“Regulation Break” project.

Team members: Alice England, Emma Gill, Eloise Melville, Emma Moseley, Kathleen O’Sullivan.

1. Introduction: Aims and objectives of the project

1.1 The Problem: What activities to do when taking a regulation break

One of the team members has worked as a special needs teacher in the past. Many of the students are regularly faced with situations in which they might benefit from taking a "regulation break," and they are encouraged to do so.

A "regulation break" is recommended when young people are emotionally uncontrollable and incapable of learning. The "regulatory break" is a period of time spent, usually outside of class, doing anything to help pupils restore emotional control and return to a learning-friendly emotional state (baseline emotional state).

The "zones of regulation" concept is used to communicate emotions. The colours employed in this paradigm to represent the various emotional states are blue, green, yellow, and red. When a person is in the "green zone," they are considered to be at baseline and open to learning. Certain types of activity are regarded to be better suited to specific zones.

Making a decision about what to do during a "regulation break" can be difficult. We determined that there is a need for a technology that can rapidly and arbitrarily generate suitable activities from which children and/or caring adults may choose.

1.2 Satisfies a need/demand for users

We conducted a thorough user experience research strategy to understand our users and their expectations, resulting in the discovery of insights to guide our design decisions. As part of this approach, we did four research experiments: secondary research on regulatory zones, market research, surveys with potential users, and interviews with education specialists.

We found that:

- On the market, there are several mindfulness products with various functionalities.
- Users respond well to evidence-based digital products in the mindfulness category.
- Repetitive exercises are a negative for most users.
- Users respond better to accessible products.
- The Zones of Regulation is a tried and tested method in practice.

We also discovered that our users are (both neurodiverse and neurotypical):

- Young people
- Educators
- Parents and carers.

Link to Atomic Research Canvas
Link to UX Database

1.3 Automates a process

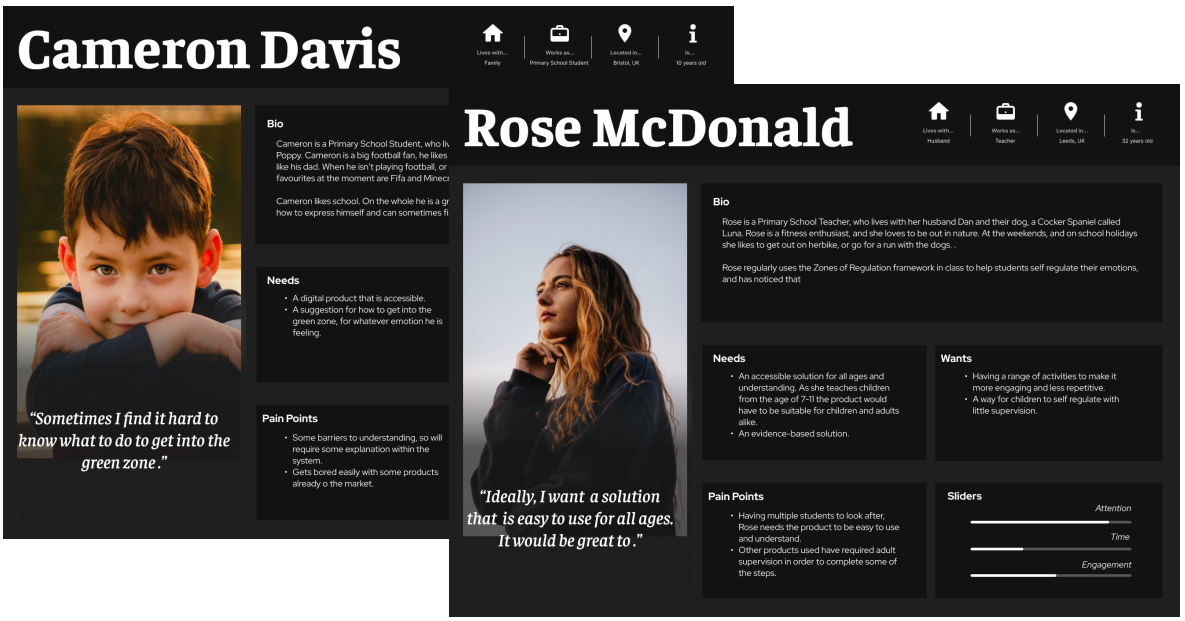
In an ideal world, the app would have a simple interface that could be clicked on to randomly generate appropriate activities.

