Use-cases and more.

The main use cases for the ***user/customer*** would likely include:

- Requesting help: The user/customer would need to be able to initiate a request for help, such as in the case of a break-in or other emergency, by pressing a button on the app or through the web.
- Viewing the location of the respondent: The user/customer would need to be able to view the location of the respondent/driver in real-time as they travel to their location, so they can see when help is on the way.
- Communicating with the respondent/driver and admin: The user/customer would need to be able to communicate with the respondent/driver, such as answering questions or addressing concerns, and providing status updates on the situation. They should also be able to communicate with the admin, such as if they need to cancel the request or if they need help with anything.
- Updating the status of the request: The user/customer should be able to update the status of the request, such as if the

situation is resolved, or if the request is no longer needed.

- Viewing and managing their own account: The user/customer would need to be able to view and manage their own account, such as viewing their personal information and managing their contact information.
- Provide their location: The user/customer should be able to provide their location to the admin/respondent.

The main use cases for the ***respondent/driver*** would likely include:

- Viewing and accepting dispatches: The respondent/driver would need to be able to view incoming dispatches, view the location of the user, and accept the dispatch. Once accepted, the driver would have to navigate to the user location.
- Tracking their location: The respondent/driver would need to be able to track their location, so that the user and the admin can see their progress as they travel to the user's location.
- Communicating with the user and admin: The respondent/driver would need to be able to communicate with the user, such as answering questions or addressing concerns, and providing status updates on their progress. They should also be able to communicate with the admin, such as if the user has cancelled the request or if they need help with anything.
- Updating the status of the request: The respondent/driver should be able to update the status of the request, such as when they arrive at the user's location or when the request is completed.
- Viewing and managing their own account: The respondent/driver would need to be able to view and manage their own account, such as viewing their personal information, and manage their availability.

For the *admin*, the main use cases would likely include:

- Managing and dispatching responders: The admin would need to be able to see incoming requests from users, view the location of the user, and dispatch the appropriate responder to the location. They would also need to be able to cancel requests and reassign responders as needed.
- Tracking responder location: The admin would need to be able to see the real-time location of the responders as they travel to the user's location. This would allow them to monitor the progress of the response and make sure that the responder arrives at the destination in a timely manner.
- Monitoring and managing user accounts: The admin would need to be able to manage user accounts, including adding, editing, and deleting users. They would also need to be able to view user information, such as contact details, and monitor user activity.
- Viewing and managing analytics: The admin would need to be able to view analytics and statistics on the app usage, such as the number of requests and dispatches, the number of active users, etc. This would allow them to monitor the performance of the app and make data-driven decisions.
- Managing and monitoring the system: The admin would need to be able to monitor the system, make sure that everything is running smoothly, troubleshoot and debug issues as they arise and maintain the system.
- Communicating with users: The admin would need to be able to communicate with users, such as answering questions or addressing concerns, and providing status updates on requests and dispatches.

Here's a general outline of the steps you'll need to take to build this app:

- **Create a Backend:** The first step would be to create a backend that can handle the communication between the user and the company/admin. You can use Node.js and TypeScript to create a

backend that exposes an API for the mobile app to consume. This API should allow the user to send an emergency request, view the location of the respondent, and communicate with the company/admin. You can use a database such as MongoDB to store data like user location, and use web sockets to send real-time updates to the user and the admin.

- **Build the Mobile App:** Next, you'll need to build the mobile app that users will use to send emergency requests and view the location of the respondent. You can use a mobile development framework like React Native to build the app, which allows you to create apps for both iOS and Android. You'll need to create an interface for the user to send a request, view the location of the respondent, and communicate with the company/admin.
- **Integrate the Backend and the Mobile App:** Once you have the backend and mobile app built, you'll need to integrate them. The mobile app should make API calls to the backend to send and receive data. You'll also need to use a library like **socket.io** to handle real-time updates between the user, the respondent, and the company/admin.
- **Test and Deploy:** Finally, you'll need to test the app thoroughly and deploy it to the app stores so that users can download it.

Additionally, you can also consider using a real-time location tracking service like Open Layers, leaflet, or Google maps to display the location of the user and the respondent on a map.

