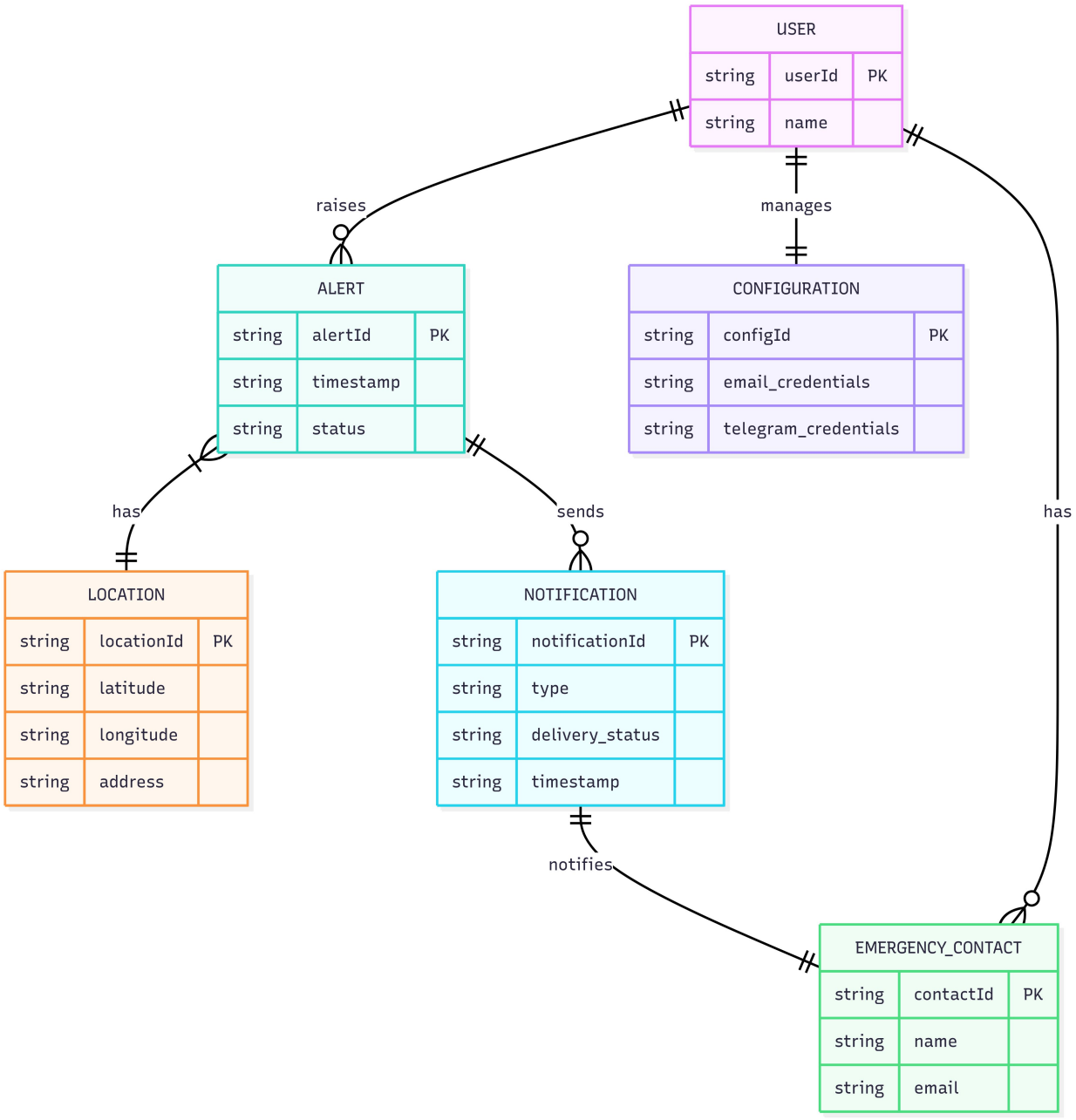
Women Safety Project Details

Entity Relationship Diagram



What an ER Diagram Is

An ER diagram is like a blueprint for a database. It shows you the different pieces of data (called **entities**) in a system and how they are all connected to each other (their **relationships**). It's a great way to understand how information is organized, even without seeing the code.