Image of example analogue activity generator

2. Background

Our solution, Regulation Break, is a random activity generator based on the 'Zones of Regulation' framework. The user is asked to indicate which zone they are in currently, based on how they are feeling, and then they are randomly given an activity suggestion to regulate their mood.

Using the findings from our research, we created two personas (archetypical users who represent the needs of a larger group of users.)



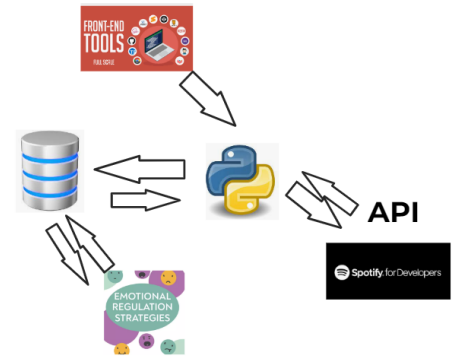
These helped us create our product and maintain a user-focused approach as we developed potential questions to ask. This is what we came up with. Link to [Prototype]

3. Specifications and design

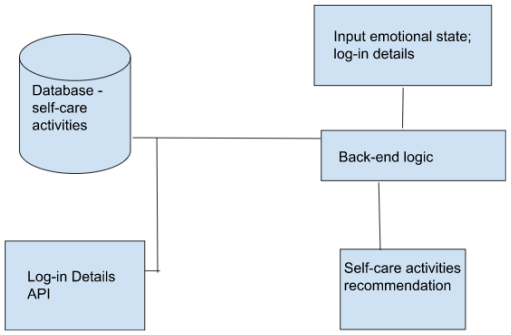
3.1 Key Features and requirements

Initial diagram showing initial ideas of how the app could be created

- An SQL database for the activities.
- Unit testing file.
- Flask, HTML, CSS for the front end.
- An API to connect the SQL database, a login API and a web API to generate music such as spotify.
- Client side file (main file)



After we did some more research and tried out some ideas we decided not to use the spotify web API for this project as the complexity of doing this wasn't possible with our time frame. We settled on a system as shown in the diagram to the right.



