple iterations were implemented by myself and will be explored further in the Implementation section.

Both the "ImOpen" and "ImClosed" classes and their sole GET methods are called by the hardware lock. This gives the controllers feedback of what state the Hardware Lock itself is in based on the method that is called (ImOpen or ImClosed). The hardware is influenced by the HTTP code which is returned on calling the GET methods. Given that we have two Boolean variables, LOCK.requested_open and LOCK.actually_open this gives us four possible states for the system. This can be seen in the two truth tables below:

ImOpen
if LOCK.actually_open is False, it is set to True for the next call.

LOCK.requested_open	LOCK.actually_open	HTTP Status Code
True	True	202
True	False	202
False	True	200
False	False	200

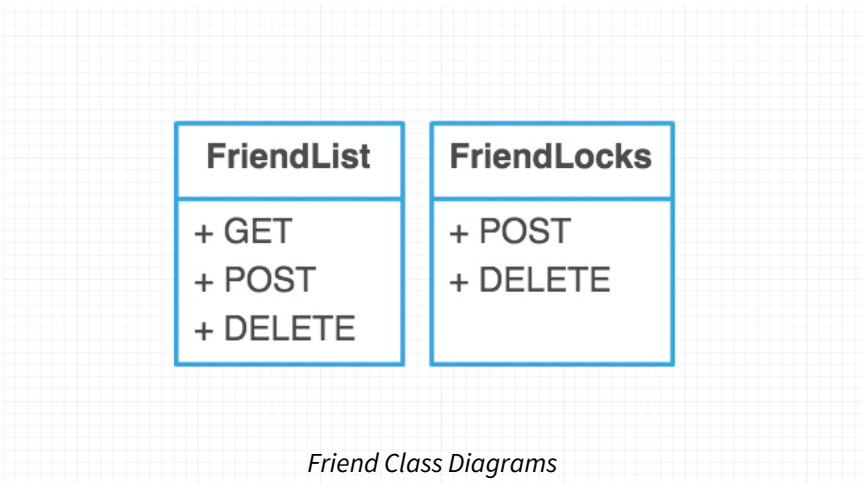
ImClosed
if LOCK.actually_open is False, it is set to False for the next call.

LOCK.requested_open	LOCK.actually_open	HTTP Status Code
True	True	200
True	False	200
False	True	202
False	False	202

The endpoint was designed like a question, ImOpen? or ImClosed?. The response code 202 was used to return "YES" to the Hardware Lock's question and the response code 200 was used to return "NO" to that question. The hardware lock would then act based on it's actual state and the response answer before sending the next "question". Assuming there was no change within the databases, the hardware would continuously send the same endpoint, dependant on it's open or closed state. This wouldn't change until LOCK.requested_open has it's state changed. The API originally just changed it's state based upon the user's input. What was built in the end however, was a very strong integration of hardware and software, both relying on each others feedback before changes took place.

Friend

File: api_friends.py[16]
Endpoints: /friend, /friend-lock



Friend Class Diagrams

The classes in this file handle the adding of users as friends by other users such that users can assign friends to their own locks.

The "FriendList" class includes a POST and a DELETE method which registers a friendship and deletes a friendship. The GET method in this class returns a list of all the user's friends with a nested list for each friend against each of the user's

locks and whether or not that friend has access to them. This enables everything that needs to be done visually within a user application end to be done with one call to the API.

The "FriendLocks" class again includes a POST and DELETE method, both of which register a friend to a lock and deletes a friend from a lock respectively. Registering a friend has to be done before assigning one to a lock and deleting a friend removes their access to all of one's locks.

