Ben Henshaw

CS 5001

Prof. Lindsay Jamieson

December 12, 2023

**Final Project Overview**

From the beginning of the project, the goals and overall product idea has remained the same: build a persistent alarm clock that is difficult to ignore that can be edited easily without bloated apps or unnecessary features. Since then, the image of how this could be accomplished has only grown more sophisticated. As such, each component of this project can be broken down into a series of interdependent undertakings, of which I will cover in categories presented as bulleted lists.

**Flask Framework**

This is a micro web framework written in Python, which provides a lightweight set of essentials for building a web application. For this small-sized web application, the flexibility of this framework was perfect for what I was trying to do it. Here are a few features that I utilized for this project:

* Routing: a means of mapping URLs to Python functions, which makes it easy to control the web application through client requests.
  + Each route is defined like `@app.route(‘ext’)` and will execute a subsequent block of code if a URL of pattern *ext* is called from the Flask server.
  + Flask allows for dynamic URLs, which allows the user to pass variables to the main script by accessing URLs
* Templates: using the Jinja2 template, error messages can be communicated to the client without requiring access to a terminal for wholesale error resolution. Information is passed to and from HTML forms using HTTP requests, which generate forms which are then processed in a Python backend.
  + GET requests are used when rendering a form
  + POST requests are used for submitting form data
* Development Server and Debugger: Flask servers run in real time, so updated code is actively reflected in the execution of the web application.
* Support for extensions: Flask\_SQLAlchemy was used to handle database management, which is made to be used as an extension of the Flask client.
  + ORM (Object-Relational Mapping) from Flask\_SQLAlchemy allows for interaction with a database using Python classes and objects instead of SQL queries
  + CRUD (Create, Read, Update, and Delete) operations allow for new entries to be added, the modification of existing entries, and accessing existing entries as part of the function of the main script.

The Flask server offers a form in which events can be created and an arbitrary number of reminders can be instantiated and assigned to the events. The user may upload images to be associated with reminders, which are then displayed on the events page. The events page shows every active event (event\_lock=False) and it can be shown in ascending or descending order. Events can also be ‘deleted’, which raises the event\_lock flag but leaves the entry present in the database itself for future reference.

*Practical Examples*

Basic page route:

@app.route("/") # When accessing the root website, which shows the alarm submission form

def index():

    print("INDEX.APP.PY: '/' Root route triggered. Rendering alarm template.")

    return render\_template("index.html")

Form submission route:

@app.route('/submit', methods=['POST']) # HTTP verb called for sending data to a server when host/submit URL is called

def submit():

    '''

    Uses functions parse\_form\_data and add\_event\_reminders to convert user input into Event and Reminder objects, which are then stored in an SQLite database

    '''

    print(f"\n\nSUBMIT.APP.PY: Event and reminder form submitted.\n Request form: {request.form}\n\n Request files: {request.files}\n\n")

    print("Passing form to parse\_form\_data...")

HTML template with Jinja2:

            {% with messages = get\_flashed\_messages() %}

                <!-- If there are any messages provided -->

                {% if messages %}

                    <ul>

                        <!-- For all of the messages returned -->

                        {% for message in messages %}

                            <!-- Print it as entries in a list -->

                            <li>{{ message }}</li>

                        {% endfor %}

                    </ul>

                {% endif %}

            {% endwith %}

**Database Design and Management**There needed to be a system in place to store and retrieve event and reminder data for relevant functions and for display to the client. The structure of the database was determined by the class definitions in Flask SQL Alchemy. Every instantiated object in the database has a primary key association to differentiate new entries. Events store this key, the event title, the event description, and a lock to determine if the event is active or not. The reminder object stores relevant reminder data, but due to the many-to-one relationship of reminder to events, they also carry a foreign key that points to the associated event object. This makes for simple queries of recent reminders that can simultaneously call the events that they’re associated with. Database sessions are used to maintain long-running conversations between the application and the database, and the nature of these sessions allows multiple scripts to access and modify databases at the same time. In this case, race conditions were considered unlikely enough to ignore, although there are methods for preventing concurrent writes.

