Harihar Subramanyam
StormSpot Problem Analysis

Overview

Motivation

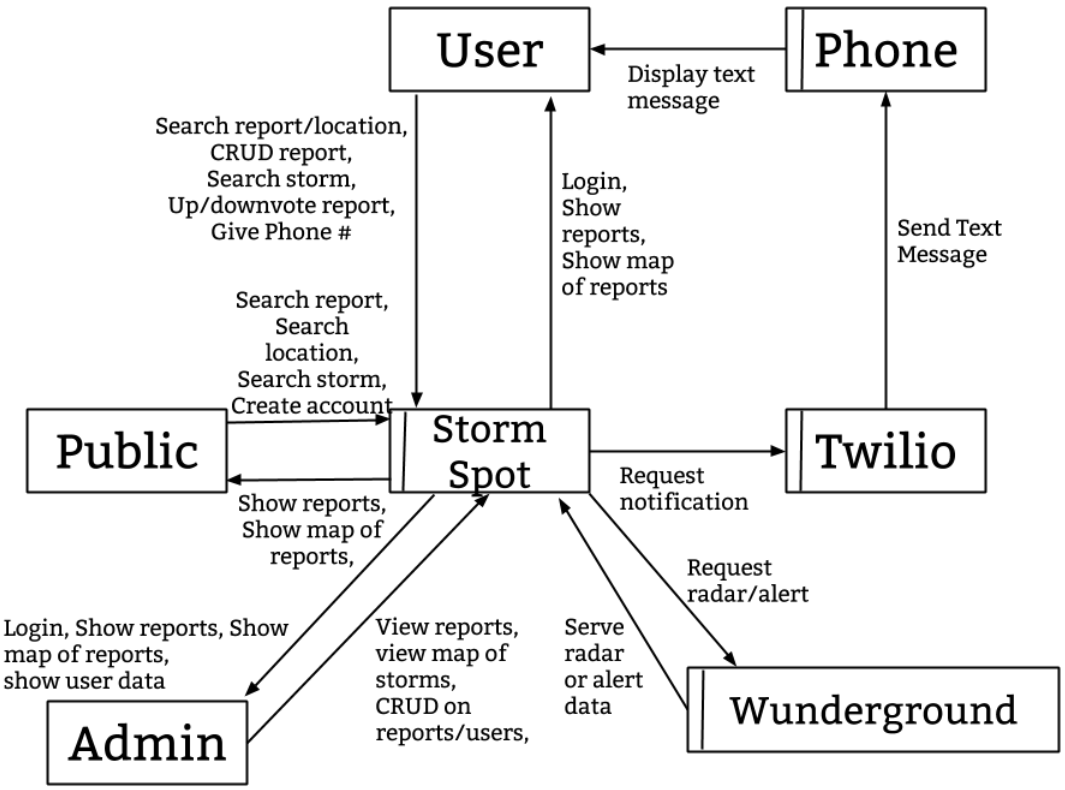
Tornadoes are the most intense storms on the planet, with wind speeds that can exceed 300mph. The key to surviving these storms is to get advance warning and react. However, since tornadoes are small compared to other storms, such as thunderstorms, they are more difficult to locate on weather radars.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

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:

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 a 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.
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 three systems that interact with StormSpot.

Twilio is a service that sends text messages. 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 achieve this, StormSpot sends a request to the Twilio.

Twilio then sends the message to the user. Then, the user's phone (another system) displays the text message to the user. StormSpot cannot control the interaction between Twilio, the phone, and the user. Unfortunately, this means there is no way to guarantee that a text message was delivered to a user.

The third system is Weather Underground (Wunderground). Wunderground provides an API to retrieve alert and radar data. StormSpot retrieves this information and overlays it on the map displayed to the user.

Design Model

Concepts

There are three main concepts in StormSpot.

