Harihar Subramanyam, Guillermo Vargas, Giancarlo Sturla
Problem Analysis

StormSpot

Overview

Motivation

Tornadoes are the most intense storms on the planet, with wind speeds that can exceed 300mph. The key to surviving these storms is to get advance warning and react. However, since tornadoes are small compared to other storms, such as thunderstorms, they are more difficult to locate on weather radars. For precise tracking, meteorologists rely on citizens to spot storms and report them. These eyewitness reports help pinpoint a tornado's location.

Purposes

StormSpot aims to solve the problem of imprecise storm because the existing solution, the weather radar, does not achieve great enough accuracy. It will achieve this by aggregating storm reports from users.

The app has three main purposes in order of importance:

1. Show user where the storm is located.
    a. The user should be able to see the locations of storms (ex. tornado, severe thunderstorm, hail, strong winds) with descriptions (ex. golfball size hailstones) and locations down to the street level. Users should also be able to see roughly where the storm is going.
2. Let the user report a storm.
    a. When the user sees a weather event of interest, they can report it with a description and pictures. Other users of the app should be able to see the user's report. One potential problem is inaccurate or false reports – to avoid this, we shall allow users to upvote or downvote reports to argue in favor or against their authenticity.
3. Notify the user about storms near them.
    a. When there is a storm near the user, they should receive a notification even if they aren't using the app. The notification should tell them what the storm is and where it is located.

Context Diagram
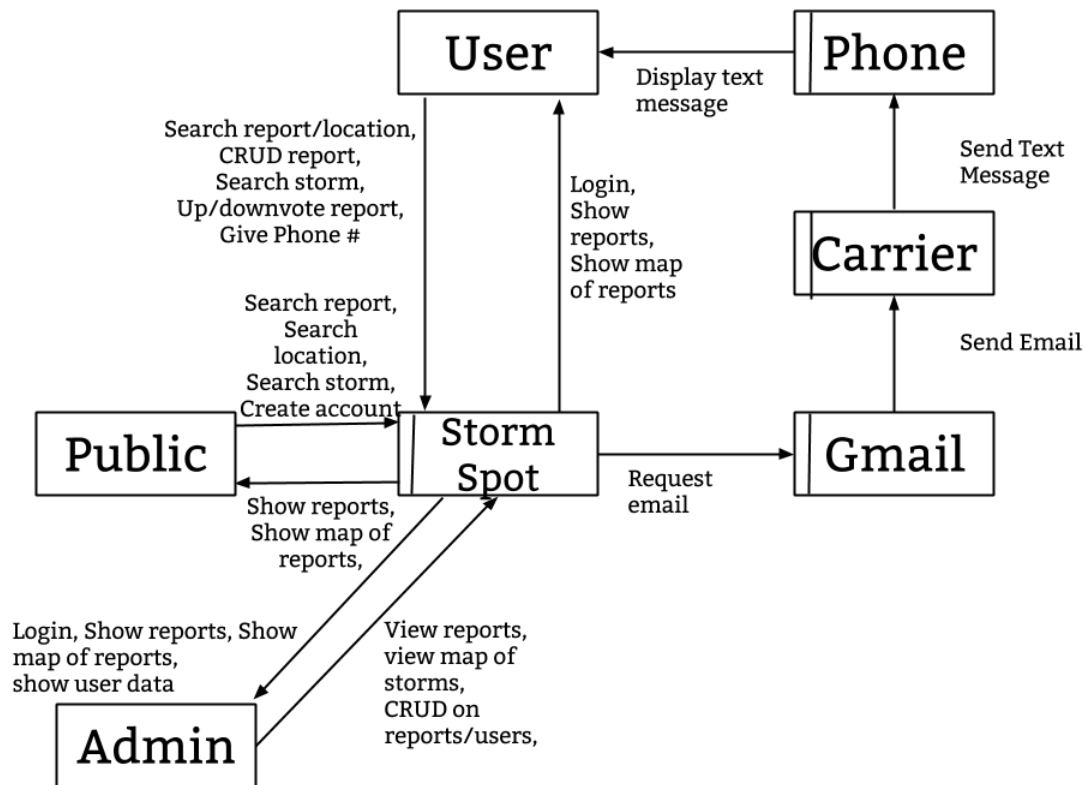
The context diagram is displayed above. There are three primary roles that interact with StormSpot.

The public (i.e. somebody who has not registered), can view the storm reports and a map indicating the storm reports.

A user (i.e. somebody who has registered for an account) can also view the reports and the map. However, a user can also create, read, update, and delete their reports. They can also upvote/downvote reports to corroborate (or testify against) their trustworthiness. Finally, a user can give their phone number so they will receive text messages when there is a storm near their location.

An admin (i.e. one of the team members) can perform the same actions as users. An admin can also create, read, update, and delete ALL reports and user data.

There are other systems that interact with StormSpot.