Simple Architecture Diagram

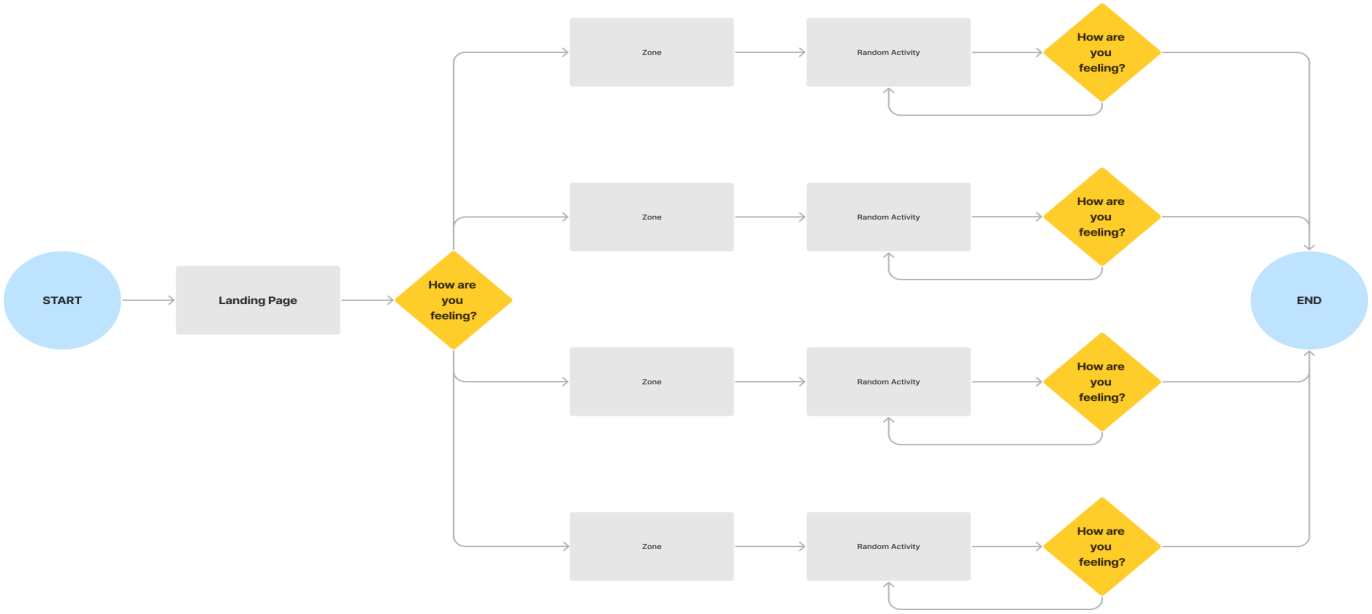
3.2 Design and Architecture

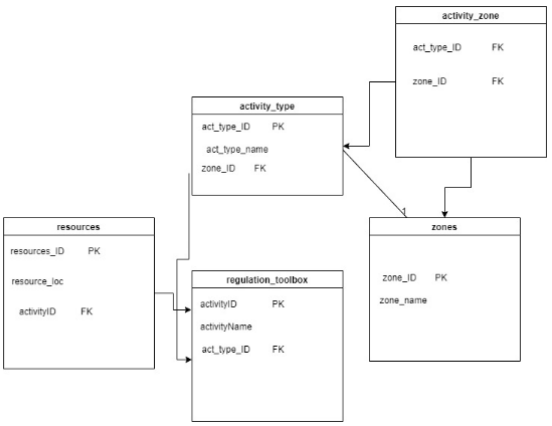
The design and architecture has been sectioned into, front end, database and back end.

3.2.1 Front End:

Users' login information is stored in a SQL database that is linked to the front end. A login can be made by brand-new users and then kept in the database. The user may visit their profile page and the homepage after logging in, where they can select their zone (activating the activity generator API). The user may access the site navigation and a number of different pages via a hamburger menu.

Diagram showing Front end





3.2.2 Database

With the help of the programme draw.io, we first created an entity relationship diagram (ERD) and a database overview schema. This served to identify the primary and secondary keys as well as ensure that the tables did not include any duplicate information.

Entity Relationship Diagram of database

3.2.3 Back end

The back end logic consists of three Python files: the backend API file, a demo main file and a testing file. The API file would create an API with addresses for a randomly chosen activity from the database as well as an address that supplied a running total of the user-inputted zones.

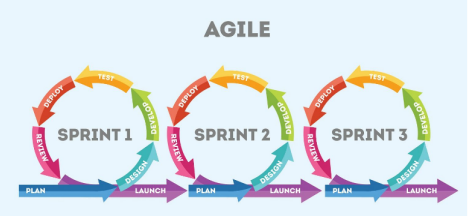
4. Implementation and Execution.

4.1 Development approach and team member roles

We chose an agile strategy and utilised Jira to assign tasks. We picked this strategy because it was deemed an excellent chance to practise this methodology, and also because we were unsure of the final product, making the waterfall method less appropriate. We communicated using a team Slack channel and had meetings via Zoom. To assign and prioritise jobs, we utilised a jira board.