API Helper Functions

File: api_helper_functions.py[17]

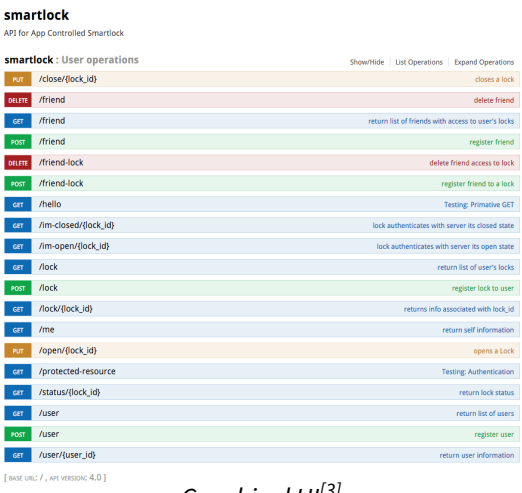
This collection of methods are simply static functions which are called more than once throughout different classes and for modularity and better readability were broken off and placed in this file. The large nested method "add_related_locks" which features extensive amounts of SQLAlchemy was written by Sam after my original implementation required much more rigorous covering of all cases as the API scaled.

Several of these methods such as "check_auth" were written to implement the database models with Flask security. "requires_auth" was written as a decorator for the classes, allowing us to cleanly place it above any endpoints which required user authentication. "change_lock_state" does as it suggests, changing the state of a given LOCK.id to whatever given state. This was the first naive implementation during the first integration, when the "pending state", LOCK.requested_open was originally just the state of the lock. Methods such as "is_user_in_db" and "get_user_id" were simply to avoid repeating the same code on different endpoints.

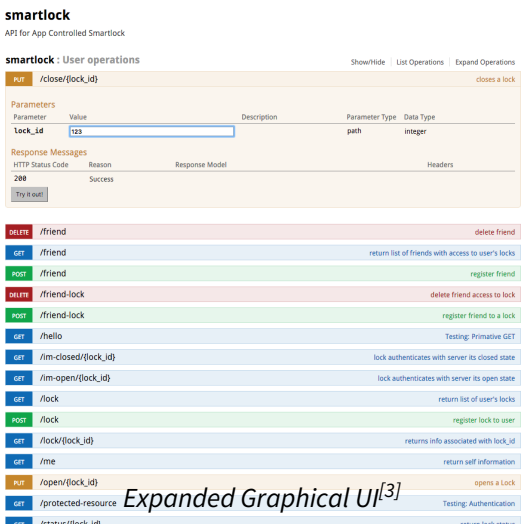
View

As the API currently stands, there are three elements that compose the View layer. The first is simply the JSON it provides back to the user. Instead of generating our own JSON, composing it as dictionaries and converting it into JSON manually, Serialisers were used to automate the process. This allowed us simply to create custom serialisers within the serialisers.py file which simply had to be specified as decorators above each JSON returning method.

In an effort to better test and bug fix the API, special attention was given to the HTTP Status Codes[10] . This allowed for extremely thorough unit testing and would ultimately help in the development of the user applications. I adhered as close as possible to the definitions given in the IETF Internet Standards. Improvisation however was required when implementing the methods for the hardware lock.



Graphical UI[3]



Expanded Graphical UI[3]

The Third element of the View layer was a GUI which was automatically generated from the endpoints and their descriptions. This gave us not only up-to-date documentation, but an interface which could be used to manually input data into the endpoints. The automation of documentation allowed me to move away from manually documenting each endpoint within the GitHub wiki, which was done for the first half of the project.

Implementation

The plan for implementation used technologies most of us were familiar with. We chose Flask to act as the framework over Django due to the lightweight nature of the framework[5]. Flask simply requires a module import. Django has such features out of the box, but everything is built in, contributing to a significant amount of bloat. The project was then deployed through Heroku, a cloud web server. Several other services such as AWS were looked at, but Heroku matched our needs due to its cheap price as regards to scalability, automatic integration of databases derived from model schema and quick deployment from GitHub. Sam and I collaborated extensively in deciding the technologies and setting up the initial project. Much of his initial work was in the setup of the file structure as we originally had major problems with circular dependencies within the imports. As stated at the introduction, I will attempt to name the contributions of others in an effort to assume I did the rest of what is mentioned in both this section and the one above.