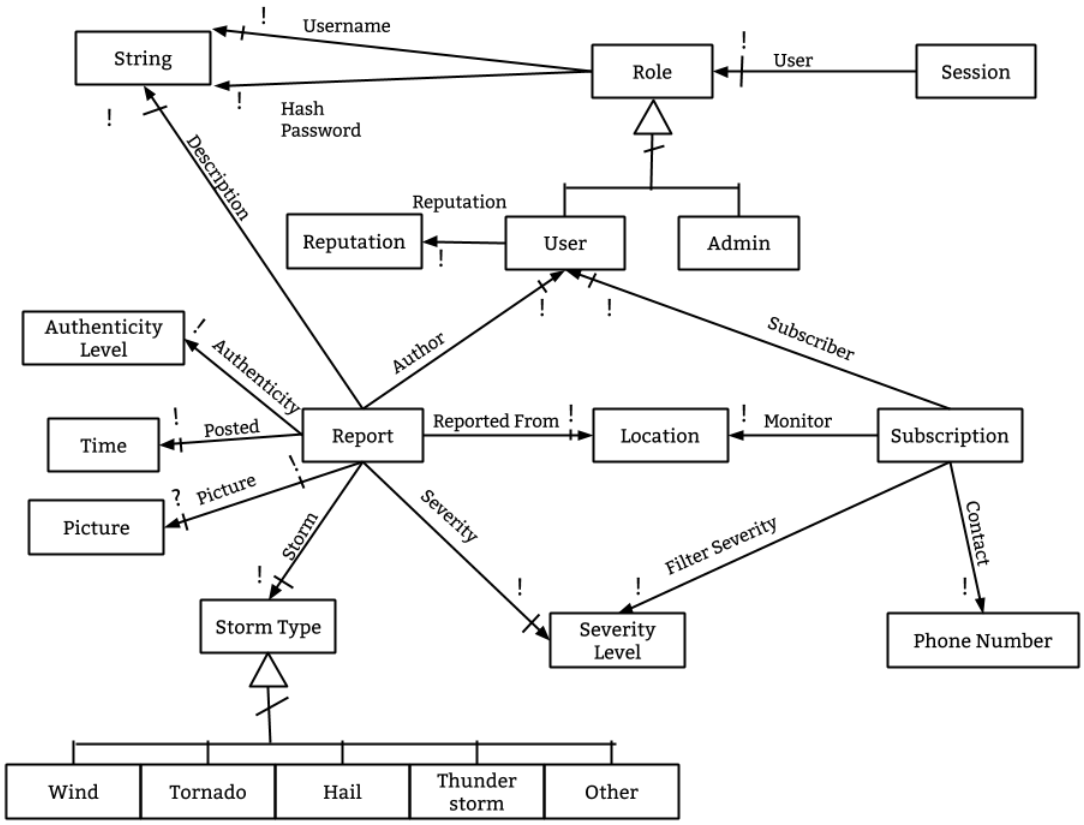
Report:
Brief Definition: User submitted information about a storm.
Motivating Purpose: "Let the user report a storm". This is the entity that captures the information that user reports about a storm.
Description: A report includes a timestamp (when the report was made), a location, an author, a storm type (ex. tornado, thunderstorm, wind, hail), a severity (the higher the severity, the more dangerous the storm), a trustworthiness (i.e. the number of users who upvoted the report minus the number of users who downvoted the report), a text description, and an optional image.

Storm Map:
Brief Definition: A map displaying trustworthy 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. the reports with the highest trustworthiness value) 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 the User and Admin roles are subsets of the Role entity. 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.

Both Admins and Users have immutable usernames and mutable passwords. A User also has a reputation, which indicates how reliable their reports are. The reputation of a user can change as the user makes reports, so it is mutable.