Women Safety Project Details

Breakdown of Your Diagram

The diagram shows six main entities, or data tables, that make up the core of your women's safety alert system.

1. **USER**
    a. **Purpose:** This entity represents the individual who uses the application.
    b. **Details:** It has a `userId` as the primary key (`PK`), which uniquely identifies each user. It also stores the user's `name`.
    c. **Relationships:**
        i. **"raises"** alerts: A single `USER` can raise many (`o{`) `ALERT`s.
        ii. **"manages"** a configuration: Each `USER` has exactly one (`||`) `CONFIGURATION`.
        iii. **"has"** emergency contacts: Each `USER` can have multiple (`o{`) `EMERGENCY_CONTACT`s.
2. **ALERT**
    a. **Purpose:** This entity represents a single instance of an emergency alert being triggered.
    b. **Details:** It is uniquely identified by an `alertId` (`PK`). It also records the `timestamp` (when the alert was sent) and its current `status` (e.g., "sent," "failed").
    c. **Relationships:**
        i. **"has"** a location: Each `ALERT` has exactly one (`||`) `LOCATION` tied to it.
        ii. **"sends"** notifications: A single `ALERT` can send many (`o{`) `NOTIFICATION`s.
        iii. **"raises"** (from USER): It's raised by one specific `USER`.
3. **LOCATION**
    a. **Purpose:** This entity stores the geographical data for each alert.
    b. **Details:** It has a `locationId` (`PK`), and fields for `latitude`, `longitude`, and `address`. The diagram shows all three are stored as strings.
    c. **Relationships:** It is a part of an `ALERT`.
4. **CONFIGURATION**
    a. **Purpose:** This entity holds all the personal settings and credentials for the user's alert system.
    b. **Details:** It has a `configId` (`PK`), and stores the `email_credentials` and `telegram_credentials` needed to send the alerts.
    c. **Relationships:** It is managed by one specific `USER`.
5. **EMERGENCY_CONTACT**
    a. **Purpose:** This entity represents a person who should receive an alert.
    b. **Details:** It has a `contactId` (`PK`), and stores the contact's `name` and `email`.

Women Safety Project Details

   c. **Relationships:** It is an emergency contact for one specific `USER`. It is also the recipient of a `NOTIFICATION`.