Defining Event and Reminder models and their relationship

class Event(db.Model):

    id = db.Column(db.Integer, primary\_key=True)

    title = db.Column(db.String(120), nullable=False)

    reminders = db.relationship('Reminder', backref='event', lazy=True)

class Reminder(db.Model):

    id = db.Column(db.Integer, primary\_key=True)

    event\_id = db.Column(db.Integer, db.ForeignKey('event.id'), nullable=False)

    # Additional fields...

Creating a new Event

new\_event = Event(title='My Event', description='Event description')

db.session.add(new\_event)

db.session.commit()

Querying the database

events = Event.query.all()  # Retrieves all events

Updating an event

event = Event.query.get(event\_id)

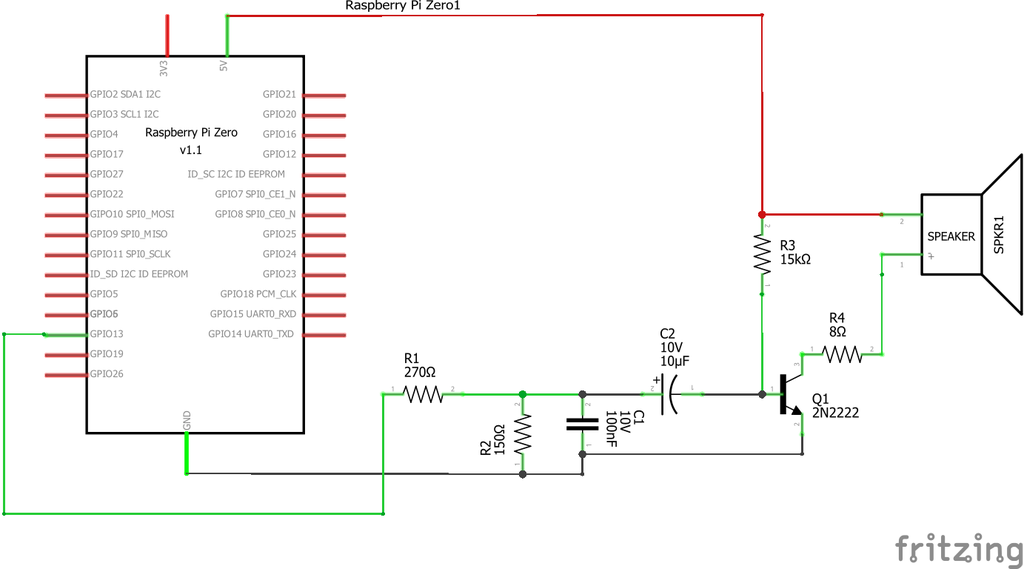
event.title = 'Updated Title'

db.session.commit()

**Raspberry Pi Integration**A Raspberry Pi is an entry-level nexus for hardware components such as buzzers, LEDs, buttons, and LCD displays. This provided a level of physical user interactivity that wouldn’t have been possible otherwise.

* GPIO (General Purpose Input/Output): pins that can provide on or off signals and occasionally act as PWM control (specifically for audio). These are used to control the output devices and take input (in the form of a button).
* I2C communication: offers an interface for a simple LCD display, on which the time and date are displayed. It also displays the web\_unlock key, that the user can then read and enter into a dynamic URL to turn off a reminder should it have that flag.
* Gpiozero: used in conjunction with the custom module `rpi\_models.py`, hardware components were controlled in their own dedicated thread to permit background control of each device.

