Date-a-Dog
Alexis Allen (laetaku)
Amanda Loh (cosy16)
Amarpal Singh (amarpal)
Anmol Jammu (jammua)
Hassan Abdi (habdi)
Hugo Salazar (hsalazar)
Lauren Wolfe (wolfela)
Raag Pokhrel (raagp)

Software Design Specification

System Architecture

Modules and interfaces between modules

Date-a-Dog consists of three major components: the software running on an Amazon AWS server, an Android app, and a web app for shelters.

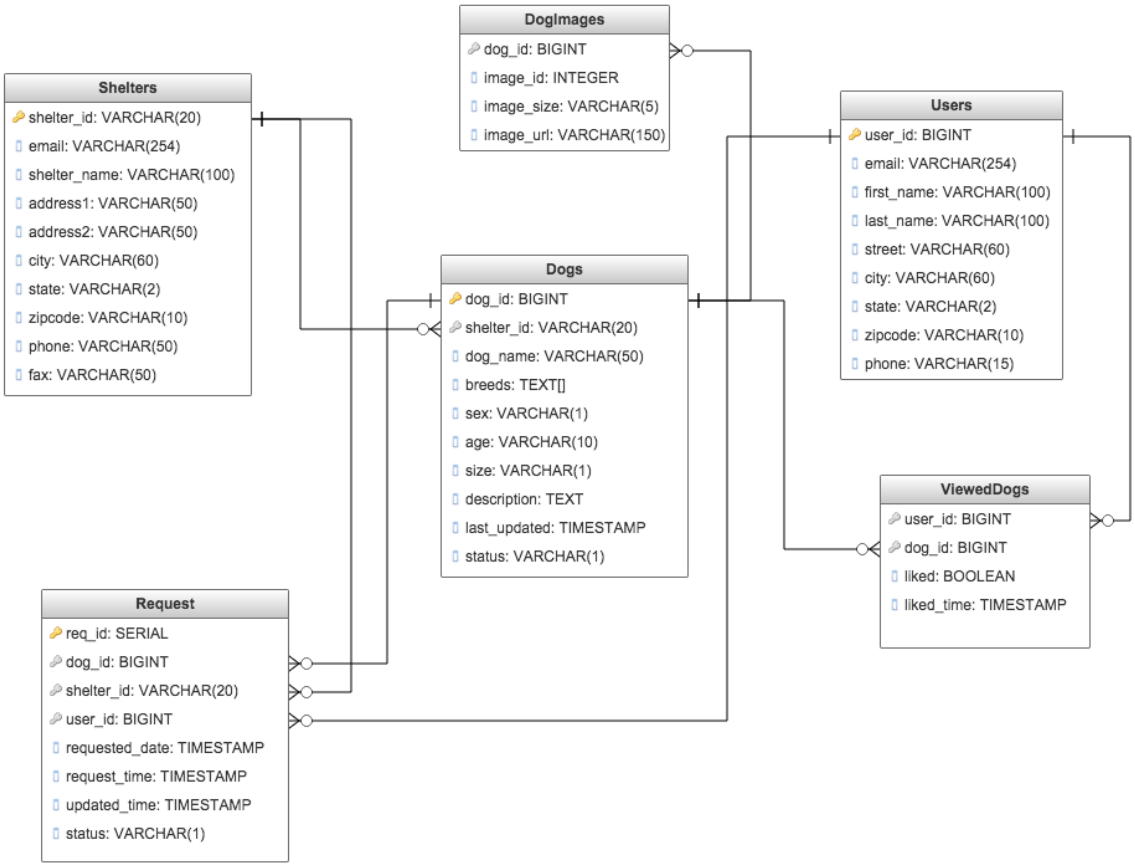
The frontend components (the Android app and the web app) must communicate with the backend whenever information about dogs and date requests is retrieved or updated. This data exchange between the components is isolated a single class in each of the components. These classes are DaDJavaAPI in the Android app, DaDJSAPI in the web app, and RestAPI in the server. See the “Overall Architecture” diagram below for a visualization. Data exchange between these is always done using REST calls and JSON to maximize flexibility and language independence.

By modularizing communication between components, the components can be developed more independently and with greater encapsulation. Moreover, this strategy allows us to develop and test each component in parallel since the API for interfacing with the backend can be set to return mock data while the backend is being implemented.

Database schema

Date-a-Dog uses data pulled from the Petfinder database. This data is retrieved using the DaDUpdateDaemon (see UML diagrams below). The data is then cached on our server using a SQL database.

Our database uses the following schema:



These tables are updated once a day by the daemon with data pulled from Petfinder. They are also updated as requests are made to the server by the Android and web apps.

Design decisions

During the initial design of the system architecture of the Date-a-Dog application, several different models were discussed and some differing ideas were compared. In the end some ideas had to be scrapped due to concerns about development time, latency and design. Here are some of the major ideas that did not make the final cut:

1. Individual tables in our database

The initial plan was to have individual tables in our database for each user. Titled with the user's name, the table was to keep an ordered list of the dogs that the user had viewed. The point of this table was to allow the user to continue wherever she had left off if the app were to crash or if the user were to log off.

While this idea made sense to us initially, it soon fell apart under closer scrutiny. This was not a smart design choice from the standpoint of space, with poor scalability. Since our app relies on regular caching of information from the Petfinder API, any time a dog was deleted from the Petfinder database, we would need to query our own database and delete the dog from every table in which it appears. While maintaining such an order might have appealed to users, the complexities of resulting operations were simply not worth such a minor benefit.