Sam's Obsessive Compulsive disorder as regards to integration and regression testing came into play from the start, starting off a test suite which I would later go onto populate. Such a robust set of tests enabled us to collaborate on the API without any problems simply because any commit to the repository would be preceded by a run of the test suite. Despite the slower development initially, writing tests before implementing functionality enabled us to reduce development time substantially towards the end as the development was simply getting the tests to pass. Debugging still took place at each cycle but we only have to debug the new parts we added instead of all the code in the whole project. Having local databases in PostgreSQL allowed us to manipulate data locally, without having to deal with deploying the project each time. Breaking up the API into a more modular design also helped speed up the development. Having a more modular structure made it considerably more readable and thus more reusable; different classes could be easily instantly repurposed for different projects without any decoupling.

The Flask-RESTful[6] extension was added to the Flask application to enable me to create endpoints and start writing tests for those endpoints, giving the app developers a decent idea of what sort of methods of data transmission would and could be used. My initial reaction was to start developing a user login system before we had finalised specifications for endpoints and functionality. This led me to using a service I'd used in the past called Stormpath which basically handles all security as regards to user logins. Once implemented, it became evident very early on that this service wasn't anywhere near as flexible as we required as regards to user data (locks, friends, etc) and as the application scaled would become very slow, due to the large amount of JSON which was being transmitted and crunched on the server. Stormpath served all the user data at once and the server would have to search through it. This waste of computation every time the server was pinged by a user could be avoided by local databases, where required data was called by relevant SQL queries. JSON was selected over alternatives such as XML due to everyone's previous experience and its readability and simplicity.

In rejecting Stormpath we chose to use SQLAlchemy, a database toolkit for Python[7]. This was used for all interactions with the database and was chosen due how it converts between data and objects in Python, not only in theory more straightforward for development as data can be manipulated easily, but extremely powerful too, able to carry out complex queries extremely efficiently. Leaving behind Stormpath had left behind a gaping hole in our security, thus Flask-Security, an extension which allowed session based authentication and password encryption was added[8]. Flask's out the box HTTPBasicAuth didn't work in conjunction with Flask-Security due to it's inability to hash a password and validate it against a user's password in a database. This resulted in me having to write a custom authentication decorator for endpoints mentioned in the previous section. The password Salt and Hash function[9] were pulled in from Environmental Variables, far more secure than having them hard coded into the API.

Once user authentication had been setup with the user accounts, I then turned my attention to the lock database and the opening and closing endpoints. In an effort to have a quick initial integration, there was simply a boolean state for the lock, stored in the database. This changed based upon the commands being called from an application and assumed that the lock itself was constantly pinging the API. This was the model we showed in our first integration. While it worked, there was no level of authentication for the user to know their door had opened or if the API had even received their message. This was why there was need to also include a pending state and logic which takes into account what state the lock currently is.

This was arguably the section which underwent most change. I implemented the next version which introduced the "ImOpen" and "ImClosed" classes along with the new pending state. The logic was based on a "Story" of a unit test written by Sam in tests/integration_tests/test_open.py. When the tests passed, we moved on to other parts of the project. Due to an assumption within the group about the functionality of the software on the lock hardware, it became evident that we would have to scale back on the computation on the lock hardware side and move it to the API, something I reimplemented, though bugs were found when it was tested with the actual hardware. Sam further altered his test and Ayrton implemented new logic to the new specifications as I was working on having the Friend classes done for the demonstration.

When I was implementing the friend database and classes, I was in tandem working on the userlock database which was would allow the previously one-to-one user-to-lock relationship to become many-to-many. Much of what I wrote initially was refactored into Sam's giant method in api_helper_functions.py. While I had attempted to do most the database interaction in memory, he did it in SQLAlchemy objects, severely reducing the load on the server.

Evaluation

Aside from my work on the API itself, I also presented the section in our presentation on the API as well as directing and editing the video[11] demonstrating the functionality of the System Level Integration.

Looking back over the development of the API and it's influence on the final project, I'd have to say I was pleased. In a normal development cycle, the user applications would be developed after the API was at least finalised on the functional requirements, not in conjunction with. This gave us the problems of prioritising what to have up and running first to enable at least a quick integration. Starting with the User accounts initially was probably an error. This took a decent amount of time to set up, considering things such as security. This probably could've been overlooked in an effort to have a primitive first integration up and running.

