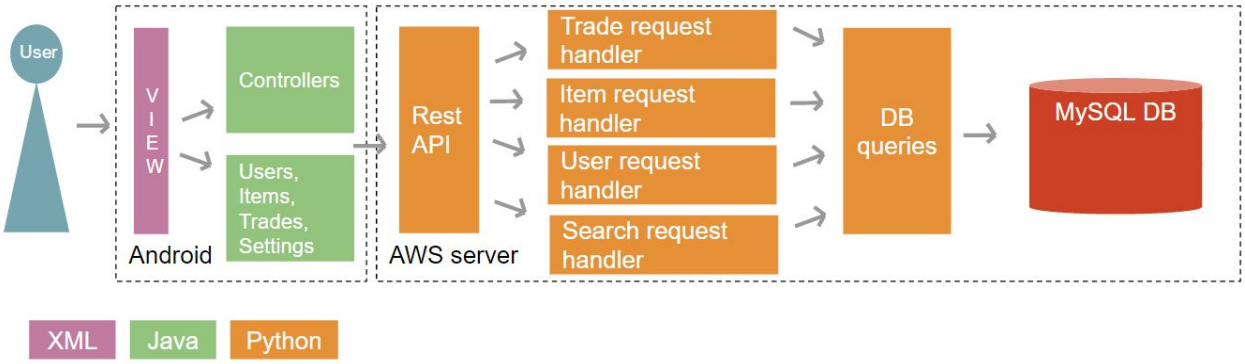
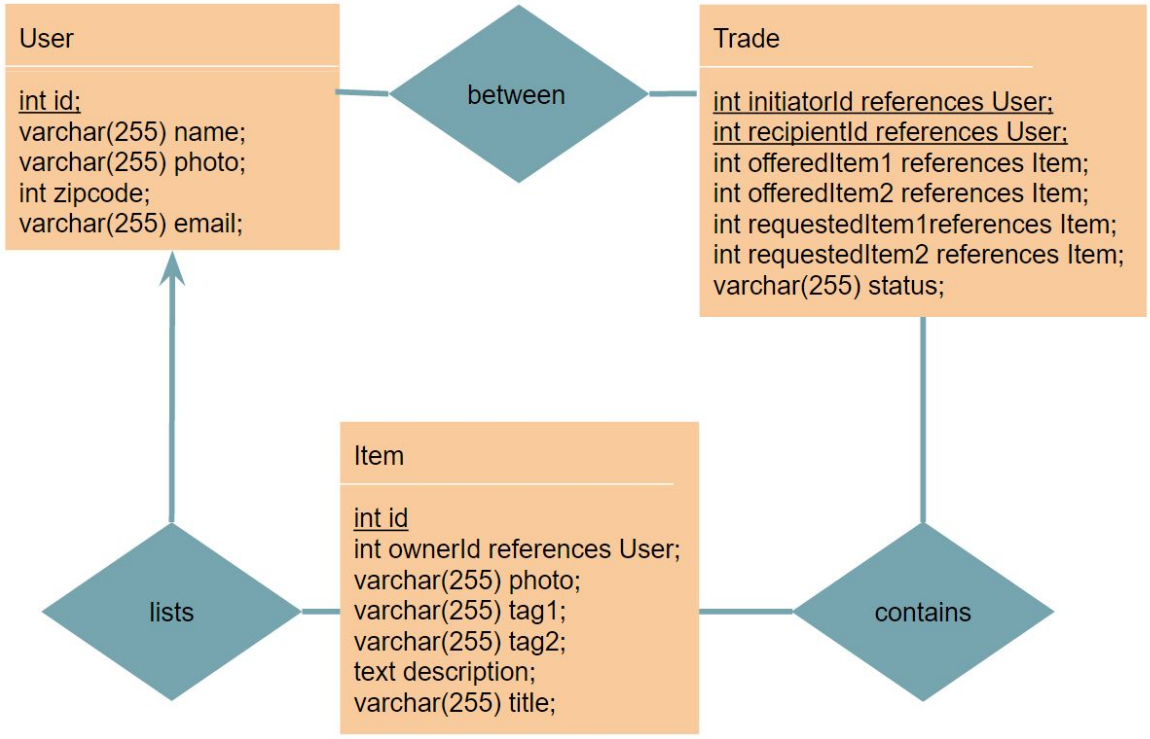
Software Design Specification