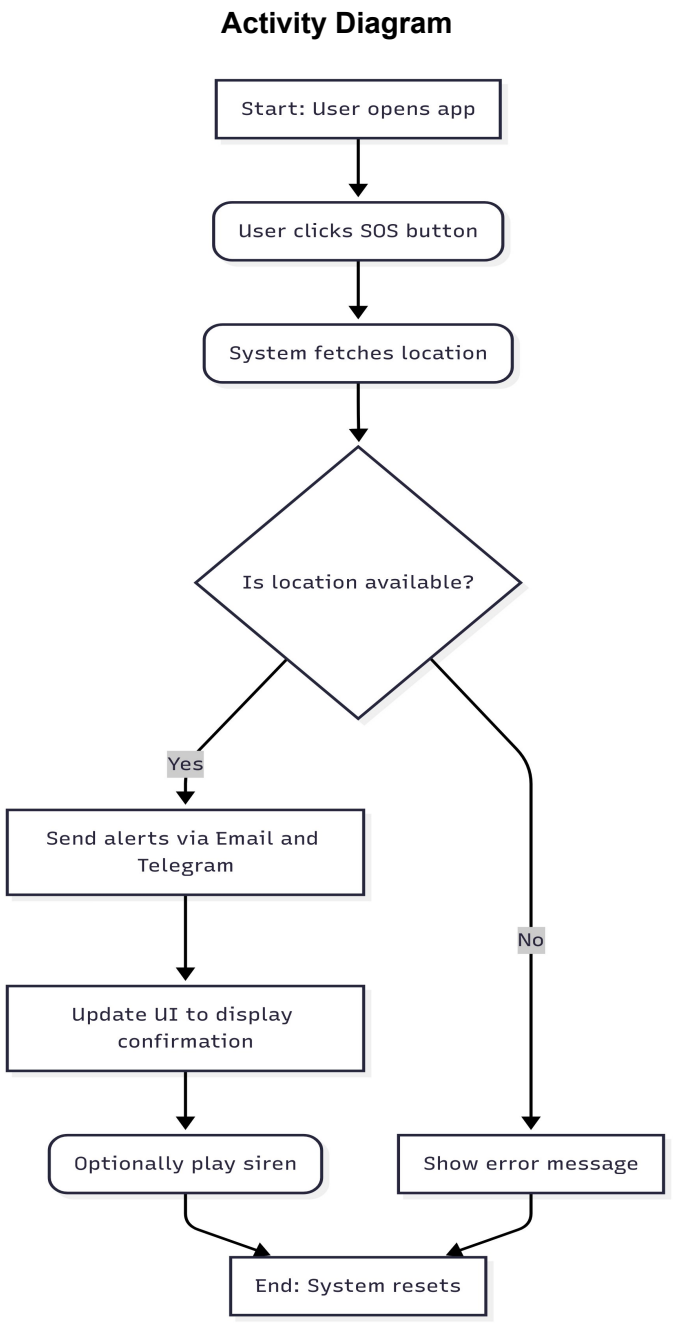
6. **NOTIFICATION**
   a. **Purpose:** This is a crucial entity for tracking how each alert is sent. It connects an `ALERT` to an `EMERGENCY_CONTACT`.
   b. **Details:** It has a `notificationId` (`PK`), records the `type` of notification (e.g., "email", "Telegram"), the `delivery_status` (e.g., "sent," "delivered"), and a `timestamp`.
   c. **Relationships:** It is "sent" by an `ALERT` and "notifies" one specific `EMERGENCY_CONTACT`.

Summary of the System's Data Flow

The diagram tells a clear story about your project's data:

- The system starts with a `USER`, who has a unique `CONFIGURATION` and a list of `EMERGENCY_CONTACT`s.
- When the user triggers an alert, it creates a new `ALERT` record.
- This `ALERT` is immediately tied to a single `LOCATION`.
- The `ALERT` then triggers multiple `NOTIFICATION`s, one for each `EMERGENCY_CONTACT`.
- The `NOTIFICATION` entity acts as a record of each message, so you can track if it was sent successfully to the right person.

Women Safety Project Details



What an Activity Diagram Is

An activity diagram is essentially a **flowchart** for a software process. It shows the step-by-step actions that the system takes from a beginning point to an end point. It is useful for visualizing conditional logic and the different paths the system can take.

In the diagram, the rectangles represent an **action** (something that the system does), and the diamond shape represents a **decision** (a point where the flow can split based on a condition).