Choosing Flask over Django I would consider in hindsight, an error. Django has out of the box user authentication and models which would have improved our security while cutting down on our development hours. Flask was selected on the assumption we would be using Stormpath to handle all of these things but for whatever reason remained after Stormpath was rejected. Opening the door to SQLAlchemy presented problems too. It's syntax is can get profoundly difficult for complex queries, a problem we wouldn't have had had they been written in Django's ORM. While less powerful, than SQLAlchemy, Django's ORM is simpler and thus more maintainable. It also encompasses every possible use case we would require in turn, making the development of the API easier.

Security could be better on the API-Lock end as it is simply communicating with HTTP requests. Hashing someone's passwords is about as far as my knowledge on Security currently goes however. It is always my intention to abstract over it and use an already reputable service for such things. We could have implemented more secure communication protocols, but it was as secure as we needed it to be for the prototype it was.

The naming of "LOCK.requested_open" should be "LOCK.change_pending" to increase readability. It is a remnant of the second iteration of API-Hardware integration attempt and was never changed.

None of these problems detract away from the fact that what we ended up with was a full System Level Integration which had both ends tightly woven into the API. Although there was only one hardware lock, the API was built to be scalable, accommodating for any number of locks which could be registered to a user and assigned to any of their friends. The design also accommodates for flexibility in terms of code reuse. The API could quite easily be repurposed for different types of hardware by simply repurposing the logic. All it currently does is accommodate for a binary state (open or closed) and could be hooked up to any hardware which has two states and is capable of relaying http feedback. Classes related to the user and the friend functionality could quite easily serve as the basis for any social network functionality without any modification. The code which handles the locks is completely separate from the core social classes. The project is publicly available on Github and may be forked for potential reuse.

While the API doesn't lend itself to quantitative evaluation, it should be noted that it was rigorously tested in the field by both the hardware and front end developers. This "user" testing definitely had a positive effect on the API, pointing out bugs and shaping it's functionality. The agile development cycle couldn't really accommodate

recording this data simply due to time constraints and overall relevancy. Special attention was also paid to things like the response speed on the Heroku server, one of the reasons for dropping Stormpath.

We worked very well as a team, collaborating with each other on different and the same parts. I particularly appreciated being able to work with both ends of the project, designing and implementing software to be used with both the hardware and front-end applications.

I'm proud with what we achieved and excited to implement what I learned in future projects.

References

1. Smartlock Github Repository (https://github.com/mingles-/SLIP-D)
2. Smartlock Github Commits (https://github.com/mingles-/SLIP-D/commits/master)
3. Deployed API (http://slip-d-4.herokuapp.com/)
4. MVC Wikipedia (https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller)
5. Django vs Flask (https://www.airpair.com/python/posts/django-flask-pyramid)
6. Flask-Restful (http://flask-restful-cn.readthedocs.org/en/0.3.4/)
7. SQLAlchemy (http://www.sqlalchemy.org/quotes.html)
8. Flask-Security (https://pythonhosted.org/Flask-Security/)
9. Custom Hash Schemes (https://pythonhosted.org/passlib/lib/passlib.hash.pbkdf2_digest.html)
10. HTTP Status Codes (http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html)
11. Smartlock Video (https://www.youtube.com/watch?v=cXNFukir_OQ)
12. api.py (https://github.com/mingles-/SLIP-D/blob/master/Project/api.py)
13. api_users.py (https://github.com/mingles-/SLIP-D/blob/master/Project/api_users.py)
14. api_locks.py (https://github.com/mingles-/SLIP-D/blob/master/Project/api_locks.py)
15. api_hardware.py (https://github.com/mingles-/SLIP-D/blob/master/Project/api_hardware.py)
16. api_friends.py (https://github.com/mingles-/SLIP-D/blob/master/Project/api_friends.py)
17. api_helper_fuctions.py (https://github.com/mingles-/SLIP-D/blob/master/Project/api_helper_fuctions.py)