When StormSpot collects phone numbers from users, it aims to send text messages to the user to notify them about storms near their area. To do this, we utilize Gmail and send an email to a carrier (ex. AT&T, Sprint), which sends a text message to the

user. StormSpot cannot control the interaction between Gmail, the carrier, the phone, and the user. Unfortunately, this means there is no way to guarantee that a text message was delivered to a user. Initially, we considered using Twilio for this, but Twilio's free plan limited the number of phones we could send text messages to, and we did not want to pay for a Twilio plan. Therefore, we wrote our own solution.

Design Model

Concepts

There are three main concepts in StormSpot.

Report:
Brief Definition: User submitted information about a storm.
Motivating Purpose: "Let the user report a storm". This is the entity that captures the information that user reports about a storm.
Description: A report includes a timestamp (when the report was made), a location, an author, a number of storm types (ex. tornado, thunderstorm, wind, hail), a severity (the higher the severity, the more dangerous the storm), upvoters, downvoters, and a text description.

Storm Map:
Brief Definition: A map displaying reports, alerts, and radar imagery.
Motivating Purpose: "Show user where the storm is located". By displaying a map overlaid with reports, severe weather alerts, and radar imagery, the user can identify the storm's precise location.
Description: The app will display a map overlaid with the trustworthy reports (i.e. reports without too many downvotes) and optionally the severe weather alert data and weather radar imagery. By scanning the reports and looking at the pictures/alerts/weather radar imagery, the user can see precisely where the storm is located. By animating the reports over time, the user can see where the storm is moving.

Subscription:
Brief Definition: A user's subscription to receiving text messages when storms are near them.
Motivating Purpose: "Notify the user about storms near them". By subscribing to text message alerts when storm reports occur near him/her, a user will be notified when storms strike.
Description: A subscription consists of a phone number, a location, and a severity level. The phone number will receive a text message when a storm report with a severity greater than or equal to the severity level is reported near the location.

Data Model
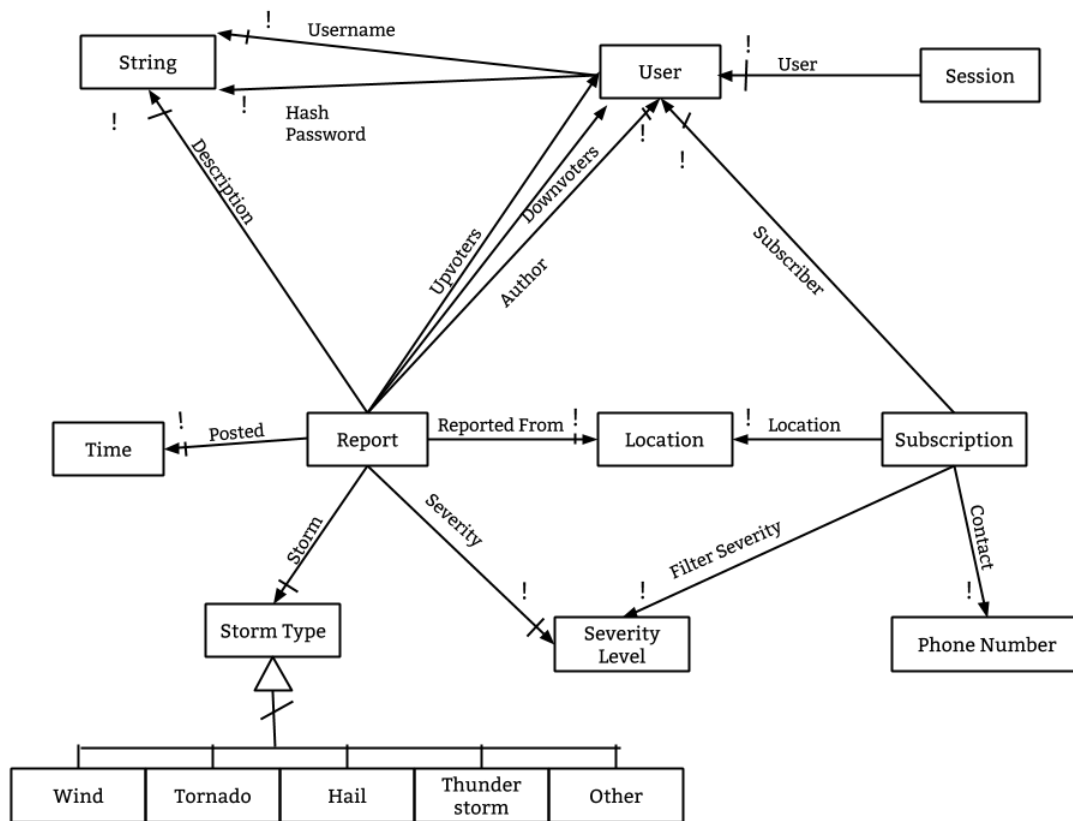
We capture the application in the data model above.

First, notice that if the public (i.e. a non-user or non-admin) use the app, there is no need to store their information in the database, so we do not incorporate the public into the data model.

A Session is associated with a singe role that cannot change. Sessions prevent users from having to log in multiple times.

Users have immutable usernames and mutable passwords.

A Report has upvoters and downvoters. It also has an author, post time, severity, reported from location, and some storm types, each of which cannot change.