Women Safety Project Details

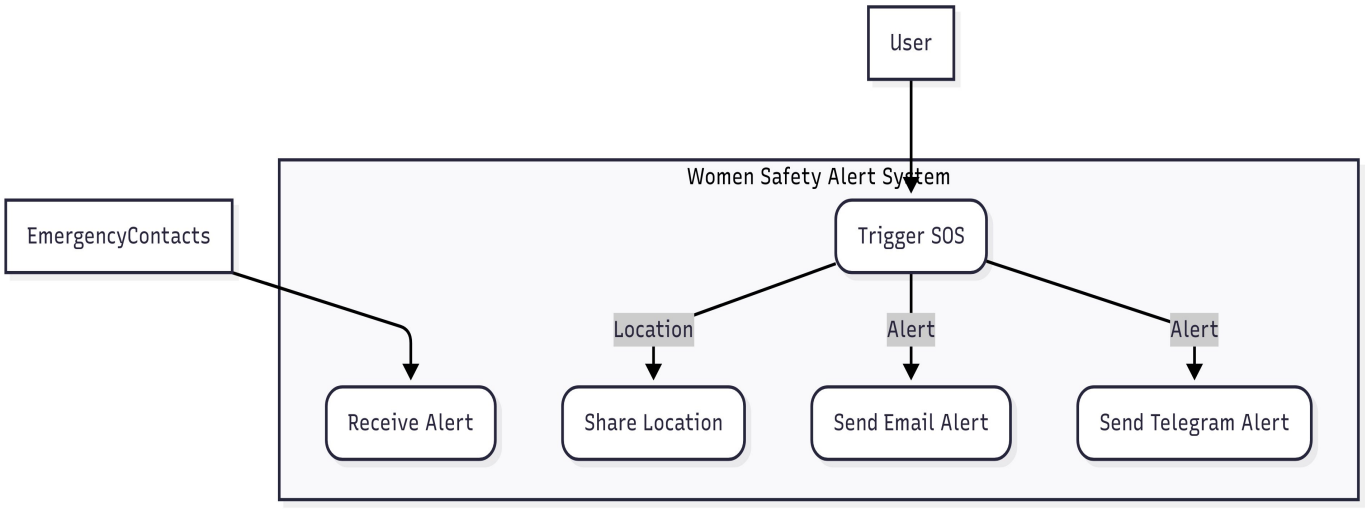
Detailed Breakdown of the Diagram

The diagram walks through the entire alert process, from the user's initial interaction to the final state where the system is ready again.

1. **Start: User opens app**
    a. This is the entry point of the process. It signifies that the system is in a ready state, waiting for user input.
2. **User clicks SOS button**
    a. This is the first **action** and the trigger for the entire alert process. The system moves from a waiting state to an active state.
3. **System fetches location**
    a. After the button is clicked, the system's first task is to get the user's GPS coordinates. This is a critical step because location data is the most important part of the alert.
4. **Is location available?**
    a. This is the **decision point**. The system checks if it was successful in getting the location. This is a vital piece of error handling.
    b. **"Yes" path:** If the location is successfully found, the process moves down the left side of the diagram to send the alerts.
    c. **"No" path:** If the location is not available (for example, if the user denied permission or is in an area with no signal), the process moves down the right side.
5. **Send alerts via Email and Telegram** (from the "Yes" path)
    a. This action represents the successful dispatch of the alert message to the configured contacts. It is a key parallel action, as both emails and Telegram messages are sent.
6. **Update UI to display confirmation** (from the "Yes" path)
    a. After sending the alerts, the system provides feedback to the user, letting them know that the message was sent successfully.
7. **Optionally play siren** (from the "Yes" path)
    a. This is a supplementary action that can run in parallel with the other steps. It's a non-critical feature that provides an audible alert.
8. **Show error message** (from the "No" path)
    a. This action provides feedback to the user when the system fails to get the location. This prevents the user from thinking the alert was sent when it wasn't.
9. **End: System resets**
    a. This is the final state. The process ends here, regardless of whether the alert was successful or not. It signifies that the system has completed its task and is ready for a new one.

Women Safety Project Details

Use Case Diagram



What a Use Case Diagram Is

A Use Case diagram is a high-level tool that focuses on the **who** and the **what** of a system. It shows the system's main functions (the ovals, called "use cases") and the people or external systems that interact with them (the squares, called "actors"). The arrows show how these actors are involved in the system's functions. It provides a simple, clear overview of the system's purpose.

Detailed Breakdown of the Diagram

The diagram for my project is very clear and shows the entire process from the user's perspective.