Alternative chosen:
One table that houses all the users and the dogs that they have liked/disliked.

While this table forces us to forego the added functionality of having an extended history for each user, it is much more scaleable and keeps our database design simple. It also allows for easier updating of our dog catalog without necessitating an individual dog catalog.

2. Mobile application for shelters

The original idea was to have a mobile application for shelters. In order to keep everything, uniform with our development of the mobile application for daters, we had intended on developing a mobile app for the shelters as well.

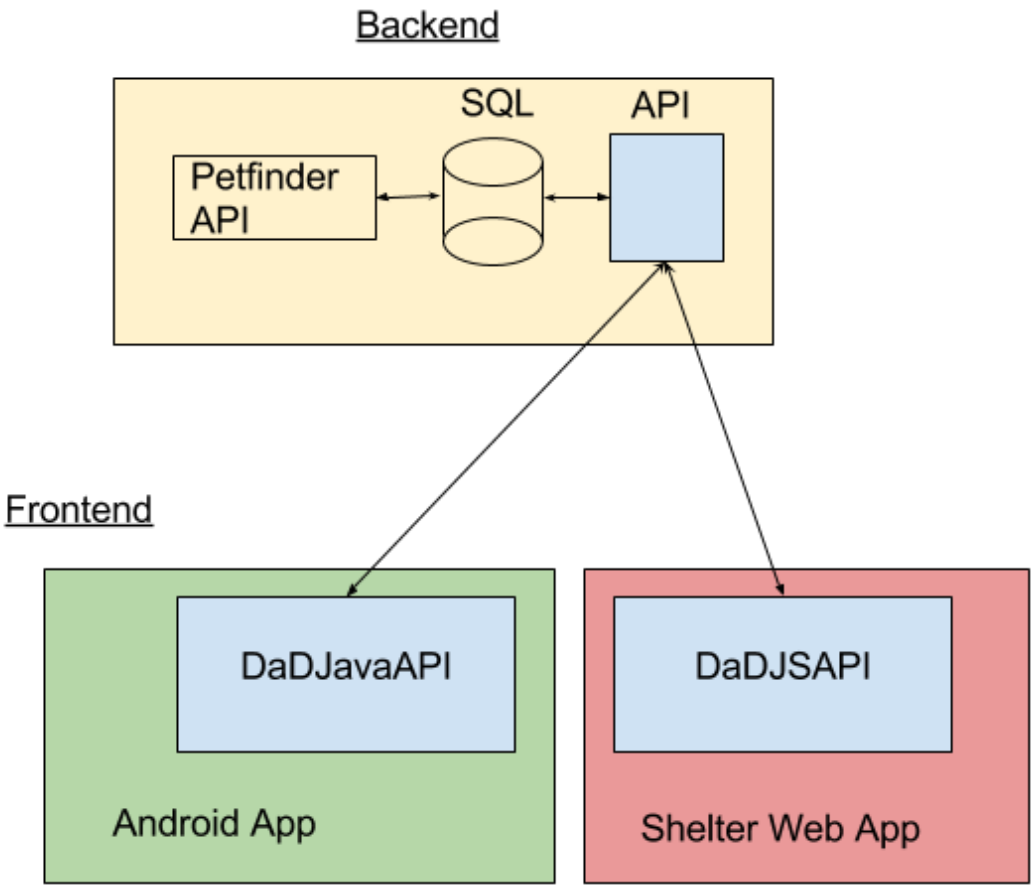
While this idea might have kept things simple for our development process (and our schedules), it did not make sense from an end user standpoint. An application that is meant to be used at work, by an employee, should run on a desktop computer.

Alternative chosen: A secure, login-driven, web application for shelter employees.