Here's a general outline of the steps you'll need to take to build the backend for your emergency response app:

- **Create a Database:** The first step would be to create a database to store information about users, emergency requests, and respondents. You can use MongoDB or any other NoSQL database. You'll need to create collections to store

user information, emergency requests, and respondent information.

- **Define Data Models:** Next, you'll need to define the data models for the different types of data that you'll be storing in the database. For example, you'll need to define a data model for a user, which might include fields like name, email, and location. Similarly, you'll need to define a data model for an emergency request, which might include fields like the user's location, the type of emergency, and the status of the request.
- **Create API Endpoints:** Once the data models are defined, you'll need to create API endpoints that the mobile app can use to communicate with the backend. These endpoints should allow the user to send an emergency request, view the location of the respondent, and communicate with the company/admin. Some of the endpoints you might need include:

```
POST /emergency-request: To create an emergency request
```

```
GET /emergency-request: To get a list of all emergency requests
```

```
GET /emergency-request/:id: To get the details of a particular emergency request.
```

```
PUT /emergency-request/:id: To update the status of a particular emergency request.
```

```
GET /respondent: To get the location of the respondent.
```

```
GET /users: To get a list of all users.
```

```
GET /users/:id: To get the details of a particular user.
```

- **Authentication & Authorization:** You'll need to create an authentication system that allows users to sign up and log in to the app. You'll also need to implement an authorization system that controls which users can access which parts of the app. You can use JWT token-based authentication to secure the API endpoints.
- **Real-time Communication:** To enable real-time communication between the users, the respondent, and the company/admin, you can use web sockets. You can use a library like **socket.io** to handle real-time updates between the user, the respondent, and the company/admin.
- **Test and Deploy:** Finally, you'll need to test the API endpoints thoroughly and deploy it to a server.

You can also consider using a pre-built platform or service that can help you with geolocation tracking, such as Open Layers, leaflet, or Google maps to display the location of the user and the respondent on a map.

The number of collections you have in your MongoDB database and how the models look like will depend on the specific requirements of your application. But here are some of the collections you might consider:

- Users collection: This collection would contain information about the users of the app, such as their name, contact information, location, and any other relevant information.
- Requests collection: This collection would contain information about the requests initiated by the users, such as the status of the request, the location of the user, and the date and time the request was made.
- Respondents collection: This collection would contain information about the respondents/drivers, such as their

name, contact information, location, and any other relevant information.

- Admin collection: This collection would contain information about the administrators, such as their name, contact information, and any other relevant information.