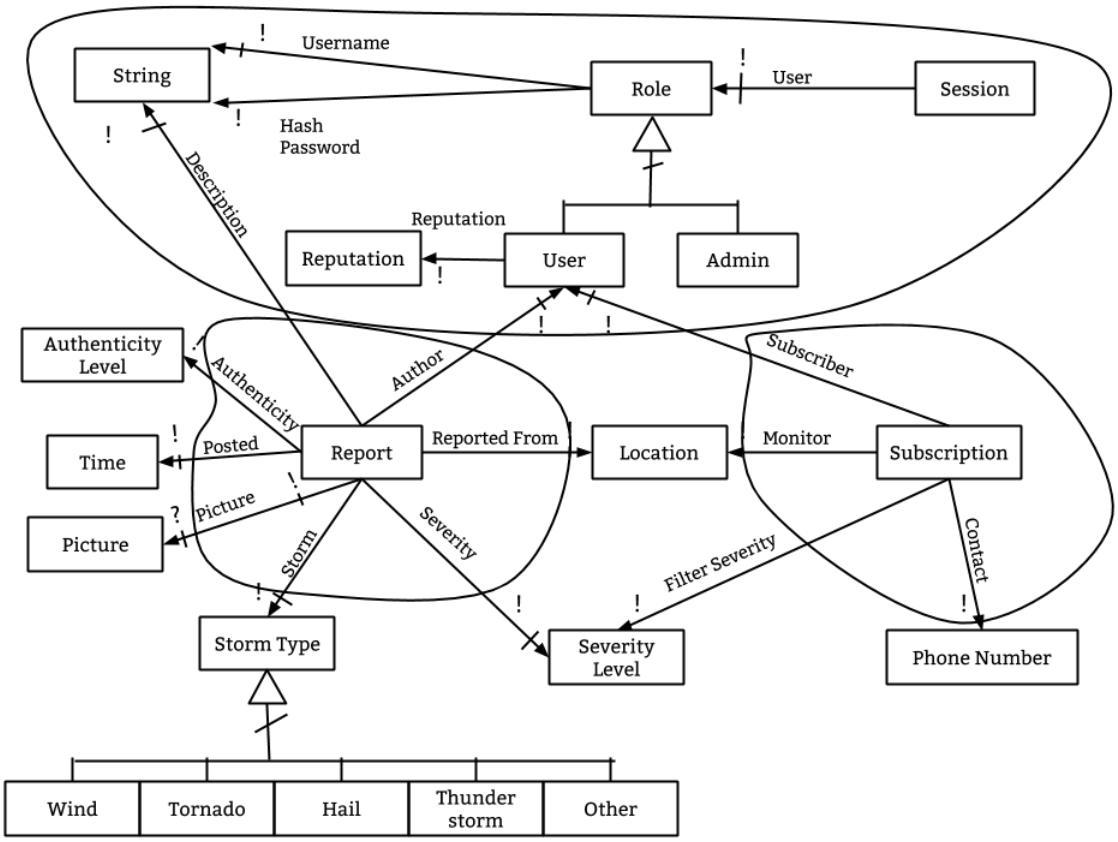
A Report has an authenticity that can change as users upvote or downvote it. It has an author, post time, severity, reported from location, and storm type, each of which cannot change. A report can also have at most one picture, which cannot change.

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.

You will notice that the data model makes no reference to maps, weather alerts, or radar imagery. This is because that data is retrieved from other services (ex. Google Maps and Wunderground). Therefore, there is no need to store that information in our data model.

Data Design



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

Role

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

Report

```
     poster: ObjectId pointing to a Role
     posted_at: Date
     posted_from: GeoJSON point
     description: String
     authenticity: Number
     severity: Number
     storm_type: “Tornado”, “Wind”, “Hail”, “Thunderstorm”,
or “Other”
     picture: ObjectId pointing to a BinData
```

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 Twilio.
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.

Challenge: How do we store pictures?
Options: BinData, Dropbox, No pictures, other APIs
Choice: MongoDB includes a BinData type, which stores binary data. If there isn't enough time, we won't support pictures.
Justification: We can store pictures in the database using the BinData type. We can use a library like Formidable to ease this process of uploading images. However, since pictures are not essential for a storm report, we may not support pictures for storm reports.

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 Role (ex. Get all reports where the storm type is a tornado, get all the reports near a given location), we segregate the Roles from the Reports.

Similarly, since we often query the Subscriptions independent of the Roles (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 Roles. If we want to retrieve the Subscriptions or Reports for a given Role, 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.