FoodWise

Made by: Lior Barak, Michael Shuster, Igal Kaminski

Table of contents

Description:

The problem:

Inefficiency and lack of intelligent management in traditional refrigerators.

- Conventional refrigerators do not provide real-time monitoring, inventory management, or personalized user assistance.

- Current high cost of smart refrigerators presents a significant barrier to widespread adoption which limits accessibility for a broader demographic.

Our solution:

Our Smart Refrigerator project introduces an innovative and cost-effective IoT scanning device, designed to revolutionize how users manage their kitchen inventory, by seamlessly integrating with their conventional refrigerators.

Placed near or on the refrigerator, the IoT device seamlessly integrates with a user-friendly mobile app named “FoodWise”.

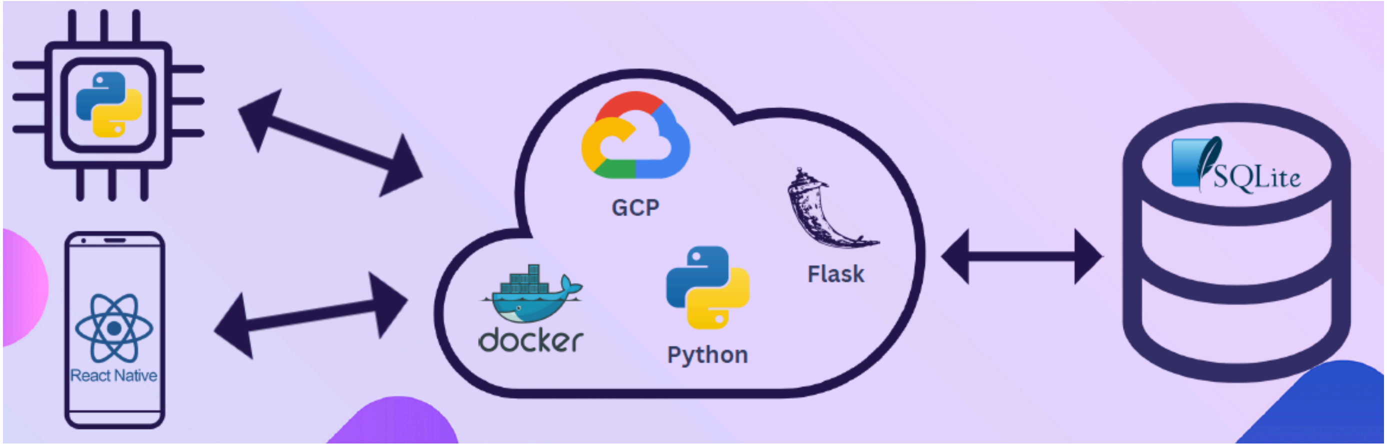
This system aims to not only streamline inventory management, but also enhance the overall kitchen experience by providing features such as: half-automated shopping list creation, track in real-time more than just one refrigerator, get recipe suggestions, view consumption reports and get expiration alerts for products to reduce food waste.

Use cases:

1. **IoT Scanning and Real-time Inventory**
   - A user wants to effortlessly scan items using the IoT device when placing them in/out of the refrigerator and being able to manage and track his food inventory.
   - The app will have to facilitate seamless item scanning through the IoT device and update the real-time inventory list accordingly, allowing the user to view and modify the list through the app.
2. **Half-Automated Shopping List:**
   - A user wants to rely on the system to analyze his scanned inventory and generate a shopping list based on the parameterized list he created before and the products he already have at his refrigerator.
   - The app will have to analyze scanned inventory data and generate a shopping list relying on his current inventory products.
3. **User Refrigerators:**
   - A user wants to track all of his refrigerators, he may have more than one major refrigerator, add new refrigerators to track and change their nicknames.
   - The app will have to track all the user linked refrigerators, allow the user to add new refrigerators by scanning with the refrigerator IoT scanner the user uniq app barcode, and let the user seamlessly update his refrigerators nicknames.
4. **Recipe Suggestions:**
   - A user wants to benefit from the system leveraging scanned inventory data to suggest personalized recipes based on his available ingredients.
   - The app will have to utilize scanned inventory information to offer personalized recipe suggestions and display them to the user in a convenient way.
5. **Consumption Reports:**
   - A user wants to get reports about his refrigerator added and removes habits during a chosen period time by his choice.
   - The app will allow the user to configure start date and end date by his configure and show him in convenient diagrams graphs about his added and removed habits for each refrigerator.
6. **Expiration Alerts to Food Waste Reduction:**
   - A user wants to receive timely notifications for upcoming expiration of scanned items at his inventory.
   - The app will have to send notifications that alert users about expiring items, by alerting dates that the user needs to update on products in his inventory.