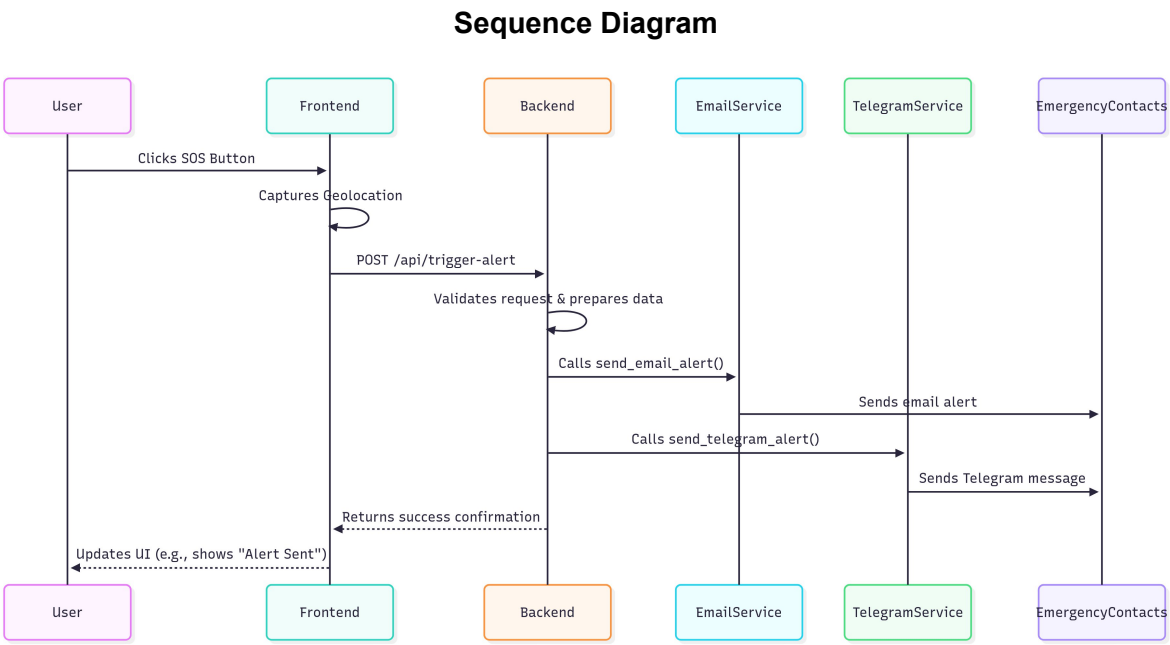
1. **Actors**
    a. **User**: This is the main actor. The diagram shows that the User is the only one who can initiate the process. The arrow from the `User` to `Trigger SOS` indicates that the User is the one who performs this action.
    b. **Emergency Contacts**: This actor represents the people who are on the receiving end of the system's actions. The arrow pointing from `Receive Alert` to `Emergency Contacts` shows that they are passive recipients of the alert.
2. **System Boundary**
    a. The large rectangle with the label "**Women Safety Alert System**" represents the boundary of the project. Everything inside this box is part of the system. This makes it clear what the project is responsible for.
3. **Use Cases (The System's Functions)**
    a. **Trigger SOS**: This is the most important use case, the primary function of the system. It's the action the `User` takes in an emergency. The arrows coming from this use case show that it automatically triggers other actions.

Women Safety Project Details

b. **Share Location**: This is a sub-function. The label "Location" on the arrow indicates that the `Trigger SOS` use case provides location data to the `Share Location` function. This shows that sharing the location is an automatic part of triggering the alert.
c. **Send Email Alert** and **Send Telegram Alert**: These are also sub-functions of `Trigger SOS`. The label "Alert" on the arrows indicates that the primary `Trigger SOS` use case provides the alert message to both of these functions. The diagram shows that both email and Telegram alerts are sent as part of the same action, which highlights the multi-channel nature of your system.
d. **Receive Alert**: This use case represents the final action from the perspective of the `Emergency Contacts`. The arrow from `Send Email Alert` and `Send Telegram Alert` shows that both of these functions lead to the `Receive Alert` use case, completing the entire process.

.

Women Safety Project Details



What a Sequence Diagram Is

A Sequence Diagram is a type of flowchart that focuses on the **order of messages** between different parts of a system over time. It's read from top to bottom, with each vertical line representing a different participant in the process (e.g., the User, Frontend, Backend). The horizontal arrows show messages being passed between these participants in the exact order they occur.

