Coursework 3: The IoT Fridge Robert Taylor

Instructions for use

Run the executable shell script 'fridge.sh' from the same folder as 'fridge.json', 'fridge.sql' and 'iotfridge.py'. This will create a database and save it to a file file called 'fridge.db'. This database file will then be passed to the 'iotfridge.py' program, along with a JSON file of pre written API requests called 'fridge.json'. This will then update the newly created database, based on the API requests.

Design

Before implementing anything it was necessary to spend time thinking about what an internet fridge might actually be used for, and what features a user might want. The main areas considered are briefly detailed below. Not all of these features have been implemented due to time constraints.

Functions an internet enabled fridge might have

- The ability to keep track of what is in fridge
- The ability to alert users about expiry dates, possibly via email
- The ability to create and store shopping lists, both automatically and manually
- The ability to order food from on-line retailers
- A list of favourite items which are frequently used
- The ability to store user preferences

What items will be stored in the fridge?

- Packaged food (maybe with bar-codes)
- Liquids/solids (these require different measurements for different products)
- Loose produce (eggs, vegetables, fruit etc.). Again these require a different measurement.
- Medicines
- Open packages, closed packages (this will probably affect expiry dates)

Where might they be stored in the fridge?

- fridge door
- fridge shelf
- freezer

Role of APIs compared to database

To design the system it was first necessary to gain an understanding of the role of APIs. APIs are assumed to be used for manipulating limited parts of the fridge database in a safe manner. For the purposes of this design, the APIs are designed to allow a user to perform basic operations on all tables in the fridge, without having any knowledge of the database schema etc. They are designed to return only the data that might be needed for programming purposes.

Security

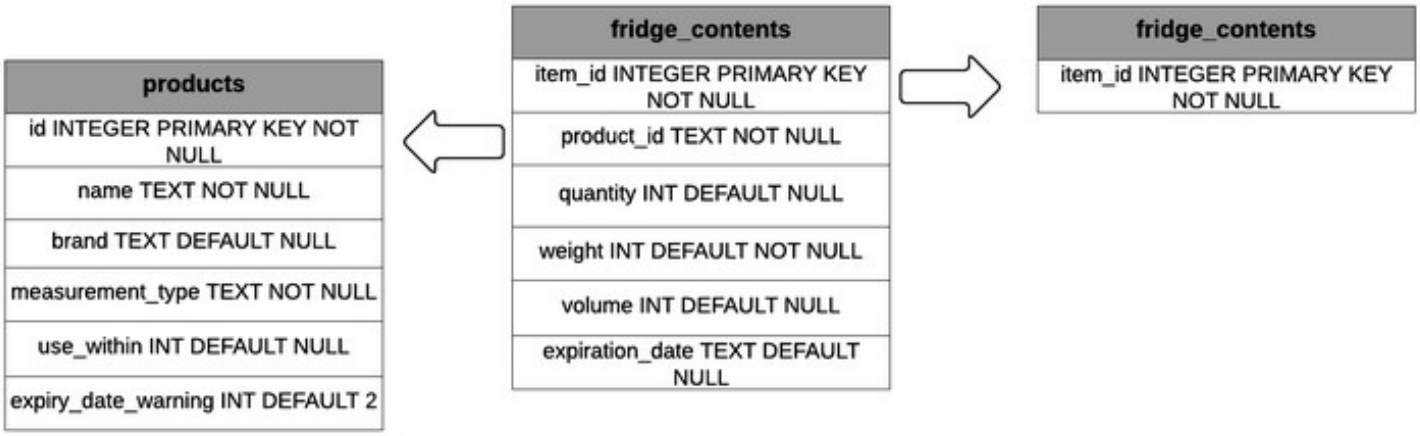
It is assumed that as long as the APIs are designed in such a way as to avoid users having access to dangerous SQL keywords (DROP etc.), the database itself should be relatively secure. For example, while the system doesn't currently store payment information. If this were included then no API would

return any sensitive data. References to user information would be purely anonymous.

This assumption relies on the assumption that it is not possible for anyone with malicious intent to access the database itself.

Implementation

Database Schema



API Documentation

All APIs return JSON data, either to inform the user that a request has been successful or not, or to pass useful data that can be used elsewhere in the application.

Products

These APIs refer to and manipulate the products table. This table stores information about specific products that the fridge is aware of. These need to be entered individually prior to instantiating items which correspond to a product.

listProducts

Returns the id primary key and unique name and brand combination of all products in the products table.
This is designed to be sufficient information for an interface designer to effectively program with.