Initially we allocated roles as following:

Alice England - UX design, Frontend
Emma Gill - SQL database, python
Eloise Melville - python testing, python
Emma Moseley - Frontend - login API, flask
Kathleen O'Sullivan - API random generating activities, python



4.2 Tools and Libraries used

4.2.1 Github

We used the version control system Github. We created a repository for our project and each worked on branches that we merged into the main.

4.2.2 Flask

We used the web framework module Flask for developing the web side of the project in Python and connecting the SQL database to the API.

4.3 Implementation processes (achievements, challenges and changes)

4.3.1 Database

Summary of Database implementation

We used MySQL Workbench to make the database. We created tables in the database, gave them primary and secondary keys, and populated them with data. We shared the code via GitHub so that we could all run the database. We had a discussion with one of the tutors about how this would work in 'the real world' with the database being hosted on a server that is running all the time and the problems associated with this, for example power cuts and server down error messages.

Challenges and changes

As we learned, the most difficult issue was the database design. It was difficult to ensure that the tables did not include duplicate data. It was also difficult to figure out where to put the resources so the resource table could be utilised; they were posted to Github and the file paths were used. Different groups in the activity type database, the type of resources stored, and the approach used to select the categories are all possible changes. We chose to employ "zones of regulation," but alternative methods exist, and this might be more personalised.

4.3.2 Python

Summary of Python implementation

The "backend api" file used chain from itertools to list the previous inputs along with the current input when the process was restarted, a counter to count the number of times that each zone was inputted, and random to select an activity from the results returned by the database query in the data layer. The local addresses where the json-formatted results can be viewed were also established in the "backend api" file using decorators.

Using get requests and the addresses defined in the "backend api", the "main" file called information from the json files. All of the established functions were executed by the main file's run function. Overall, the code produced the subsequent procedure. The user's zone is first requested, followed by a suggested activity and a zone count. If they want to do another activity, the preceding stage is repeated. If they choose not to try another activity, they are sent the message "Hope you are feeling better!".

Challenges and changes

The separation between client side and backend development presented another difficulty. It was tempting to keep user input and print functions inside the backend code to facilitate our understanding of what's happening in the program. To help overcome this issue, we carried out integration testing and unit tests, as well as reexamining the project architecture and creating distinct layers for the "frontend," "backend," and "data".

4.3.3 HTML, CSS

Summary of HTML, CSS implementation

The HTML files made use of responsive web design to ensure that the material would display properly on screens of all sizes. Flask requires HTML to be in the 'templates' folder and CSS/images to be in the 'static' folder else they will not be found. Then, these files were connected to the login API. The CSS improves the user experience and the appearance of the app. Additionally, it allowed us to include an off-screen hamburger menu for site navigation.

Challenges and changes

Getting the CSS to appear like the UX design was the toughest obstacle. There were some challenges with trying to align the coloured boxes horizontally and adding the hover function that displayed the image. Although the hamburger menu required some trial and error, it proved successful. A further challenge was incorporating responsive web design and making sure it operated as the UX design intended. Unfortunately, instead of introducing a scrolling

capability (or swiping as it would be on a phone), the coloured boxes on the site just grow smaller when the browser is made smaller, which does not fully match the aesthetic.

4.3.4 Login API

Summary of login API implementation
User accounts' names, email addresses, and passwords are generated and saved in a simple SQL database. Users can fill out a registration form and have their information kept in the database. The API will locate the account information through the login portal, allowing the user to log in to the app. Each account in the database has its own ID number, which is generated by an auto-increment mechanism.

The API connects to the HTML files and the database using Flask. The URLs for the various app pages are organised using the Flask decorator @app.route. Since the login portal and registration are connected to the database, this API employs the methods "GET" and "POST," which are also visible in the HTML code. Following that, a SQL query will be sent to the database to either authenticate the user or add a new user. When a person successfully logs in, a session is formed to provide them access to various parts of the website; when they log out, the session is terminated.