The user will notify about the alerts he configures and can search for recipe suggestions to reduce food waste through recipe ideas.

Project Architecture:



Backend: Python, Flask, SQLite Database.

Frontend (Mobile app): React-Native.

Embedded: Raspbian OS, Python.

Hardware: Raspberry Pi, Touchscreen , Scanner.

Cloud: GCP (Google Cloud Provider)

Technical and general server application explanation:

In the FoodWise project, the server main modules includes the following components:

1. **Server.py:** This module contains all the API endpoints for the application.
It handles requests and responses, serving as the main interface between the Frontend and Embedded to the Backend.
It uses all the other server classes from below for its needs.

2. **Database Class:** This class interacts with the SQLite database.
It performs SQL queries to retrieve, insert, update, or delete data as needed, ensuring a structured communication with the database.

3. **Utils Class:** To prevent code duplication in the `Server` class, common methods that are used repeatedly are implemented in this utility class.
This allows for easier maintenance and code reuse.

4. **Model Classes:**

- **Product:** Encapsulates all the product-related data, such as product name, image, quantity, added_date and alert_date.

- **User:** Holds data related to the users of the system, managing user-specific information, such as user name and password.

- **Refrigerator:** Stores data about the refrigerator’s state, including the refrigerator id and the array of products in the refrigerator inventory.

This structure ensures a clean separation of concerns and maintains the modularity of the codebase.

Server dependencies:

Flask

Werkzeug

Flask-Bcrypt

Flask-JWT-Extended

Flask-Mail

Technical and general mobile application explanation:

The application consists of a few screen components:

1. InitialLoadingScreen - the screen is used when the application is booting up to display a loading placeholder.

2. WelcomeScreen - the first screen that the user sees, the screen allows to navigate to the login or registration screens.

3. SignupScreen - the screen allows users to sign up to the application.

4. LoginScreen - the screen allows the user to login to the application, the login authenticates the user with a JWT from the server api.

5. InventoryScreen - the screen is used to display the current inventory of a selected refrigerator. by utilizing the SearchProduct screen, the screen also allows users to add products without a barcode to the inventory (fruits, vegetables, etc). By long pressing on an item in the inventory the user can set up an alert day and change the item's quantity.

6. MyRefrigeratorsScreen - the screen is used to link new refrigerators to the user by displaying a qr code, and select the current desired refrigerator.

7. ShoppingList - the screen is used to access the option to generate and configure the shopping list.

8. RecipesScreen - the screen is used to display recipes that can be made with ingredients that are currently in the inventory of the refrigerator.
The app uses an external API called Spoonacular to access a database of recipes.

9. ConsumptionScreen - the screen is used to display a user’s consumption in a form of graphs.
The app uses the react-native-chart-kit library to display the graphs.

10. NotificationsScreen - the screen is used to display alerts set up by the user in the inventory screen.

11. SettingsScreen - the screen is used to allow the user to change his mail and password, and logout from the application.

12. SearchProduct - this is a utility screen used in some other screens, the screen allows user to search products.

13. EditList - the screen is used to allow users to edit the shopping list/configuration.

The application uses Axios for HTTP communication with the server.

Application Dependencies:

```
"@react-native-async-storage/async-storage": "^1.23.1",
"@react-native-community/datetimepicker": "^8.2.0",
"@react-navigation/drawer": "^6.6.15",
"@react-navigation/native": "^6.1.17",
"@react-navigation/native-stack": "^6.9.26",
"@rneui/base": "^4.0.0-rc.7",
"@rneui/themed": "^4.0.0-rc.8",
"axios": "^1.6.8",
"react": "18.2.0",
"react-native": "0.74.0",
"react-native-chart-kit": "^6.12.0",
"react-native-gesture-handler": "^2.16.2",
"react-native-linear-gradient": "^2.8.3",
"react-native-paper": "^5.12.5",
"react-native-qrcode-svg": "^6.3.1",
"react-native-reanimated": "^3.15.0",
"react-native-safe-area-context": "^4.10.1",
"react-native-screens": "^3.31.1",
"react-native-share": "^10.2.1",
"react-native-svg": "^15.3.0",
"react-native-toast-message": "^2.2.0",
"react-native-vector-icons": "^10.1.0"
```

Technical and general embedded application explanation:

The embedded application has two threads running as part of the program.