Components of RPi integration:

1. Software control:
   1. GPIO interactions were scripted using Python
   2. Event-driven programming governed the overall function of these devices, responding to events like button pressed or incoming scheduled reminders.
2. Data exchange:
   1. The RPi retrieves information from the Flask application’s database and responds based on the attributes of the reminders received.
   2. The web unlock feature generates a random key and stores it in a file in the directory shared with the Flask application as well as a flag indicating if the web-unlock feature has been called. This allows dynamic URLs to deactivate the web-unlock that read from the file to verify if it was entered correctly. Entering the values correctly results in the web-unlock feature to be reset.
3. Custom RPi models
   1. rpi\_models controlled the behavior of the output devices so that their behavior can be controlled in their own thread. This allows for beepers and vibration modules to have scripts running in the background of the main script so that the main script can run in the foreground.
4. Hardware considerations:
   1. Audio: one of the aspects of this project involves alarm sound files and text to speech applications. As such, a speaker is absolutely necessary to deliver this. The RPi Zero that was used in this project did not have any built-in audio out, so a hi-pass, one-transistor filter circuit was made to receive signals from a PWM pin and output them as audio at a voltage controlled by the parent power supply[[1]](#footnote-1). This allowed for a mono-speaker setup to be possible for the Zero.
   2. Vibration: an ERM (Eccentric Rotating Mass) motor was utilized to create vibration feedback as an optional flag for the reminders. ERMs do not operate on the output voltage of a single GPIO pin, so some sort of relay controlling a higher voltage was necessary. The ULN2803 chip is an array of Darlington transistors that allow a much smaller signal to switch a larger signal on or off. The 2803 has a freewheeling diode to dispel flyback, or the voltage spike that occurs when an inductive load (such as a motor) is cut off[[2]](#footnote-2). Since only two speeds were necessary (off or all the way on), there was no need for PWM control.
   3. Buzzer: an active piezoelectric buzzer can take a high low-voltage DC signal and generate a high-pitched sound, as it contains its own oscillator to control pitch. This can be powered directly with a GPIO pin
   4. Button: a standard momentary push button will close a signal between a pin and ground, and the voltage drop can be registered as input if that pin has an associated Button object. A limiting resistor is added in-line to a button in the event of accidental output to the pin, as high currents can destroy the functionality of a GPIO pin.
   5. LCD screen: using an I2C module to control contrast and the bussing of I2C information, the number of pins necessary to control the LCD screen used in this project is reduced to 2 as opposed to 8. The power for the screen is provided by an auxiliary power supply to reduce the load on the RPi
   6. Power supply: a simple USB 5V power supply was used as the primary power source for this project. With optional leads to add a battery, this was a necessity for the sake of portability and for the sake of providing enough power for all of the output devices that wouldn’t have been possible with the RPi’s onboard voltage regulator.
5. Real-time scheduling
   1. Scheduling script check for active reminders at regular intervals
   2. Logic present for web-unlock reminders and other optional flags, as well as controlling snooze functionality.

Practical explanation:

Every 30 seconds, the script checks for any active reminders that have occurred since the last check. Then, the script triggers a series of output devices associated with the optional flags associated with each actively triggered reminder, including an alarm sound, a buzzer beeping, or a vibration module vibrating. If a web-unlock flag is raised for one of these events, the web-unlock protocol will begin. A web-unlock key will be generated and displayed on the LCD. As long as the key is displayed, the backlight will be off to make it difficult for the user to read the string without turning on the light. If the user successfully enters the key into the /unlock/ URL, then the web-unlock flag will be lowered. Every 30 seconds, the script checks for new events and if the alarm flag has been lowered, either by pressing the snooze button or by following the web-unlock protocol. For every additional reminder that is tripped during the alarm loop, all of the optional flags from the additional reminders will trigger additional devices, so that way triggered devices accumulate and no new reminder overwrites the activity set by a previous reminder. Audio alarms are randomly picked from a set associated with the urgency of the reminder. If a new reminder has an audio alarm of a higher urgency, it will start playing that higher urgency until the alarm loop is closed. The alarm\_trigger flag is raised whenever an active reminder is discovered and lowered when the snooze button is pressed. The web\_unlock flag is raised when a web-unlock reminder is triggered and is only lowered when the file (unlock.txt) has a value associated with the flag being lowered, allowing the web application to interact with the rpi\_main script indirectly. The alarm activity will continue to loop until both the alarm\_trigger and web\_unlock flags are lowered. Once this occurs, a text-to-speech library will read out all of the events that were associated with the recently triggered reminders. When the device is not in an alarm state, it will display the time and date on the backlit LCD screen, updating once a minute.