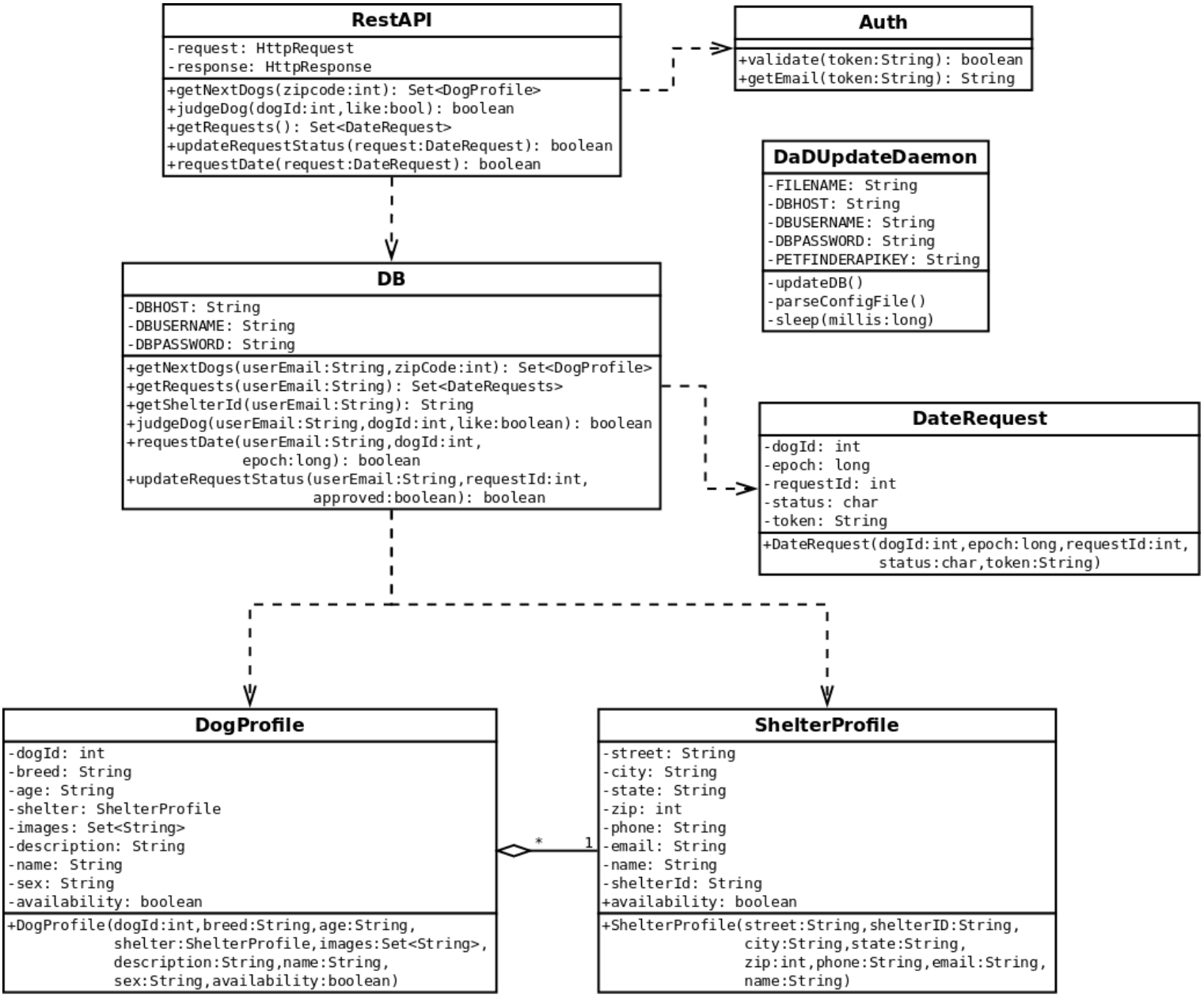
While this alternative adds yet another component to our application, which means more attention to design and the need to learn more skills, it makes more sense from the point of view of the experience of using the application. It is much easier for the average person to perform the functions necessary for a successful dog date, while operating a desktop computer.