The **main thread** is responsible for interacting with the GUI displayed on the touchscreen (Using Tkinter library), which includes three buttons for switching between modes: Add Mode, Remove Mode, and Link Mode.In Add Mode, every scan attempts to add the scanned barcode. In Remove Mode, every scan attempts to remove the scanned barcode. In Link Mode, every scan attempts to link a user to the device. The main thread is also responsible for communication with the server using the `requests` library.

The **secondary thread** is responsible for handling scans (using the evdev library) from the scanner and processing the scanned data.

Both threads use a shared memory space, which is a queue. The secondary thread adds processed scan tasks to this queue, while the main thread retrieves these tasks and sends them to the backend server. The system is designed to be thread-safe, ensuring proper synchronization to handle concurrent access to the queue safely.

When the operating system starts, the software also automatically launches (as explained in the technical notes above). At this point, the main thread imports its identifier. If this is the first time the device has been powered on, it will send a request to the server to obtain a new identifier. Otherwise, it will retrieve the identifier from the device's local memory.

Additionally, if the communication with the server is temporarily lost, the device will manage the situation as follows:

During scanning, it will store all unsent scans in a queue and continue to add new scans to this queue. Once communication is restored, it will immediately send all the queued scans to the server one by one.

If the loss of communication occurs while retrieving the identifier, the device will keep attempting to send requests to obtain a new identifier until the connection is reestablished.

Embedded application Dependencies:

evdev

requests

tkinter

threading

Application Programming Interfaces:

Embedded endpoints:

1. **Get** `‘/request_refrigerator_id’` : Create a new IoT device uniq refrigerator id, only at the first time it starts to work.

   Example of a request: `/request_refrigerator_id`

2. **Post** `‘/link’`: Link between a user to his refrigerator.

   Example of a request: `/link , json={"user_id": 1, "refrigerator_id": 1}`

3. **Post** `'/scan'` : get product barcode, scanning mode and a refrigerator id and add/remove the product with this barcode to the specific refrigerator.

   Example of a request: `/scan , json={"barcode": 7290008757034, "mode": "add", "refrigerator_id": 1}`

Frontend endpoints:

1. **Post** `'/add_product_with_app'`: get product barcode and refrigerator id and add one product to the specific refrigerator. For products without barcodes like vegetables we create internal “fake” barcodes and by their picture and name at the app the user can add them to his inventory.

   Example of a request: `/add_product_with_app , json={"barcode": "7290008757034", "refrigerator_id": 1}`

2. **POST** `'/register'` : Register a new user to the system by email, password, first name and last name.

   Example of a request: `/register , json={"email": "liorbaa@mta.ac.il", "password": "12345678", "first_name": "Lior", "last_name": "Barak"}`

3. **Post** `'/user_login'` : Authenticate users within the system by using their email and password.

   Example of a request: `/user_login , json={"email": "liorbaa@mta.ac.il", "password": "12345678"}`

4. **Post** `'/update_user_email'` : Update the email for a logged in authenticated user by entering a new uniq email.

Example of a request: `/update_user_email , json={"email": "liorbarak99@gmail.com"}`

5. **Post** `'/update_user_password'` : Update the password for a logged in authenticated user by entering a new password.

   Example of a request: `/update_user_password , json={"password": "12345678"}`

6. **Get** `'/linked_refrigerators'` : Get all the linked refrigerators of a logged in authenticated user.

   Example of a request: `/linked_refrigerators`

7. **Get** `'/refrigerator_contents'` : Get the refrigerator content of a logged in authenticated user by a refrigerator id.

   Example of a request: `/refrigerator_contents?refrigerator_id=1,`

8. **Get** `'/number_linked_refrigerators'` : Get the number of the linked refrigerators that the user linked to for logged in authenticated user by a refrigerator id.

   Example of a request: `/number_linked_refrigerators?user_id=1`

9. **Post** `'/update_refrigerator_name'` : Change the nickname of the relevant refrigerator for the given user for logged in authenticated user by a refrigerator id and new name.