**Testing and quality assurance**

No functional project is complete without competent means of testing the code prior to deployment. This is especially important when working with sensitive hardware applications, where misapplied functions stand to destroy the components you’re working with. By setting up code to identify and fix bugs as I go, I was able to deploy the code with little issue.

Functional testing

* Performed as the code was developed to ensure that software and hardware behaved as expected
* A dedicated hardware\_test script was generated which replicated the desired functions of the hardware objects using the gpiozero library. This was used during prototyping stages before everything was soldered into place. It allowed me to determine if there would be any issues with where the devices were being assigned.
* The code is populated with diagnostic print statements to allow me to follow the behavior of the program in real time.

Unit testing

* Not as robust as I would’ve liked because I ended up performing a lot of the tests functionally as the program was developed, so I didn’t have as much of a need to test all of the components of a Flask server whose code I could edit in real time.

Integration testing

* Started by running the web server to input events then asynchronously running the alarm loop to see how and when it responds, then became running each script simultaneously to observe the dynamic response with new events.
* Prior to implementation with the RPi, I mocked gpiozero behavior with corresponding print statements so that it could all be tested on my computer.

End-to-end testing

* After implementation with RPi, I used secure shell (SSH) to access it remotely. I then ran multiple sessions using the `tmux` utility so that I could monitor the console output to both scripts in real time.
* During this time, I was using the application as it was expected to, actively generating reminders and waiting for the alarm loop to respond.
* Once fully tested (still in progress), I will add the two scripts to the boot order so they run on startup.

**~~Bed, Bath and~~ Beyond**

1. A 3D printed case
2. A louder speaker
3. Dedicated web-serving for remote access
4. A page indicating currently triggered reminders
5. A feature to edit events and reminders

This is the most complicated project I’ve worked on so far. It integrated front-end web development with backend hardware interactions in a way that required extensive testing, research, prototyping, and planning. The end product is something that’s totally usable for its intended purpose and the models are flexible enough for future expansion and refinement. The final product I am finishing mere hours before I present it, but as of 4:07PM 12/13, I have a fully functioning product with all desired features implemented. While it was technically difficult, it was easy to stay committed to a project of this scope, as it posed a unique learning opportunity and gave me a grade to justify spending this much time on it. Finishing this actually solves a problem that I have personally and once I have a case and a battery for it, it will have a permanent residence on my bedside table

**Acknowledgements**

Big thanks to Joshua Partridge for helping me set up the Flask server and helping me answer some of the questions I had throughout the process. Shout out to Ethan McCue for helping with random database questions. An additional shout out to Evan Sabin for helping me with the web design.