Diagrams

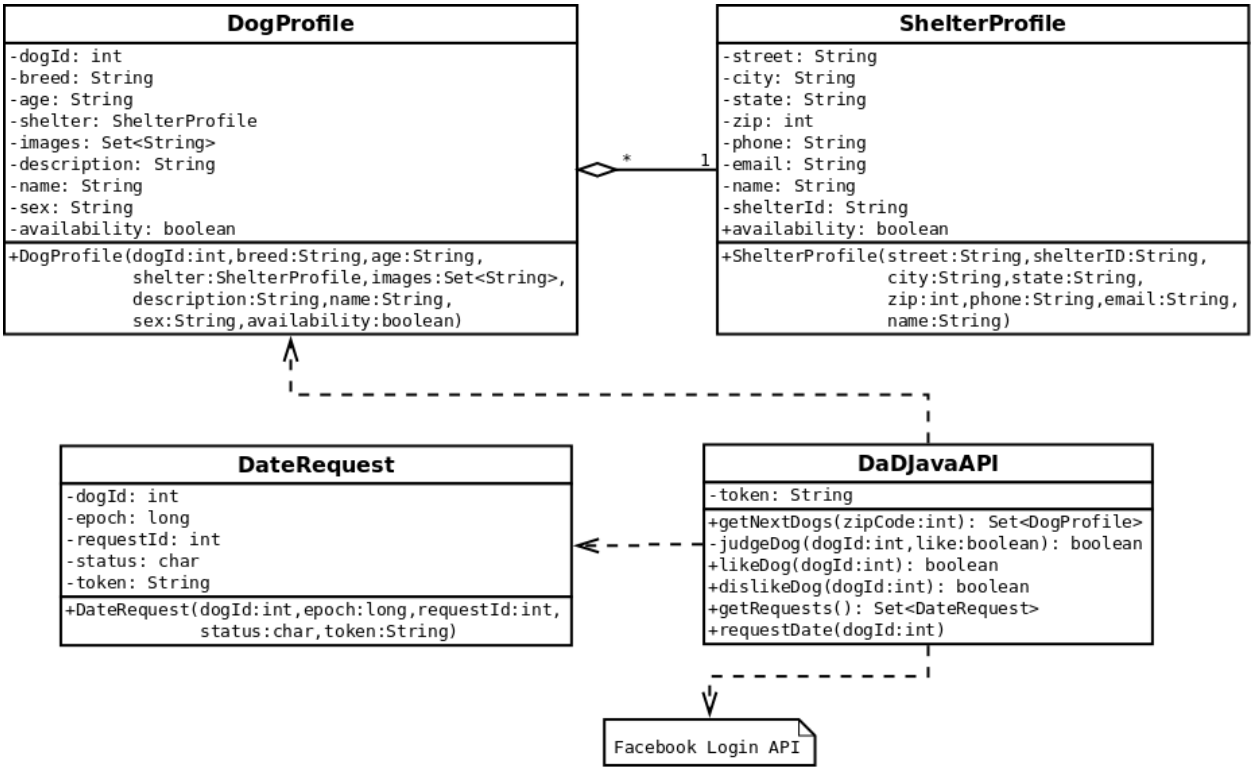
Overall System Architecture



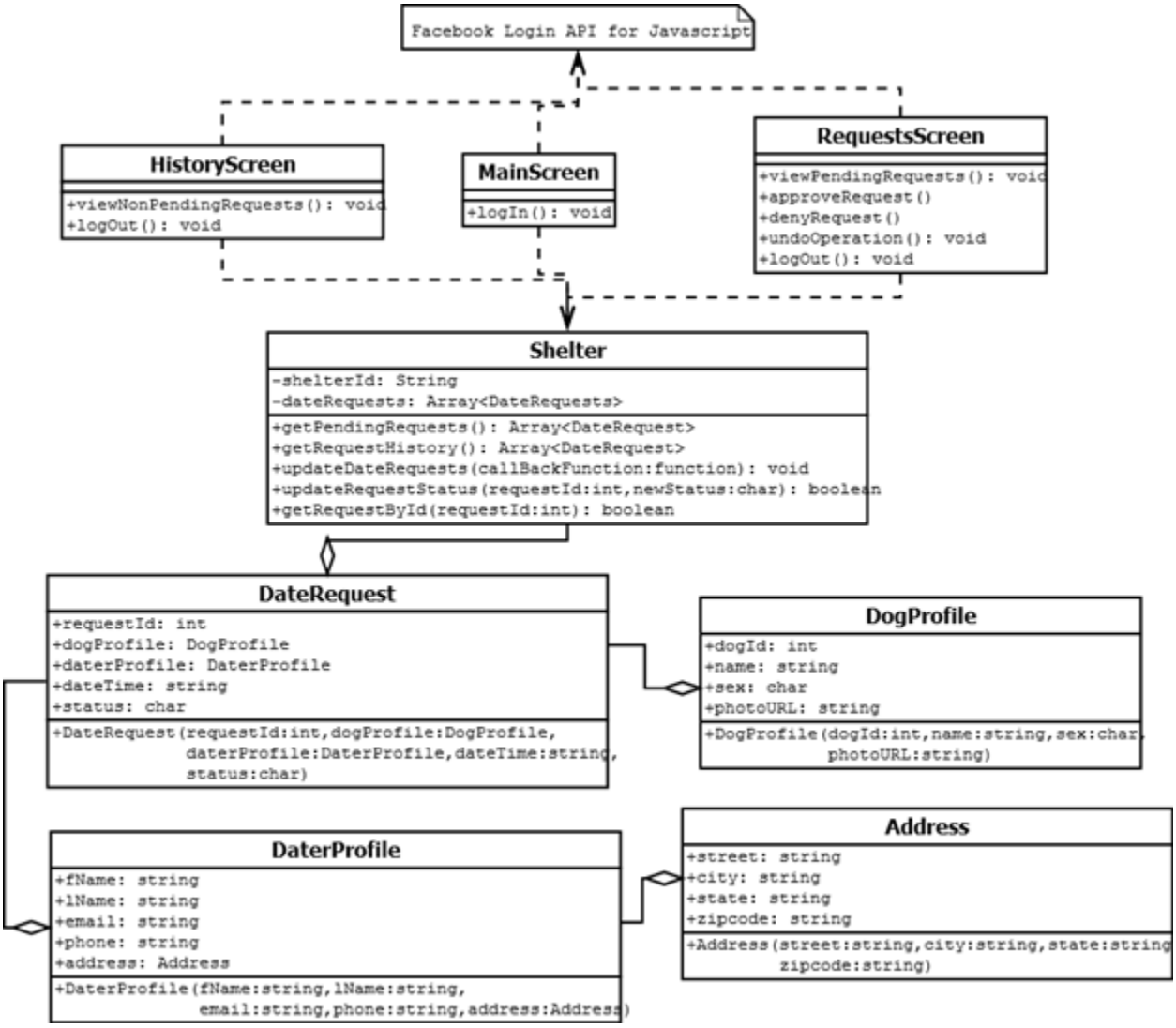
Backend (Server) UML Class Diagram



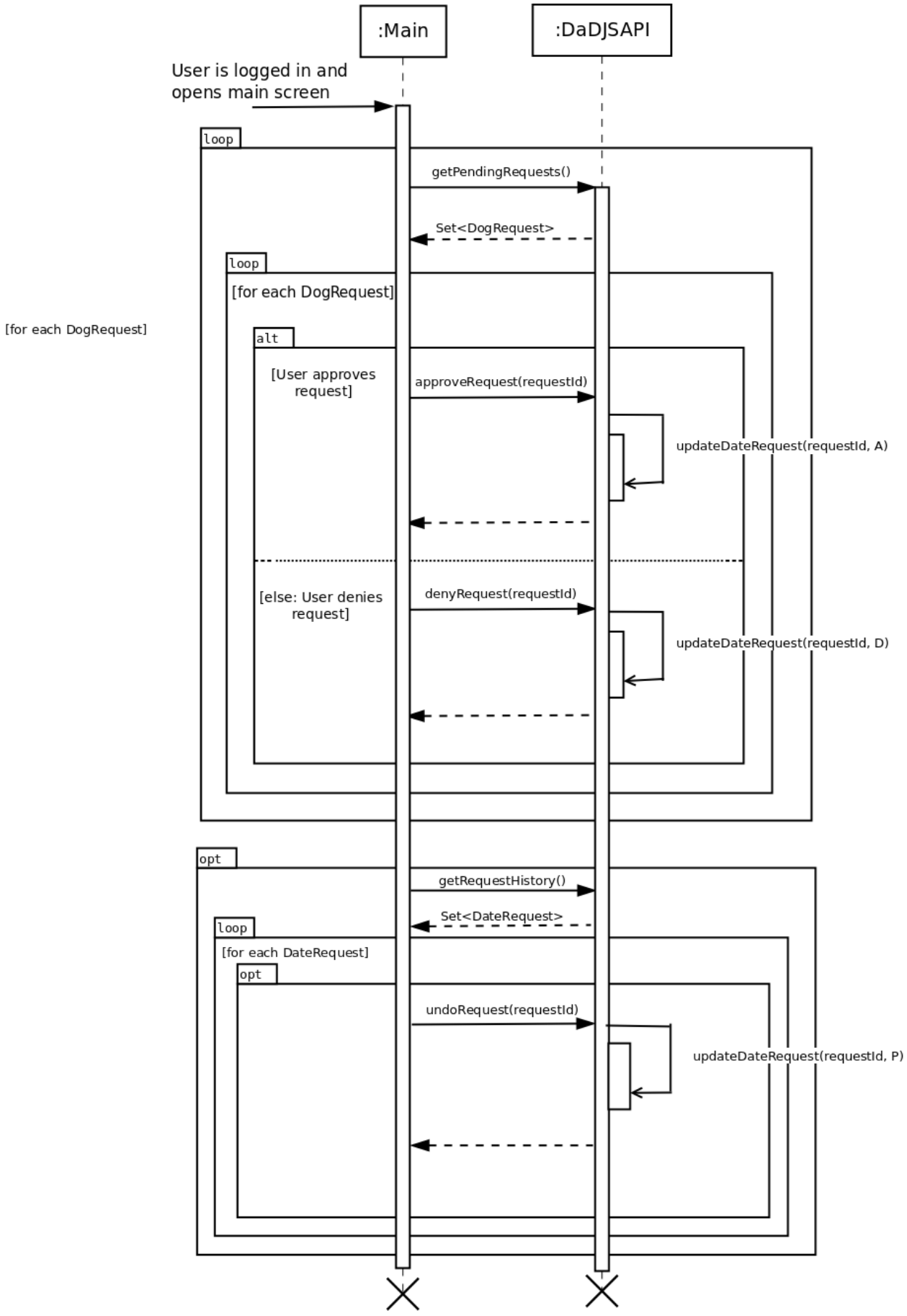
Android App UML Class Diagram



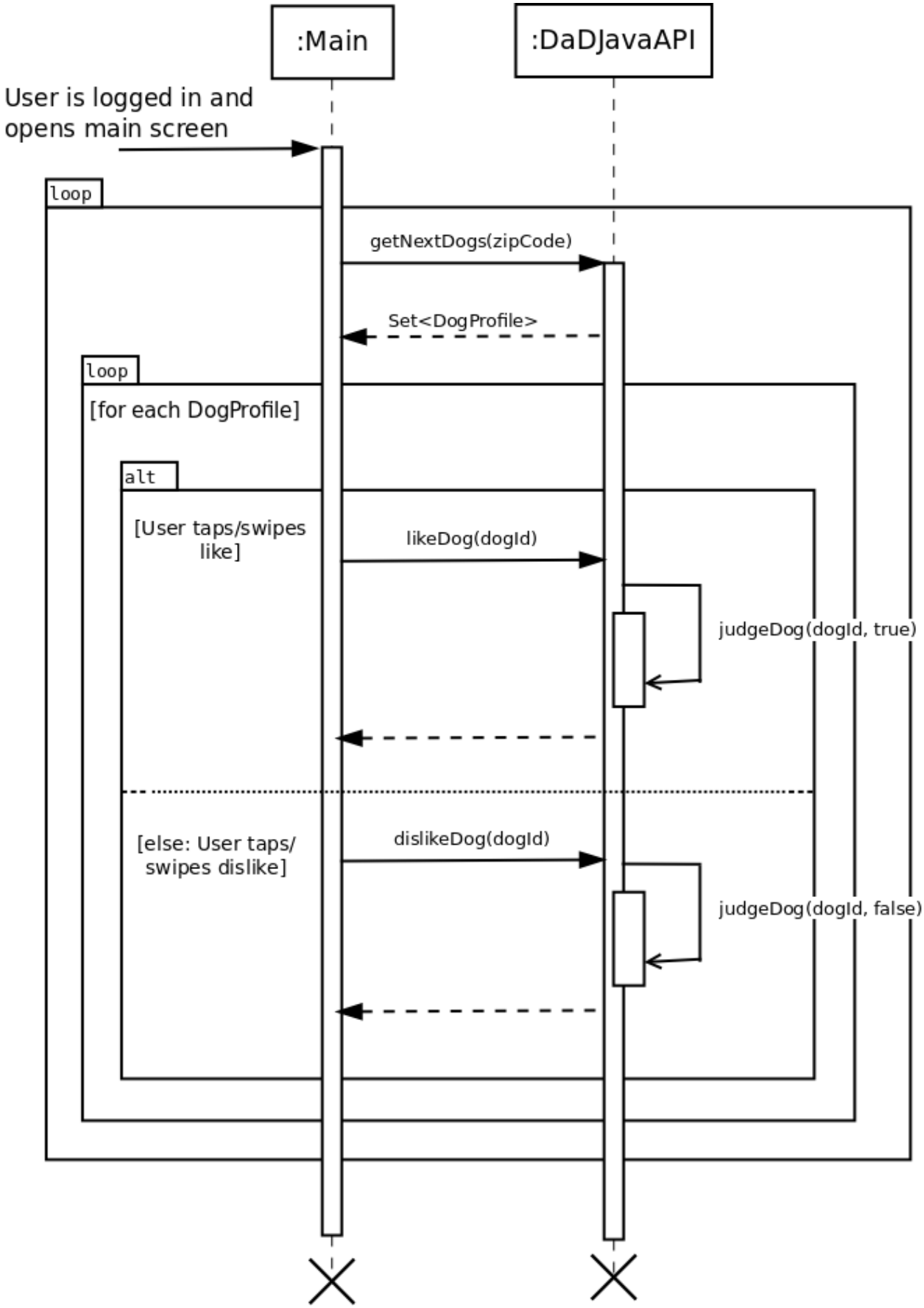
Shelter Web App UML Class Diagram



UML Sequence Diagram for Managing Requests in Shelter Web App


:Main
:DaDJSAPI
User is logged in and opens main screen
loop
getPendingRequests()
Set<DogRequest>
loop
[for each DogRequest]
[for each DogRequest]
alt
[User approves request]
approveRequest(requestId)
updateDateRequest(requestId, A)
[else: User denies request]
denyRequest(requestId)
updateDateRequest(requestId, D)
opt
getRequestHistory()
Set<DateRequest>
loop
[for each DateRequest]
opt
undoRequest(requestId)
updateDateRequest(requestId, P)

UML Sequence Diagram for Viewing/Liking Dogs in Android App


:Main
:DaDJavaAPI
User is logged in and opens main screen
loop
getNextDogs(zipCode)
Set<DogProfile>
loop
[for each DogProfile]
alt
[User taps/swipes like]
likeDog(dogId)
judgeDog(dogId, true)
[else: User taps/ swipes dislike]
dislikeDog(dogId)
judgeDog(dogId, false)

Process

Risk assessment

After reviewing the program architecture and system specifications, the following risks were identified as being major hurdles to the development of Date-a-Dog:

1. Privacy and security

Date-a-Dog collects user information for two types of users: daters and shelters. The application authenticates users with Facebook information and also requires daters to fill out a detailed personal information form that will be submitted to shelters for approval. Shelters, on the other hand, will need to keep track of dog date requests, including requested dates and dogs, as well as personally-identifying information. Security and the privacy of this information needs to be a major priority for this application.

- The likelihood of security issues becoming an issue in the project remains high, as the application actively collects personal information and stores it on the Date-a-Dog database.
- This risk will have a high impact if we are not able to sufficiently address security concerns. User information such as names and contact information could be leaked to malicious hackers.