   Example of a request: `/update_refrigerator_name?user_id=1 , json={"refrigerator_id": 1, "new_name": "Main Refrigerator"}`

10. **Get** `'/search_products'` : Get up to 10 products that include substring of the product name, if all == 1 for products with barcodes and else for products without barcodes, for an authenticated user.

    Example of a request: `/search_products?product_name=C&all=1`

11. **Post** `'/update_refrigerator_parameters'` : Update the list of products that always needs to be in the refrigerator, by refrigerator id and the list of the parameters products, for an authenticated user.

    Example of a request: `/update_refrigerator_parameters?refrigerator_id=1 , json=[{"barcode": 7290004131074, "amount" : 3}]`

12. **Post** `'/save_shopping_list'` : Save the current shopping list for the relevant refrigerator, by refrigerator id and the list of the shopping products, for an authenticated user.

    Example of a request: `/save_shopping_list?refrigerator_id=1 , json=[{"barcode": 7290004131074, "amount" : 3}, {"barcode": 7290004127329, "amount": 2}]`

13. **Get** `'/generate_initial_shopping_list'` : Get initial shopping list for the relevant refrigerator, that contains all the products that need to be in the refrigerator with their amounts and currently are not in the inventory or their amount less than in the shopping list, for an authenticated user.

    Example of a request: `/generate_initial_shopping_list?refrigerator_id=1`

14. **Get** `'/get_refrigerator_parameters'` : Get the last inserted list of the products that always need to be in the refrigerator, for an authenticated user.

    Example of a request: `/get_refrigerator_parameters?refrigerator_id=1`

15. **Get** `'/fetch_saved_shopping_list'` : Get the last saved shopping list that was created by the user, that is linked to the relevant refrigerator, for an authenticated user.

    Example of a request: `/fetch_saved_shopping_list?refrigerator_id=1`

16. **Get** `'/get_product_alert_date'` : Get a product alert date for a logged in authenticated user, by a refrigerator id and product name.

    Example of a request: `/get_product_alert_date?refrigerator_id=1&product_name=Eggs pack 12L free organic`

17. **Get** `'/get_refrigerator_content_expired'` : Get all the products whose alert date expired for a specific refrigerator of a logged in authenticated user by a refrigerator id.

    Example of a request: `/get_refrigerator_content_expired?refrigerator_id=1`

18. **Post** `'/update_alert_date_and_quantity'` : Update product alert date and quantity of a logged in authenticated user by a refrigerator id, product name, alert date and product quantity.

Example of a request: `/update_alert_date_and_quantity ,json={"refrigerator_id": 1, "product_name": "Milk 1% 1L Tnuva", "alert_date": "2024-09-25", "product_quantity": 3}`

19. **Get**`'/get_statistics'` : Get refrigerator consumption statistics of added and removed products of a logged in authenticated user, by a refrigerator id, product name, alert date and product quantity.