Challenges and changes
The biggest difficulty with the login API was getting it to operate. The initial plan was to connect to the frontend using an existing login API. To begin, we used Okta, a two-factor authentication API that required configuration on the Okta server. This API had numerous flaws, the first of which was that it required older versions of the Python libraries (which might make it incompatible with the rest of the program). The second difficulty was with the server side setup, since it was difficult to find updated documentation on what was required.

The Google OAuth API was the second API we tested. There was plenty of current documentation on this API, but we couldn't locate any that was created just for Pycharm. We were unable to apply several of these documents' advice since the command prompt restricted some of the things they recommended. Even though we could connect to the API, it didn't seem to establish a login session or take us to the appropriate sites. It was agreed that we should create a straightforward SQL database and API to connect to it due to this and numerous time restrictions.

5. Testing and evaluation

5.1 Testing strategy

- Continuous testing
- Unit tests
- Integration testing

- User testing

Unit testing challenges and changes

We used integration testing to test the running process of all layers and communication between layers, as well as separate unit tests to test the data layer. One of the biggest challenges was debugging when exceptions obscured vital or useful error messages. In future, it would be good to use more test driven development to help clarify the design of the program earlier in the process. For example, in the backend api file there is a method record_add_count function which was using the Counter function from collections, however the empty record list was a global variable which would mix all clients records, so we adjusted this by making a local counter within the method which stores a tally of how many times a user chooses each given colour zone. Given more time, we would make further adjustments to save this counter to the database.

Another example of debugging we did through testing was when running the testRandomActivity test in the testingAPI file. The test kept failing because we had a 'display_results' method within the DataLayerRegulationBreak class which was appending two strings 'Here are your {} randomly generated activities' and '---Thank you for using Regulation Break—' to the list of activities and then printing out all items in the list on the console. As such when we called the test, sometimes it was interpreting one of these strings as a random activity and failing the test. To solve this issue we moved the 'display_results' method into a demo file within the data layer to give the developer the option to print these strings as part of a demo instead.

5.2 Functional user testing

We conducted two moderated user testing sessions with participants from our core demographic using the figma prototype, where we found that:

- The landing page interaction has to be improved, as several participants had trouble navigating this.
- Participants praised the system's logical flow and reported that the CTAs were easy to understand.
- The flow was completed by each participant.
- The file paths in pycharm did not work on some users' computers and needed to be changed.

The product would benefit from more user testing, provided there is time. To address the flaws with the interaction, it would be advantageous to do an A/B testing session.

5.3 System limitations

The system is limited by:

- The data stored in the database.
- GDPR if users log in.
- Front-end communication with the database (given more time we would move this function to the backend).
- Safeguarding and safety if users are young people.

5.3.1 Possible next steps

- Linking the suggested activity to a webpage with more resources on the given activity.
- Personalised system: the system suggests things based on your likes and activities.
- Users are able to add their own activities.
- User suggestions and fault reporting.
- Linking the frontend and login API to the activity generator.
- Creating a blog feature to the app for users to use as a journal.
- Adding a search feature to allow users to find different activities based on keywords.

6. Conclusion

Our initiative aimed to solve the issue of locating appropriate activities for assisting vulnerable individuals and children in discovering self-regulation activities. To identify the target market and their demands, we conducted user experience and market research. We built a database filled with self-care tasks, and then we developed a Python API to retrieve and produce these tasks. The user may then keep track of their zones and check their mental health thanks to the vocabulary we established to store the responses. To allow users to engage with the app, we generated web pages. We want to expand the app's capabilities in the future, for example, by connecting the frontend to the activity generator api.

We have improved our understanding of the course's subject matter and had a chance to test our newly-learnt skills in a real world environment. Additionally, we've learnt how to effectively organise the files and other industry best practices. When we shared our code, we ran across a number of problems, such as files that would run on one computer but not another. We improved our debugging skills as a result of being able to execute the files on all of our computers. Finally, we want to keep working on this project to incorporate the other features we've talked about and link the frontend and backend.