- The very nature of security, and the fact that there is no quantitative way to measure it, is what leads us to believe in the gravity of this issue.
- We will need to rely on existing APIs and frameworks and see how they are implemented in currently existing applications in order to combat problems as they arise. Relying on established APIs will reduce the amount of code that we have to review for security issues.
- We plan on writing automated tests in order to check for security bugs or information leaks. These will be integrated with the unit tests and system tests that are created. After the beta release, we will also do a security review to see if there are any glaring security issues.
- In order to mitigate any security problems, should they occur, we will minimize the amount of private information the user has to submit to the shelter via the phone app, letting the user supply this information in person when they go to the shelter.

2. Relying on external data

One of the most prominent features of Date-a-Dog is its ability to query the Petfinder API and automatically download dog information to display to the user. This information includes details such as the name of the dog, age, sex, shelter location, weight, etc. While the existence of this API is making it possible to build a large, diverse database of dogs, it does come with the risk of ingesting data that is outdated or simply incorrect. Since we cannot be sure of the level of QA that has gone into the construction of the Petfinder data, our app runs the risk of displaying incorrect information.

- The likelihood of this happening is somewhat likely, given the information we have extracted from Petfinder as well as some of experiments that we have run by pulling example data from the Petfinder database.

- We identify this is a medium to high level risk since data accuracy is a primary goal of this application to ensure user satisfaction. Date requests are useless and even irritating for users if the dog they'd like to date is no longer available.
- We have made these estimates based on some sample experimental calls that were made to the Petfinder API. Based on queries run on 1000 dogs, we discovered that approximately 22% of the dogs on the Petfinder database had not had their profiles updated in the last 45 days.