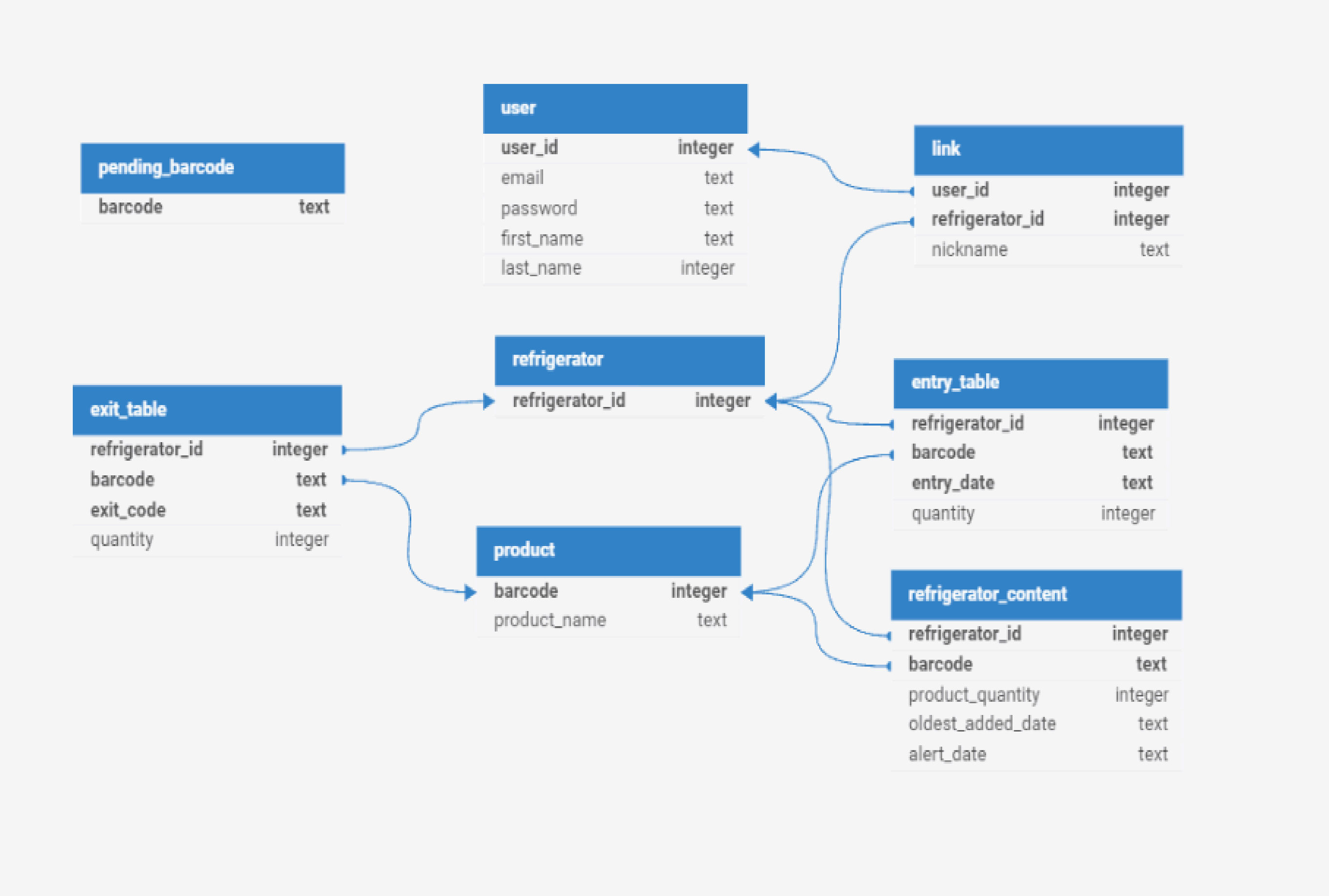
    Example of a request: `/get_statistics?refrigerator_id=1&start_date=2024-07-20&end_date=2024-08-09`

Managers endpoints:

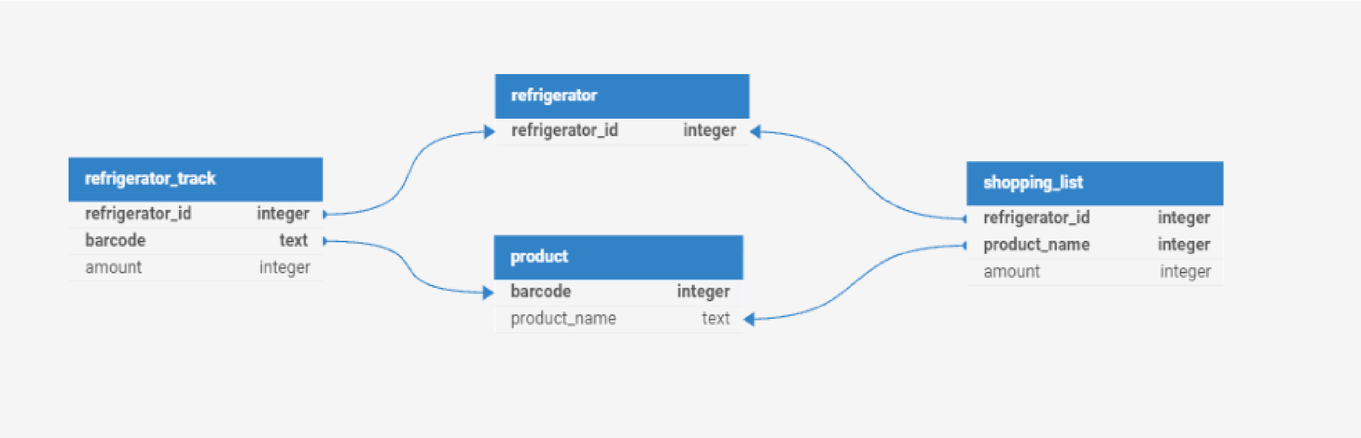
1. **Post** `'/add_new_product_to_DB'` : End point that is intended for managers to add new products to the 'Product' database by product barcode and product name.