JSON request format: {“request”: “listProducts”}

JSON data returned: {response: [{ “brand”: “brand name”, “name”: “product name”, “product_id”: “INT”}, {“brand”: “brand2 name”, “name” “product2 name”, “product2_id”: “INT” } …], “success”: true}

addProduct

Adds a product to the products table, including useful information to be used in later operations. The measurement type is specified (quantity | volume | weight), as well as the integer duration in days after

opening that a product must be used by.

JSON request format: {"request": "addProduct", "name": "product name", "brand": "product brand", "data": {"measurement_type": "quantity | volume | weight", "use_within": INT} }

JSON data returned: {“response”: OK, “success”: true}

removeProduct

Removes previously added product from products table.

JSON request format: {"request": "removeProduct", "name": "product name", "brand": "product brand"}
JSON data returned: {“response”: OK, “success”: true}

Fridge Contents

These APIs refer to and manipulate the fridge_contents table. This table stores information about specific items that are currently in the fridge. An item can be viewed as an instance of a product. Items can be added, removed and opened.

listContents

Returns relevant information about items currently stored in the fridge. Among other data, the item_id is returned for use in subsequent API calls.

JSON request format: {"request": "listContents"}
JSON data returned: {“response”: [“item_id”: INT, “expiration_date”: “YYYY-MM-DD”, “amount”: [“amount”, “measurement_type”], “name”: “item name”]}

insertItem

To insert an item you must specify the measurement type in the request. For example, below the product is something measured by quantity, as such this is what is passed to the request. This information is then routed to the correct field by the python file. The measurement fields that are not used are set to null automatically.

JSON request format: {"request": "insertItem", "name": "eggs", "brand": "Lovely Eggs", "data": {"use_by": "YYYY-MM-DD", "quantity": "INT"} }
JSON data returned: {“response”: OK, “success”: true}

removeItem

JSON request format: {"request": "removeItem", "item_id": INT}
JSON data returned: {“response”: OK, “success”: true}

openItem

This targets a specific item in the fridge_contents table and alters its expiry date to take account of the item being opened. For example if a bottle of ketchup was opened it would replace the expiry date with the 'use within x days of opening' value added to the current date.

JSON request format: {"request": "checkDates"}
JSON data returned: {“response”: OK, “success”: true}

Favourites

These APIs refer to and manipulate the favourites table. This table is where favourite items may be stored. This table is likely to be useful for improving the day-to-day usability of the fridge.

addFavourite

JSON request format: {"request": "addFavourite", "product_id": INT}
JSON data returned: {“response”: OK, “success”: true}

removeFavourite

JSON request format: {"request": "removeFavourite", "product_id": INT}
JSON data returned: {“response”: OK, “success”: true}
listFavourites

JSON request format: {"request": "listFavourites"}
JSON data returned: {“response”: [“product_id”: INT, “product2_id”: INT…], “success”: true}

Expiry Dates

checkDates

This checks the current expiry date of all products in the fridge and compares them against the current date. Any products that are within the expiry date warning period (set to 2 days by default) will be returned.

JSON request format: {"request": "checkDates"}
JSON data returned: {“response”: [{ “item_id”: INT, “name”: “item name” }, { “item_id”: “item2 id”, “name”: “item2 name” } …], “success”: “OK”}

Evaluation

There were a number of features that were not implemented due to time constraints as well as many areas of the system that could be improved. These are briefly detailed below:

- There is no internet connectivity. Given more time the system could be altered to generate email alerts/ ordering products on-line based on expiry dates and user preferences.
- There is currently no explicit integration of bar-code information. Again with more time this would be incorporated. Currently the system is designed largely with a touch-screen in mind.
- The current unique constraint on the combined name/brand per product might become a limitation.
- Unsuccessful API calls are not reported and error checking is minimal. For example it is possible to try and put an item in the fridge that has not been specified in the products table. The user is not warned about this and the item will not be inserted. This is far from ideal.
- Opening items at the moment can result in expiry dates being extended past a safe limit as currently the API that deals with this works on a very simple calculation. This should certainly be improved.
- There is also not a consistent use of ids vs. other field values for accessing tables via APIs. This should all be based off ids in order to make operations more stable and consistent. This has only be left as it is due to time constraints.