**Citations (Documentation)**

*GPIOZERO*

<https://gpiozero.readthedocs.io/en/latest/>

*FLASK*

<https://flask.palletsprojects.com/en/3.0.x/>

*SQLALCHEMY*

<https://flask-sqlalchemy.palletsprojects.com/en/3.1.x/>

*RELATIVEDELTA*

<https://dateutil.readthedocs.io/en/stable/relativedelta.html>

*TEXTWRAP*

<https://docs.python.org/3/library/textwrap.html>

*PYTTSX3*

<https://pyttsx3.readthedocs.io/en/latest/>

*PYGAME*

<https://www.pygame.org/docs/>

**Add’l Resources***VAY3T’S I2C LCD DRIVER*

<https://gist.github.com/vay3t/8b0577acfdb27a78101ed16dd78ecba1>

INDEX

**Database Models Definition (models.py)**

**Classes:**

1. **Event**:
   * Represents events in the database.
   * **Attributes**:
     + id: Primary key, unique identifier for each event.
     + title: Title of the event.
     + description: Description of the event.
     + event\_lock: A boolean indicating whether the event is active or locked.
     + reminders: A relationship that links to multiple reminders associated with this event.
   * **Method**:
     + \_\_repr\_\_: Representation method for printing event details.
2. **Reminder**:
   * Represents reminders in the database.
   * **Attributes**:
     + id: Primary key, unique identifier for each reminder.
     + date\_time: Date and time when the reminder is scheduled.
     + buzzer: Boolean indicating if the buzzer is used.
     + vibration: Boolean indicating if vibration is used.
     + web\_unlock: Boolean for web-based unlock feature.
     + reminder\_lock: Boolean indicating whether the reminder is active or locked.
     + alarm: String indicating the urgency level of the alarm.
     + repeater: Frequency of repeating the alarm.
     + event\_id: Foreign key linking to the associated event.
   * **Method**:
     + \_\_repr\_\_: Representation method for printing reminder details.

**Major Variables:**

* db: Instance of SQLAlchemy for database operations

**Flask Server for User Interaction (app.py)**

**Functions:**

1. **index**:
   * Route for the root URL ("/"). Renders the main alarm submission form.
2. **submit**:
   * Route ('/submit') for processing submitted alarm data. Parses form data, validates reminders, adds events and reminders to the database, handles image uploads, and redirects to the index page.
3. **validate\_reminders**:
   * Validates reminder dates and times from user input, ensuring they are future-dated and correctly formatted.
4. **parse\_form\_data**:
   * Extracts and structures data from the alarm submission form for creating Event and Reminder objects.
5. **add\_event\_and\_reminders**:
   * Takes parsed event-reminder data and creates Event and Reminder objects in the database.
6. **key\_check**:
   * Route ('/unlock/<path:key>') for web-based unlock feature. Validates a user-entered URL key.
7. **clear\_web\_unlock**:
   * Handles the process when the web unlock is successful, like resetting flags.
8. **read\_unlock\_val**:
   * Reads unlock keys from a file.
9. **events**:
   * Route ('/events') for displaying all active alarms with options for sorting.
10. **latest\_reminder\_date** and **earliest\_reminder\_date**:
    * Functions for sorting events based on reminder dates.
11. **add\_image\_filepath\_to\_event\_dictionary**:
    * Adds image file paths to event dictionaries for display on events page.
12. **relevant\_events\_filter**:
    * Converts event query objects into dictionaries for display in the HTML template.
13. **delete\_event**:
    * Route ('/delete-event/<int:event\_id>') for deactivating (deleting) an event and its reminders.

**Major Variables:**

* **app**: Flask application instance.
* **db**: Database instance, initialized with Flask app context.
* **image\_folder**, **UPLOAD\_FOLDER**: Configurations for storing uploaded images.

This script serves as the backbone of the web application and handles user interactions through web forms, processes and validates alarm event data, interacts with the database to add or modify alarm events and reminders, and manages the web unlock functionality. The routes enable viewing, adding, and deleting alarm events, as well as uploading related images.

**Raspberry Pi Hardware Control Models (rpi\_models.py)**

**Classes:**

1. **Buzzer**:
   * Controls an active piezoelectric buzzer.
   * **Methods**:
     + \_\_init\_\_: Initializes the buzzer object with GPIO setup.
     + start: Starts the buzzing in a separate thread.
     + stop: Stops the buzzing and cleans up the thread.
     + buzz: Defines the buzzing behavior (on and off sequence).
2. **Vibration**:
   * Manages a vibration motor.
   * **Methods**:
     + \_\_init\_\_: Initializes the vibration object with GPIO setup.
     + start: Activates the vibration in a separate thread.
     + stop: Deactivates the vibration and cleans up the thread.
     + vibrate: Defines the vibration behavior (on and off sequence).
3. **Speaker**:
   * Handles audio playback for alarms.
   * **Methods**:
     + \_\_init\_\_: Initializes the speaker object and pygame mixer.
     + start: Starts playing a sound based on the alarm's urgency in a separate thread.
     + play\_loop: Loops the selected alarm sound.
     + stop: Stops the sound playback and cleans up the thread.
     + select\_random\_alarm: Selects a random alarm sound based on urgency.
     + update\_urgency: stops current audio, updates urgency, and starts playing new audio