- In an effort to reduce the impact of outdated information, we are working on making API calls that will only focus on dogs with recently updated profiles.
- In order to detect the problem, we will need to check a subset of the profiles which have not been detected recently and see if those dogs are still available at the shelters.
- If this should become a problem at an incapacitating level, we will perhaps need to start identifying dogs who may possibly not be available anymore, or send a notification to the shelter that they need to update their information.

3. Android Material Design UI

Date-a-Dog is a very interactive application, featuring swipes and different views and forms as well. There are a lot of user interactions that require animations to keep the app interesting and fun to use. These animations may be difficult to create, but they are an essential part of Material Design. Thus, we think that there is a risk that using a Material Design UI could take developer time away from more important tasks.

- Given the user-event driven nature of the application we see a medium chance likelihood of this risk becoming an issue.
- We identify this as a medium risk, since it does not affect the application negatively, but also reduces the quality of the product if not dealt with properly.
- This risk is based on our recent experience with the Android UI documentation. There are a lot of new API calls and classes required to create a Material Design UI, and this will certainly require some time to get familiar with.
- Currently, our plans to combat this risk consist of trying to make an internal schedule with checkpoints, independent of class requirements, in order to keep us on track.
- Not meeting our planned checkpoints and a weekly analysis of work done will be indicators that front end design is consuming too much time.
- Our mitigation plan, in case this problem should occur, is to cut back on features and simplify the application. We can ultimately drop the Material Design UI and use a simple Ice Cream Sandwich or Gingerbread UI if we need to.