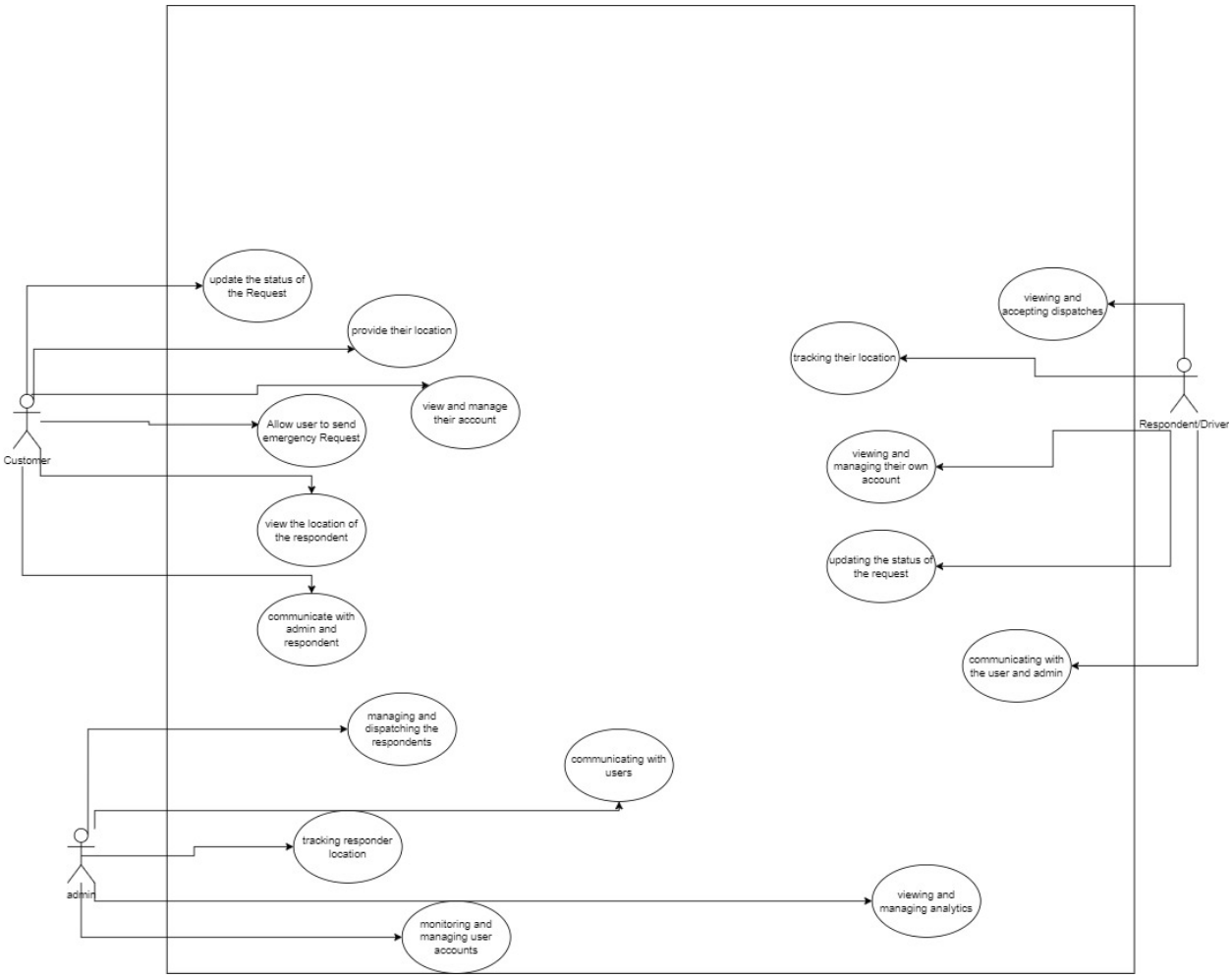
These collections can be further divided into sub-collections based on the needs of your application, but this would be the basic structure. The models in MongoDB are called Schemas and they define the structure of the data in a collection, including the fields and data types of each document in the collection.