This script is crucial for the physical alarm functionality of the project, integrating the Raspberry Pi's GPIO capabilities to control hardware: a buzzer, vibration motor, and speaker. Each class is designed to operate its respective hardware component independently, using threading to allow for concurrent operations without blocking the main program flow.

**Main Raspberry Pi Alarm Clock Program (rpi\_main.py)**

**Functions:**

1. **snooze\_button\_press**:
   * A callback function for the snooze button. It deactivates the current alarm.
2. **initialize\_globals**:
   * Initializes global variables like alarm\_trigger, options\_dict, current\_events\_dict, and current\_urgency.
3. **reminder\_looper**:
   * Adjusts the datetime of reminders based on their repeater value (e.g., "Daily", "Weekly").
4. **fetch\_active\_reminders**:
   * Fetches reminders from the database that are due (current time >= reminder time) and not locked.
5. **update\_reminder**:
   * Modifies a reminder based on its repeater value and updates the reminder in the database.
6. **update\_options\_dict**:
   * Updates a global dictionary with options (buzzer, vibration, etc.) from active reminders.
7. **update\_current\_events\_dict**:
   * Updates a global dictionary with event details from active reminders.
8. **update\_urgency**:
   * Updates the global current urgency level based on active reminders.
9. **process\_event\_reminders**:
   * Processes active reminders and updates global variables and database records accordingly.
10. **reset**:
    * Resets global variables to their default states after an alarm cycle is complete.
11. **write\_to\_file** and **get\_from\_file**:
    * Writing to and reading from a file (used for web unlock feature).
12. **get\_web\_unlock** and **set\_web\_unlock**:
    * Manages the state of the web unlock feature by reading from and writing to a file.
13. **speak**:
    * Uses the pyttsx3 library for text-to-speech functionality.
14. **main**:
    * The main loop of the program. It continuously checks for active reminders, triggers alarms, updates the LCD display, and handles user interactions (snooze, web unlock).

**Major Variables:**

* **app**: Flask application instance for database access.
* **lcd\_screen**: Instance for controlling the LCD display.
* **snooze\_button**: GPIO Button instance for the snooze feature.
* **voice\_engine**: Instance of pyttsx3.init() for text-to-speech.
* **buzzer**, **speaker**, **vibration**: Instances for controlling the buzzer, speaker, and vibration motor.
* **alarm\_trigger**: a boolean flag indicating if there is an active alarm. It controls the main alarm loop.
* **options\_dict**: a dictionary that aggregates the current options for all active alarms. It includes flags for buzzer, vibration, web\_unlock, and a set of alarm urgencies.
* **current\_events\_dict**: a dictionary that keeps track of all current events triggered by the alarm. It uses event IDs as keys and stores tuples of event titles and descriptions.
* **current\_urgency**: a string indicating the highest urgency level among the currently active alarms. It determines the urgency level for the alarm sound.

This script is the central component of this alarm clock system and handles the interaction between the database, user interface, and hardware components. It ensures that reminders trigger the appropriate physical responses (sound, display, vibration) and manages user inputs like snoozing and web unlocking while aggregating hardware output for every additional alarm triggered.

1. https://www.instructables.com/One-Transistor-Audio-for-Pi-Zero-W/ [↑](#footnote-ref-1)
2. https://www.ti.com/lit/ds/symlink/uln2803c.pdf?ts=1702413760176&ref\_url=https%253A%252F%252Fwww.ti.com%252Fproduct%252FULN2803C%253FkeyMatch%253DULN2803C%2526tisearch%253Dsearch-everything%2526usecase%253DGPN [↑](#footnote-ref-2)