There are multiple storms that are subsets of Storm Type.

A Subscription is associated with one user, which cannot change. It also has a filter severity, monitored location, and phone number, each of which can change.

Data Design
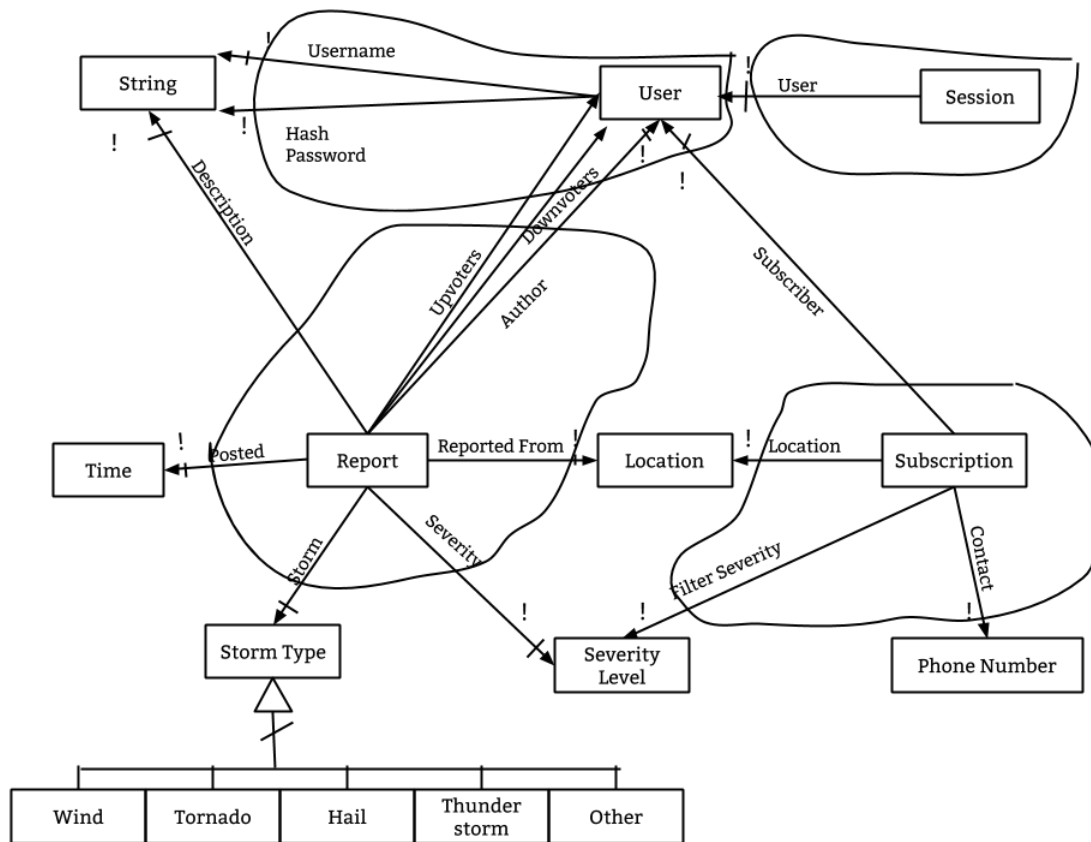
We draw the contours above, which lets us represent the data model using three MongoDB collections. They are:

User
```
    username: String
    hash_password: String
    type: "Admin" or "User"
    reputation: Number (0 for Admins)
    session: String
```

Session
```
    session_id: String
    user: ObjectId pointing to User
```

Report
```
    poster: ObjectId pointing to a Role
    posted_at: Date
    posted_from: GeoJSON point
    description: String
    upvoters: Array of Strings
    downvoters: Array of Strings
    severity_level: Number
```

```
    storm_type: Array of distinct strings where each
string is "Tornado", "Wind", "Hail", "Thunderstorm", or
"Other"
```

Subscription
```
    subscriber: ObjectId pointing to a Role
    phone_number: String
    severity: Number
    location: GeoJSON point
```

Challenges

Design Challenges

Challenge: How do we notify users about storms?
Options: Email, Text, Phone Call
Choice: We'll send text messages using Gmail.
Justification: Weather apps typically notify users with push notifications. Since we don't have the ability to send push notifications (we don't have time to develop an app), we'll use text messages, which are a good approximation to push notifications Both text messages and push notifications display a banner and cause the phone to buzz, indicating an alert.

Data Design Justification

We've separated our data into four collections. This way, we decouple information that is likely to be retrieved separately. It does not make sense to group the Subscriptions and the Reports together. Since we may seek to retrieve the reports independent of the User (ex. Get all reports where the storm type is a tornado, get all the reports near a given location), we segregate the Users from the Reports. Similarly, since we often query the Subscriptions independent of the Users (ex. get all the Subscriptions near the given area so we can send them text messages about the storm in the area), we segregate the Subscriptions from the Users. If we want to retrieve the Subscriptions or Reports for a given User, we can query by the subscriber or poster, respectively. Therefore, the data design is effective in decoupling our different kinds of data, but also quite efficient.