4. Scheduling issues

Despite our awareness and plans to write features into our application and complete it on time, we all have busy schedules and full course loads to maintain. Based on our experiences in the past two weeks, it appears that maintaining our school schedules might be a big threat to or work on the application. Thus, we see scheduling issues as being a big risk that might affect the completion of the project.

- The likelihood of scheduling problems becoming prevalent are high given the number of people and schedules on our team.

- If scheduling becomes a problem, then it will have a high impact on the success of the project.
- This is based on team member experience from the last three weeks.
- We are actively working to combat this problem. We have collected team member specifications on availability. We also have a Slack account set up and communicate extensively regarding the project. Slack also allows for working remotely so we do not always need to be face to face in order to get work done.
- Our mitigation plan, in case scheduling should be a problem is to start cutting features, and in fact, our application milestones are currently structured in such a way as to put functionality over extensive features.

5. Coordinating development on the web and Android apps

There is an added level of complexity to the Date-a-Dog application because it features both a mobile app and a website as well. The mobile application is the user interface for people wishing to request time with a dog whereas the web interface will be for shelters wishing to look at date requests and approve/deny them. Given these two features and their equal importance in making a successful app transaction, we run the risk of not being able to efficiently develop both the site and the mobile application. The risk we run is of either spending too much time on one interface and not the other, or worse yet, going back and forth between the two and not properly developing either.

- The likelihood of this risk formulating into an issue is somewhat high as we have already experienced during the design phase.
- If this issue were to arise during the development process, it would have a high impact on our application success as Date-a-Dog relies heavily on both interfaces to complete a successful transaction.
- We are basing this on current experience with our meetings, where most of our focus has gone on the mobile application with very little thought given to the web interface.
- In order to combat the problem, we are incorporating the website into our design specifications as a major feature.
- In order to detect the problem and stop it from escalating, we will start documenting our development time to which areas are being worked on. Implementation of SCRUM stand up meeting should also help stop this issue from occurring.
- A mitigation plan, should this problem prove to be difficult to overcome, is to create a smaller team in charge of the website, to ensure that it does not get left behind.

Project schedule
Development will roughly follow the schedule below:

October 14th:

- Complete Software Requirements Specification (10/14/16)

October 15th to October 21th:

- Set up GitHub repo and development website
- Design the Android app, web app and server backends
    - These designs will be done simultaneously since the Android app takes as input the data output by the server.
    - Create UML class diagrams and sequence diagrams.
- Complete Software Design Specification (10/21/16)

October 22nd to October 28th:

- Set up the app and server development environments
  All group members will,
    - Initialize their development environments (e.g. Android Studio).
    - Become familiar with using the build tools, like Gradle and GitHub.
    - Be able to build a zero feature release of the component they are working on.
    - Become familiar with writing unit tests in their environment.
- Begin implementing RestAPI in server
- Begin implementing and testing server DB daemon
- Zero feature release (10/28/16)

October 29th to November 4th:

- Begin implementing and writing unit tests for Android model classes
- Simultaneously, UI programmers will create Android “Activities” (UI screens)
- Create HTML and Javascript for web app pages
- Implement and test internal server classes

November 5th to November 11th:

- Run usability tests with Android UI and completed portions of Android and web apps
- Finish implementations of Android model/controller classes and test extensively (unit test and system tests)
- Begin styling web app with CSS
- Finish implementation of server classes and test extensively (unit test and system tests)
- Beta release (11/11/16)

November 12th to November 18th:

- Improve UI based on feedback and tests
- Fix problems found in tests
- Feature-complete release (11/18/16)

November 19th to November 25th:

- Perform more integration tests to test all three components running simultaneously
- Begin work on most useful bells and whistles features

November 26th to December 2nd:

- Test and finalize existing classes

- Finish implementing bells and whistles and test them
- Release candidate (12/2/16)

December 3rd to December 9th:

- Final release (12/6/16)

Team structure

Team members chose the areas that they would most like to work in. We have a central project manager, as well as a frontend and backend team.

The team structure for overall development is as follows:

Front End

- UI design: Amanda, Lauren, Alexis
- Android app programming: Alexis, Hassan, Amarpal, Hugo, Amanda
- Web design: Raag, Hassan, Amanda, Alexis
- Web programming (XML, CSS, JavaScript): Raag, Amarpal, Lauren, AJ, Hassan, Alexis

Back End

- AWS (coding/architecture/database): Hugo, Raag, Amanda, Hassan, Lauren
- Rest API: Lauren, Amarpal, Hugo

Team members were assigned the following tasks for the SDS assignment:

System Architecture: Hassan, Raag, Amarpal, Hugo, Lauren, Alexis, AJ

- Assumptions
- Class diagrams
- Sequence diagrams

Process

- Risk assessment: All team members will identify risks
- Project schedule: AJ
- Team structure: Amanda
- Test plan: Amanda
- Documentation plan: AJ
- Coding style: AJ

The system architecture section is assigned to everyone since it requires substantial collaboration. Nonetheless, individual sections of the architecture have been assigned based on who will eventually implement those sections.

Tasks for the SRS assignment were previously assigned as follows:

- Product description: Hugo, Raag
- UI diagrams: Amanda, Alexis, Lauren
- Use cases: Amarpal, Hassan
- Process description: AJ

Communication

Team meetings are held every Tuesday at 9:30 a.m, and weekly status reports and summaries of the meetings are available in a shared Google Drive folder.