Detailed Breakdown of the Diagram

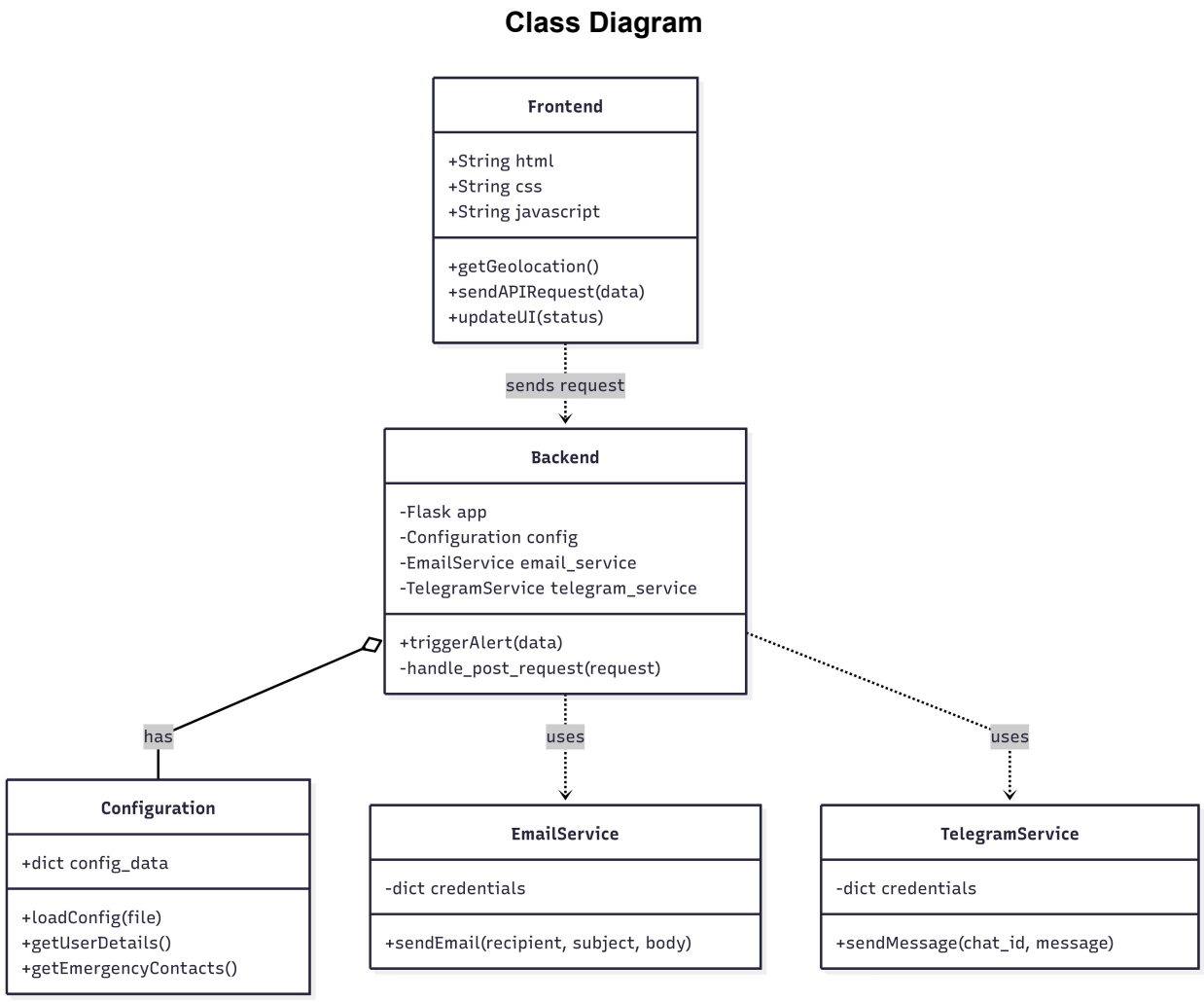
This diagram clearly shows the entire alert process in a step-by-step timeline.