System Architecture

High Level Overview Diagram:



Database Schema:



Overview:

The app will consist of a view that the user can see. The markup of this view is in XML, and controlled by a Java layer. The controller utilizes a Java API to ask the server

for data. This data is saved into objects representing items, users, trades, and user settings. The server's request handler matches the request to a worker function. Each request function makes requests to the database handler and organizes that information into JSON for the server client. All of this server work will be written in Python. The database handler is the only Python file that will directly touch the database. Images will be stored on the server, and the database will hold links to them.

Alternatives:

The team weighed the idea of storing item tags as json strings in the database versus storing each tag in a separate column in the database. We ultimately decided that storing JSON in a MySQL database is a bad idea, and that MySQL has useful data types and we should use them. This design choice will make it harder to increase the number of tags an item can have later. We have to limit the number of tags that can apply to an item so that we know how many columns to create for tags, which we feel is a good constraint to have anyway for usability.

Another hard decision the team made was how to initiate server requests on the front end. We had initially considered using asynchronous calls through an API service, and perform as much of the work relating to loading as possible in the background. However, we soon came to the conclusion that doing so would require far more work than simply doing synchronous HTTP calls only when we needed them. Since we already had a very tight schedule in our project, we switched to the simpler solution to ensure we would be able to implement everything in time.

Assumptions:

As of the current design, the front end will be making many requests for each little thing. This could mean a lot of traffic for the server, so we are assuming AWS will be able to handle a lot of traffic. It will also potentially need to store a lot of data, because of item and user images. This could cause scalability issues in the long run but we are assuming AWS can handle our work for now.

We also have no infrastructure for making inventories private. If users do not like this feature, it could be quite hefty to implement later. We are assuming they will want everyone to see their inventory because this is after all a trading app.

Our version of security is requiring the UserId for requests to the server, so if someone has another person's userId, they can pretend to be that person and make requests to the server. This is a security risk for the app, but the consequences are hardly catastrophic, so we will assume our current implementation is enough.

UML Class Diagram:

Here is a link to the class diagram in full size, since it is too big to display here:

https://dowelc.github.io/finders-keepers/NewestSDSClassDiagram.class.violet.html

UML Sequence Diagrams:

This is a sequence for a user searching for an item and sending a trade request to another user:

https://dowelc.github.io/finders-keepers/searchDiagram.seq.violet.html

This is a sequence for a user adding an item to their inventory:

https://dowelc.github.io/finders-keepers/Add%20Item.seq.violet.html

Process

Risk Assessment:

1. Incorrect API specification leads to frontend and backend of the app doing different things.

   [Likelihood: Low] [Impact: Medium] Our process for how the front end utilizes the API minimizes the chances of this risk happening, as the front end defines the API commands they need, while the back end hosts a dummy API that mirrors actual API behavior during development. Should any API specifications lead to confusion between the two teams happen though, the effects would pretty heavily impact the front end, especially in the controller logic.

   In order to detect the problem, all decisions on modifying the API need to be discussed together to make sure no one has outdated knowledge of it. Should the problem occur though, we would have to first discuss what API changes are needed to satisfy the front end's needs, and then modify existing code to work with the updated API.

2. We are not able to secure funding for and/or configure AWS to set up our backend hosting in a timely manner.

   [Likelihood: Low][Impact: Low] AWS is well established, and given our educational credits, we should be able to obtain funds for it/free credit. In the small chance that the service itself doesn't work with our design, we would need to spend more time looking for another service, or setting up a personal machine to host everything.

   To minimize the impact should it occur, we have made sure to design the back end to be hostable from any machine, given it has 24/7 uptime and a stable network. We are also investigating setting up AWS over the weekend, so we should know for sure if there will be any problems fairly soon. If after investigation, we deem the service to be unusable for our needs, we would have to set up a personal machine to act as the host (spare laptop, desktop, etc.).