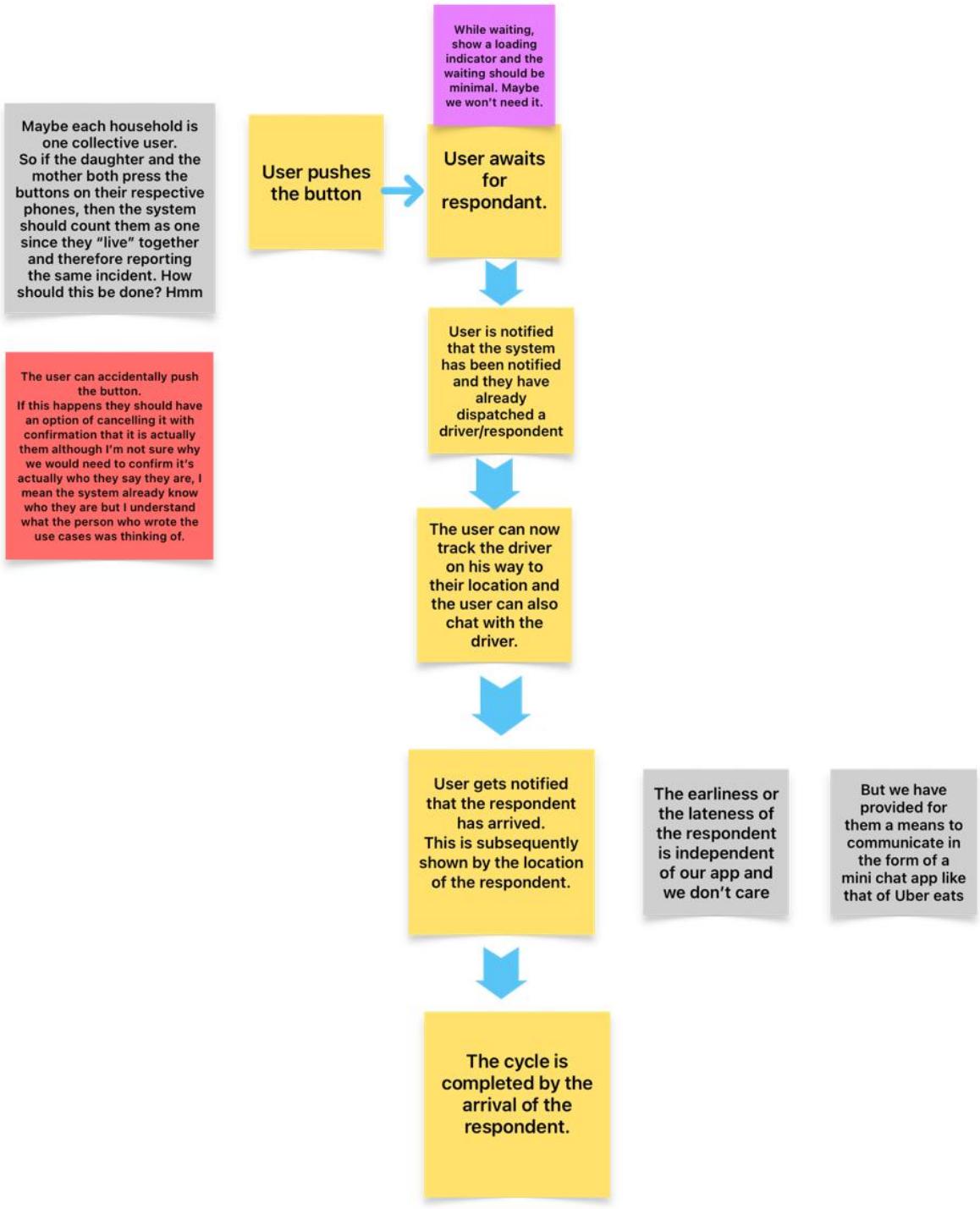
For example, the Users schema could look like this:

```
const mongoose = require('mongoose');
const UserSchema = new mongoose.Schema({
name: {
type: String,
required: true
},
email: {
type: String,
required: true
},
password: {
type: String,
required: true
},
location: {
type: String,
required: true
},
date: {
type: Date,
default: Date.now
}
});
```

```
module.exports = User = mongoose.model('user', UserSchema);
```

This is just an example, and the exact schema structure will depend on the specific requirements of your application.


update the status of the Request
provide their location
view and manage their account
Allow user to send emergency Request
Customer
view the location of the respondent
communicate with admin and respondent
viewing and accepting dispatches
tracking their location
Respondent/Driver
viewing and managing their own account
updating the status of the request
communicating with the user and admin
managing and dispatching the respondents
communicating with users
tracking responder location
admin
viewing and managing analytics
monitoring and managing user accounts


While waiting, show a loading indicator and the waiting should be minimal. Maybe we won't need it.
Maybe each household is one collective user. So if the daughter and the mother both press the buttons on their respective phones, then the system should count them as one since they "live" together and therefore reporting the same incident. How should this be done? Hmm
User pushes the button
User awaits for respondant.
User is notified that the system has been notified and they have already dispatched a driver/respondent
The user can accidentally push the button. If this happens they should have an option of cancelling it with confirmation that it is actually them although I'm not sure why we would need to confirm it's actually who they say they are, I mean the system already know who they are but I understand what the person who wrote the use cases was thinking of.
The user can now track the driver on his way to their location and the user can also chat with the driver.
User gets notified that the respondent has arrived. This is subsequently shown by the location of the respondent.
The earliness or the lateness of the respondent is independent of our app and we don't care
But we have provided for them a means to communicate in the form of a mini chat app like that of Uber eats
The cycle is completed by the arrival of the respondent.