1. **User Initiates the Action:**
    a. The process begins with the **User** clicking the "SOS Button" on the **Frontend**. The horizontal arrow labeled `Clicks SOS Button` shows this is the first message.
2. **Frontend Takes the Lead:**
    a. After the click, the **Frontend**'s first action is to get the user's location. The arrow labeled `Captures geolocation` shows a message being sent from the Frontend to itself, which indicates an internal action.
3. **Client-Server Communication:**
    a. Once the location is ready, the Frontend sends a message to the **Backend**. This is a `POST` request to the `/api/trigger-alert` endpoint, which is the key signal to start the backend's work.
4. **Backend Processes the Request:**
    a. The **Backend** receives the request and, as shown by the self-referencing arrow, it internally validates the data and prepares the alert messages.

Women Safety Project Details

5. **Multi-Channel Alert Dispatch:**
    a. This is where the core logic of your system is shown. The Backend sends two separate messages:
        i. It calls the **EmailService** to trigger the email alert.
        ii. It calls the **TelegramService** to trigger the Telegram message.
    b. The arrows from the `EmailService` and `TelegramService` to `EmergencyContacts` show the final messages being delivered.
6. **System Feedback:**
    a. After successfully calling both services, the Backend sends a message back to the Frontend. The arrow labeled `Returns success confirmation` shows that the Backend confirms the alerts were sent.
    b. The Frontend then updates its UI to show a message like "Alert Sent," providing crucial feedback to the User.

Women Safety Project Details



What a Class Diagram Is

A Class Diagram is a fundamental tool in software design. It's a blueprint that shows the main components of a system (the **classes**), their properties (the **attributes**), their functions (the **methods**), and how they are all connected. It gives you a clear, static view of how your code is organized, and it proves that your project has a well-thought-out structure.

Detailed Breakdown of Your Diagram

This diagram is an excellent representation of a well-organized, modular application. It breaks down the system into five main components, each with a clear role.

1. Frontend Class

a. **Role:** This class represents the user-facing part of your application. It's what the user sees and interacts with.
b. **Attributes:** It includes properties like `html`, `css`, and `javascript` to show that the front end is built using these languages.

Women Safety Project Details

c. **Methods:** It has functions such as `getGeolocation()`, `sendAPIRequest(data)`, and `updateUI(status)`. These methods represent the core actions the front end is responsible for: getting the user's location, sending data to the backend, and updating the user interface.

2. Backend Class

a. **Role:** This is the central hub of your application. It's the server that handles all the business logic and orchestrates the alert process.
b. **Attributes:** It contains instances of other classes like `Configuration`, `EmailService`, and `TelegramService`. This is a key design pattern known as **composition** (shown by the solid diamond), which means the `Backend` "has a" relationship with these other components.
c. **Methods:** Its main function is `triggerAlert(data)`, which kicks off the entire process. It also has a helper method `handle_post_request(request)`.

3. Configuration Class

a. **Role:** This class is responsible for loading and managing all the system's settings and credentials from the `config.json` file.
b. **Attributes:** It contains a `dict` (dictionary or object) called `config_data`, which stores all the settings.
c. **Methods:** It has methods like `loadConfig(file)` to read the settings and `getUserDetails()` and `getEmergencyContacts()` to retrieve specific information. This proves you've separated your settings from your core logic, which is a best practice.

4. EmailService and TelegramService Classes

a. **Role:** These two classes are specialized components that handle a single, specific task: sending alerts via their respective channels. This is an example of the **Single Responsibility Principle**.
b. **Attributes:** They both hold a `dict` for `credentials` to access their respective APIs.
c. **Methods:** The `EmailService` has a method to `sendEmail(recipient, subject, body)`, while the `TelegramService` has one to `sendMessage(chat_id, message)`.
d. **Relationships:** The dotted lines labeled "uses" show that the `Backend` **depends on** or uses the services of these classes to perform its duties. This makes the code modular; you could easily swap out one service for another without changing the backend's main logic.