3. Our item tagging structure as we have defined it now may be too general for usability and performance - too many different types of tags may make it difficult for users to find items, and difficult to search the database quickly.

   [Likelihood: High][Impact: Medium] Allowing multiple tags on an item is necessary for items to match well on searched, but with our current schema, each tag is in it's own column. It is very likely that this will

introduce performance and scalability issues. While the likelihood is high, it isn't a critical problem, as the core functionality of tagging will still exist.

To minimize the chances of it occurring, we are investigating other possible ways of storing multiple tags, to allow the database to store it efficiently, as well as allowing searches to be performant. To detect if performance is suffering, we can always compare our search times to our non-functional performance requirements, making sure it doesn't approach the upper limit unless we are searching through tens of thousands of entries. Should it become a problem though, we will have to focus more effort into finding a suitable replacement schema or search algorithm.

4. Our client side application features a very tall activity stack for certain user workflows, which may lead to performance issues.

   [Likelihood: Medium][Impact: High] If we are not careful of how we manage the Android activities, the possibility of this risk becoming a problem becomes plausible. If it does end up being a problem, it would be at an architecture/design level, which would have considerable impact on our development schedule.

   To minimize the chances of it occurring, we are investigating Android's best practices for developing to make sure we are using what it provides in the intended manner. We will be able to detect if this becomes an issue through frequent manual testing, checking if certain workflows start to bog the application down. Should we notice performance problems, we would first investigate the problematic components of the front end before fixing. If the problem escalates to be beyond specific components, we would have to modify the design to have a shorter activity stack for tasks.

5. Our architecture as it is now is structured around synchronous HTTP requests, which may become slower as the database scales up. Switching to asynchronous HTTP requests may be difficult if we do not account for it.

   [Likelihood: Low][Impact: Medium] Assuming we can keep our back end processing time per request fairly low -- 10-100ms -- we should be able to handle 10-100 users in one instance with decent response times. This is more of an issue beyond development, if we wanted to have a larger user base, as performance could suffer during high usage periods.

   To minimize the chances of it occurring, we are investigating how we would incorporate asynchronous requests instead, and if it is viable within

our time frame. Checking for the problem can be done during group meetings, by attempting to simultaneously run multiple instances of the application, and seeing if there is notable slow down compared to running individually. If it becomes an issue, we might have to deploy multiple servers for the development phase, until we have time to come back and redesign the application to use asynchronous calls.

Project Schedule

	Task/Milestone Description	Person Hours	# of People
Week 1	Complete Design Specification & familiarize with front/back-end tools.		
Front End	Familiarize with Java, Retrofit, Android Studio, and Android Bootstrap.	4	3
Back End	Familiarize with Python, Flask, MySQL, and PushNotify.	4	3
Entire Team	Complete Design Specification.	10	6
Week 2	Complete Zero-Feature Release. This means having the front end establish the base screen of the application, while the back end starts setting up the infrastructure necessary for front-to-back end communication.		
Front End	Create a runnable startup page for Zero-Feature Release.	6	3
Back End	Set up AWS.	4	2
	Set up MySQL database.	2	3
	Implement basic API functions for remote servers.	4	1
Week 3	Begin User & Inventory Functionality. Since a basic remote server will be set up by this week, the front end can begin work on the login screen and the user/inventory pages. Back end will have a dummy API that the front end can use, while the actual API commands are implemented and integrated with the database.		
Front End	Create login screen.	6	1
	Create profile page, with inventory & wishlist.	12	3
Back End	Set up mock API for testing with front end application.	10	3
Week 4	Complete User and Inventory Functionality. The backend will finish implementing the API calls.		