   Example of a request: `/add_new_product_to_DB , json={"barcode": "#027", "product_name": "Tahini"}`

Database schemes



Generated with erd.dbdesigner.net



Generated with erd.dbdesigner.net

DataBase tables:

1. product: stores information about each product.

2. user: stores information about each user.

3. refrigerator: stores id of each refrigerator.

4. refrigerator_content: stores the products that are available in each refrigerator.

5. refrigerator_track: stores information on parameters that the user always wants to have in his inventory with their amount.

6. shoppint_list: stores the latest shopping list that was created by one of the linked users per refrigerator.

7. link: stores the links and nicknames between each user to each refrigerator.

8. pending_barcode: stores barcodes of new products that a user scans and we don’t have them yet in our ‘product’ table at the database.

9. entry_table: stores information on every product that enters a refrigerator.

10. exit_table: stores information on every product that is removed from a refrigerator.

How to deploy

Explanation on how to deploy the Flask app to GCP (GOOGLE CLOUD):

First, we need to build a Docker image containing the Flask app and then upload the image to Docker Hub and then upload the Docker to the google cloud.

In the `Server` directory, there is already a `Dockerfile` with all the necessary instructions to create the image.

Now the following steps need to be done.

1) Ensure that the `docker` variable at the beginning of the `server.py` file (line 25 at the time of writing this document) is set to `True`.

2) Ensure that all libraries not included in Python's standard library are listed in the `requirements.txt` file, located in the `Server` directory, in the following format: `<library_name> == <version_number>`.

3) Make sure Docker Engine is installed on your computer and is connected to your Dockerhub account.

4) Open cmd and run the following commands:

- docker build -t my-flask-app .
- docker tag my-flask-app username/my-flask-app:<version_tag>
- docker push username/my-flask-app:<same_version>

5) Now, go to Google Cloud Console, then to Cloud Run, and click on the Create Service Button.

6) In the `Container image URL` field, enter the following URL (after making the necessary adjustments):

`docker.io/username/my-flask-app:<version_tag>`

7) In the `Service name` field, enter `foodwise`.

8) In the `Authentication` section,
check the box for `Allow unauthenticated invocations`.

9) In the Minimum number of instances section, select 1.

10) In the Container section, ensure that the Container port is set to 12345.

Creating a debug APK for the application

1) Go to the root of the application in the terminal and run this command:

```
react-native bundle --platform android --dev false --entry-file index.js --bundle-output android/app/src/main/assets/index.android.bundle --assets-dest android/app/src/main/res
```

2) Go to android directory:

```
cd android
```

3) in the terminal run this command:

```
./gradlew assembleDebug
```

4) The APK will be created in

Application/android/app/build/outputs/apk/debug/app-debug.apk

Technical Explanations and Important Notes for the IoT Device:

The device is a Raspberry Pi 4 running the following operating system:
Raspbian GNU/Linux 11 (Bullseye).

The device also has a touchscreen connected via HDMI and a scanner connected via USB.

To give the device access to Wi-Fi, you need to connect it to the local Wi-Fi network through its operating system.

To do this, you need to connect a keyboard and mouse to the device. After that, turn on the device, and once the operating system has started, click on the Wi-Fi icon and select the Wi-Fi network you want to connect to.

After the device is connected to Wi-Fi, you can access it remotely using a program called VNC Viewer. You'll need the local IP address that the device received from the local Wi-Fi network.
You can find the local IP address by running the command `hostname -I` in the terminal of the operating system running on the device.

When connecting, you will be prompted to enter a username and password. The username is `codeCrafters` and the password is `12345`.

On the device's operating system desktop, there is a file named `app.py`, which is the program running on the device.

To ensure that the program runs automatically when the operating system starts, you need to make sure that there is a file named `autorunmyfile.desktop` in the file system at `/home/codeCrafters/.config/autostart`.

The file should contain the following content:

```
[Desktop Entry]

Exec = bash -c "cd /home/codeCrafters/Desktop && sudo python3 app.py > /home/codeCrafters/Desktop/app.log 2>&1"
```

This file runs automatically when the operating system starts, giving the operating system the instruction to navigate to the directory where the `app.py` file is located, run it with full permissions, and direct all the program's output, including logs, to a file named `app.log` on the desktop.