Slack is used for most communications. We have various channels set up for scheduling, meeting agenda, and various tasks that team members are currently working on. In particular, we have a general discussion page for overall inquiries and information, a scheduling page to talk exclusively about meeting times, and channels for architecture, each sub-team, and weekly assignments.

Test plan

We will be performing several types of tests, including unit tests, system (integration) tests, and usability tests. These tests will be performed on all three major components of Date-a-Dog: the Android app, the web app, and the server. Details on these tests are below.

Unit test strategy

- Unit tests will be created for Android model/controller classes, the Javascript code in the web app, and the backend classes. JUnit will be used to make the test classes for the Android app. Informal test methods will be written to test the Javascript and backend software.
- All team members will be writing unit tests for any classes that they write.
- Gradle and Travis CI will be used together. This will allow us to run unit tests at every commit to ensure that we always have a working build.

System test strategy

- System tests will test how components of Date-a-Dog work together. These tests will make sure that the output from one component (e.g. the backend) can be used as the input for another component (e.g. the frontend). These tests are important to ensure that there are no misunderstandings about how components of the project fit together.
- These tests will be added as test cases which may be run with a script. Group members associated with each component under test will be responsible for writing the system test.
- The script will be run after group members assigned to a component have completed the task assigned to them. This will ensure that the components are well-integrated early in the process.

Usability test strategy

- Usability tests will test the behavior of Date-a-Dog components from the perspective of users.
- These tests will be done informally by group members testing each other's work. For example, a group member who adds swipeable cards to the Android app will have the usability of this feature tested by other group members who are not familiar with the implementation. Users other than group members will also be used to test components as they become more developed (near the beta release).

- Browser compatibility will be tested after the web app is deployed. This will be done at first by hand using a free browser tester such as the browser sandbox by turbo.net. If possible, this will later be automated using Selenium.
- These usability tests will be conducted after every assigned task is completed (e.g. after group members finish creating a page in the web app).

Adequacy of test strategy

- Our testing strategy tests individual modules in a component of the project (e.g. individual modules in the Android app). It also tests the component as a whole (e.g. the entire Android app). And finally, it tests the usability of Date-a-Dog as a whole by testing how the three major components interact with one another. This allows us to test Date-a-Dog at multiple scales. Moreover, our usability tests ensure that the end-user experience is pleasant and easy to use.

Bug tracking

- GitHub Issues will be used to track bugs. Team members will be required to submit a detailed bug report for every bug that is found. The following instructions should be referred to before submitting a bug report: https://developer.mozilla.org/en-US/docs/Mozilla/QA/Bug_writing_guidelines. However, it does not need to be followed exactly, especially for a project of this size.

Documentation plan

For users, two types of user documentation will be available:

- Daters will see, in the Android app, a quick overview of the basic flow and common actions.
- Shelters will see, in the web app, a quick overview of the requests page and how to use it.

Developer documentation will be on the project's developer website (https://date-a-dog.github.io/). Already, instructions on how to set up the development environment (with Android Studio) have been posted on this website, and more resources/documentation will be posted as they are needed.

The Android app's Java code itself will also include Javadoc documentation. Using the Javadoc tool, these comments can be turned into browsable documentation pages.

JavaScript will be documented with simple code comments.

As a bells and whistles feature, we may add help screens accessible from a menu to the Android app and web app.

Coding style guidelines

The three main programming languages we will be using are Java for the Android app and JavaScript for the server and the web app.

For Java, we will use the Google Java Style Guide (https://google.github.io/styleguide/javaguide.html). This will be enforced via a Git pre-commit hook which verifies that all source files follow the style guide before they can be committed. This pre-commit makes use of the following tool: https://github.com/google/google-java-format. We will also use an IntelliJ plugin/preferences file that automatically formats code to the Google Style Guide.

For JavaScript, we will use the Google JavaScript Style Guide (https://google.github.io/styleguide/javascriptguide.xml). There is no convenient tool to automatically format code to the style guide before commits, so team members will instead be instructed to check the style guide before they commit any code.

Design changes & rationale

1. Database Schema**:**
The entire database schema was updated extensively to make the database more efficient. Previously, the database was constructed by taking JSON objects and storing them in the database by separating out all their fields. This made the database extremely complicated and cumbersome to run queries on. PostGres, which manages our database on EC2, can hold JSON objects, something that we discovered after the original database was already built. Nonetheless, we redesigned the database schema and rewrote all the database queries in order to take advantage of this benefit. The end result was a more concise database, that stores data in a native format and simpler, cleaner queries which were more efficient as well.

Tradeoffs*:* The original design of the database was closest to SQL functionality while the new database was cleaner and easier to use.

2. Mobile Application Architecture:
Originally, the user interface classes were designed as “Activity” classes in Android Studio. The original development of the application saw us using Activity classes to develop the user interface for the mobile app. After research and testing the mobile application on an actual device, we realized that we actually needed to use “Fragments” in order to achieve a tabbed layout. We will need to convert all of our Activities to Fragments before the final release of the product.

Tradeoffs: The original implementation had the advantage that people had researched Activities more thoroughly and the needed interfaces had already been written. A disadvantage, however, was that our interfaces were not displaying correctly. This is an advantage of using Fragments as opposed to Activities.

3. Date-Time Objects to Epochs:
Originally, our database was holding DateTime objects and we decided to change them to epochs.

***Tradeoffs*:** While it is harder to write SQL queries that generate epochs, they are easier to instantiate than DateTime objects and do not require keeping track of time zone. We found epochs to be much simpler for our database needs.