Front End	Set up calls to User Settings API.	12	1
	Set up calls to Inventory API.	12	2
Back End	Finish user settings and inventory API functionality.	15	3

Week 5	Complete Feature Complete Release. Complete trading and browsing functionality. Using the same dummy API pattern, front end can work on implementing the necessary trading and browsing pages, while back end implements the real API.		
Front End	Create browse window.	12	1
	Create requests window.	12	1
	Create trade window.	12	1
Back End	Implement browse functionality.	12	1
	Implement trade and request management.	12	1
	Update API to handle necessary front end calls.	12	1

Week 6	Prepare for Release Candidate. By this point, all features should be complete, but may still have bugs. This week will focus on removing the bugs that remain in our existing features, and finishing any features that are incomplete.		
Entire Team	Bug fixing and complete any work items from previous weeks that could not be completed on time for Feature-Complete Release	20	6

Week 7	Complete Release Candidate. This week focuses on a final quality check; finalizing the integration test suites to catch more bugs, and eliminating them is everyone's responsibility. External/Internal documentation should be the only remaining component by this point.		
Front End	Plan/Record tutorial video footage.	1	3

	Edit tutorial video footage.	3	1
Back End	Create application manual.	3	1
	Create in-app help/info pages & links to manual/tutorials.	4	1
Entire Team	Finalize integration test suites.	6	6
	Eliminate remaining bugs.	10	6
	Determine possible stretch features in remaining time.	2	6

Week 8	Complete Final Release. Since the Final Release is Dec 06, there won't be much work towards the application itself.		
Entire Team	Prepare for demo.	4	6
	Create post-mortem SRS schedule.	4	6

Team Structure

Team Composition:
No one is responsible for one piece of the project; tasks will be distributed among members within teams so fair work distribution is kept consistent throughout the quarter. Everyone has a strong enough understanding of the overall architecture and their respective team's design decisions that overhead within teams should be minimal.

Front End	Back End
Responsible for implementing the UI and application code that lives on the android device itself. Components: ● Markup/UI Controls ● API Connector/Handler + Data ● Misc Device Logic Members: ● Artem ● Kierran ● Jared	Responsible for implementing everything that lives outside of the android device. Components: ● Remote Server ● Database ● API Members: ● Taylor <Project Manager> ● Danial ● Courtney

Team Communication:
The team will mainly communicate over slack, serving as a transparent platform for all group discussion. 1-2 weekly in-person meetings are arranged on Mondays and Wednesdays, from 6:30pm-9:00pm, in Odegaard, and additional meetings can be scheduled if necessary. In-person meetings in the future will try to be between 1-2 hours long.

Team Document Management:
All documents are shared on a google drive, allowing multiple people to edit files at the same time. Documents are organized by week, containing notes, weekly reports, finalized documents, etc. The entire team agrees on who's turn it is to write up the weekly report, and determine who's responsible for submitting files.

Testing Plan

For code testing purposes, we plan to leverage the extensive testing frameworks integrated into Android Studio for front-end unit and integration testing, and to use the Python unittest framework to perform back-end testing on the server infrastructure. Android Studio includes several substantial test frameworks, including jUnit for unit and integration testing, and Espresso for UI testing. Android Studio's jUnit implementation supports both simple in-place testing using custom-made mock code for the android API, and more sophisticated on-device testing either on emulators or on hardware. Frameworks for UI testing are also included, with Espresso being the standard option for testing in a single app. Support for the Gradle build system allows for the automation of tests whenever a build is made. We also intend to perform much of our frontend/backend integration testing using the Android Studio-integrated tools, particularly jUnit.

Our backend is being built primarily in Python3, which has an excellent built-in unit testing framework called unittest. Unittest provides similar semantics and capabilities to jUnit, and should be adequate for backend testing needs. If needed, we can also build tests using the MySQL Test Framework, if unit testing is desired before server-side database logic is complete.

Our unit testing will be geared towards ensuring both proper behavior of procedures *and* congruency between documentation and actual behavior. This is especially critical due to the lack of continuous integration deployment between our frontend and backend. Correct behavior of API calls is especially critical in this case, as implementation that disagrees with expected results could cause serious and hard to resolve issues with system integration. To help resolve this, we plan to set regular, incremental goals for API completion and test each stage thoroughly with the frontend before moving to the next. These tests will be automated on the frontend using the Gradle build system. On the backend, python unit tests will have to be run manually before each push, and practices and policies will be put in place to ensure that this happens consistently. Each coder will be responsible for writing comprehensive unit tests for their own code, ideally before the code itself is written. Additional unit tests may be added by others as time or circumstance permits, particularly when fixing bugs -- Any bugfix should be accompanied by a counterpart unit test designed to test against regressions.

System testing will consist of two primary parts, integration testing and UI testing. Integration testing will be achieved primarily from the frontend, using jUnit on emulators and live devices to test behavior of the full system. UI testing is a frontend exclusive task as well, and is achieved through the Espresso framework in Android Studio. Thorough UI tests are a useful tool to ensure that UI behavior is consistently correct, and that said behavior is not unintentionally affected by changes to code elsewhere. The system and UI tests will be delegated to individual team members to complete as these components come together. Like the client-side unit tests, integration and UI tests will be automated using the Gradle build system, to ensure that they are run before every push and/or code deployment. CircleCI will be used to run all tests as they are pushed to Git.

Usability testing is planned for later in the quarter, when a more complete and stable product UI has been established. Several of our members have friends that have offered to assist in this step, including several HDCE majors. At test time, testers will be asked to perform several tasks (such as managing their inventory, submitting a trade request, or changing their settings), and their process will be observed. After testing, comments on the user experience and any difficulties encountered will be solicited and compiled, then used to update the UX for the next round of testing. Only a few (possibly two) rounds will likely be run due to limitations in time.

Together, the individual components of this test strategy should cover most major testing concerns. Android Studio has excellent testing facilities that cover much of our needs, especially on the frontend, and we have an excellent pool of UX testers to work with for usability testing. The weakest spot for this test plan is on the backend, with unit tests that have to be manually run and limited facilities for integration testing beyond what python's unittest provides (though those facilities should be more than sufficient). This is caused primarily by the lack of a single development environment on the backend -- while the frontend is to be coded almost entirely in Android Studio, the backend includes multiple different technologies, and Python additionally does not have a free IDE that compares to Android Studio in quality. While continuous integration and build utilities exist, they can be a challenge to set up and make interoperable in a limited timeframe.

For bug-tracking we will (of course) be using GitHub issues. Additionally, we plan to use GitHub issues to track major feature releases and milestones. Issues has excellent facilities to help delegate and assign work on specific parts, and using it to track features and milestones will help keep major releases on-track.

Documentation Plan

The product will feature an in-app link to our developer website and our client website. The developer website will describe the implementation of the app and the process we took to get there. It also will provide access to the open source code itself. The client website will primarily feature a demo video of how to use the app. The video will instruct the user on how to perform critical tasks such as changing their settings, posting an item, and sending a trade proposal.

Coding Guidelines

We plan to use the Google Java and Python style guides for our frontend and backend, respectively:

Java: https://google.github.io/styleguide/javaguide.html
Python : https://google.github.io/styleguide/pyguide.html

In order to enforce adherence to these guides, we will have weekly code reviews, in groups separated by subteam (frontend team and backend team). We will review the code together to ensure all members of the team are following the guidelines and correct any mistakes which have been made in the process.

Design Changes and Rationale

We decided to store URLs to images on our database server, which will link to the actual images which we will store on the API server. We originally considered storing these as a blob in the database, but ultimately decided that doing so would be make our REST calls too slow. Storing links to the images makes our API calls lighter, since we can just pass string links in our JSONs rather than Bitmap blobs, and means we can load images as they are needed by simply fetching them from the given links. The only downside is having to make more calls from the client side to fetch the images, rather than getting them along with the rest of the user data.

Our schedule has changed somewhat. Our relatively empty week prior to feature complete release will be spent implementing searching and trading, and our Beta Release will not have trading complete, only inventory management. This is due to time constraints. We were not able implement things as quickly as we hoped, and we will not be able to get as far as we thought prior to the Beta Release.

When initially writing this document, we had not planned on using any CI or automated build tools because we did not think they were required. We then learned that they were, so we decided to use CircleCI for continuous integration due to it being the simplest tool to set up from our perspective.