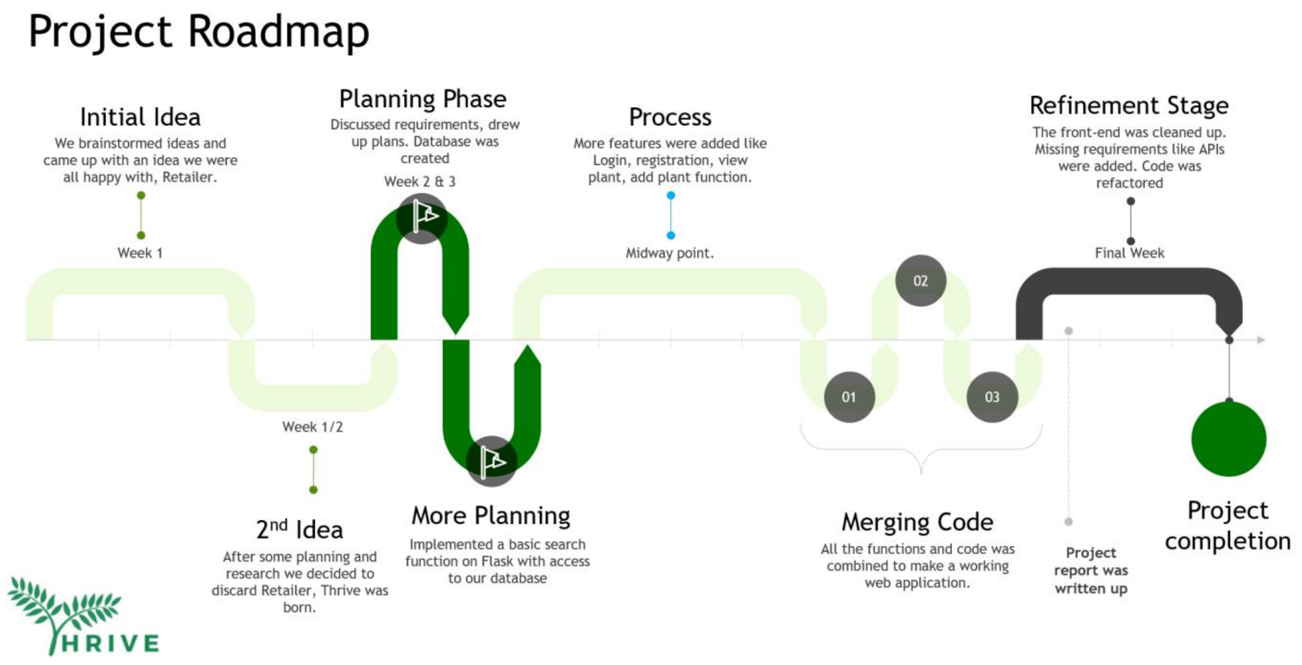
INTRODUCTION

Our object for this project was to build a website named "Thrive". Thrive is a place where individual users can store, keep track and add plants that they own to their own personal database via a profile they create when they sign up and login. The aim is to have detailed information and care instructions for the plant provided enabling the user to create the best environment for their plants to "thrive"!


Project Roadmap
Initial Idea
We brainstormed ideas and came up with an idea we were all happy with, Retailer.
Week 1
Planning Phase
Discussed requirements, drew up plans. Database was created
Week 2 & 3
Process
More features were added like Login, registration, view plant, add plant function.
Midway point.
Refinement Stage
The front-end was cleaned up. Missing requirements like APIs were added. Code was refactored
Final Week
02
01
03
Week 1/2
2nd Idea
After some planning and research we decided to discard Retailer, Thrive was born.
More Planning
Implemented a basic search function on Flask with access to our database
Merging Code
All the functions and code was combined to make a working web application.
Project report was written up
Project completion
HRIVE

SPECIFICATIONS AND DESIGN

Our main criteria and specifications for this project were:

- **Login page:**

The login page is the landing page for our application. It provides two text fields for the user's username and password. The command button "login" initiates the SQL query to check whether this user exists in our database, and whether the credentials are correct. If any of the boxes are left blank, this will return an error message to the user.

- **Registration:**

If the user doesn't have an account, the user has the option to make an account and sign up. When registering, the user is prompted to input their username, password, email and city. An API has also been used to list all cities with a population over 5000. This allows the user to select their closest city from an extensive list. If the user attempts to sign up with a username or email that has already been used, an error message will be returned.

PROJECT BACKGROUND

Our project is a web application. The user can create their own profile by registering and logging into Thrive which enables them to access their own personal database of houseplants.

Within the application, the user can view all the plants stored in the database, and select from the extensive list, picking out the ones they own, we have also included pictures in case the user isn't aware of the plant name. By doing this, the user can keep track of all the plants they own via their own profile, and find out information of how to care for their plants.

- **Profile page:**

A profile page displays all the plants the user has added to their database. The user will be able to hover over the plant which would display information needed to care for it.

- **API:**

An API to display cities for the user to pick from, and a weather API to display the weather, so the users may know their weather conditions in their area to maximise the care for their plants.

- **Database:**

A SQL database to store user information and login details. This will allow us to implement action methods by using SQL queries for the registration and login page, which can be initiated by the command buttons. The database should also have a column for salt, and a hashed_password. The user table will include ID, username, hashed_password, email, city and salt. A primary key would be ensured on ID, to support registration and login the database table should have columns for user passwords variables, a method for getting the password for a username and a way to add a password for a new user. It should also contain columns for emails and usernames. Since MySQL would be used, a Primary Key constraint would be added to ensure the text fields aren't left empty.

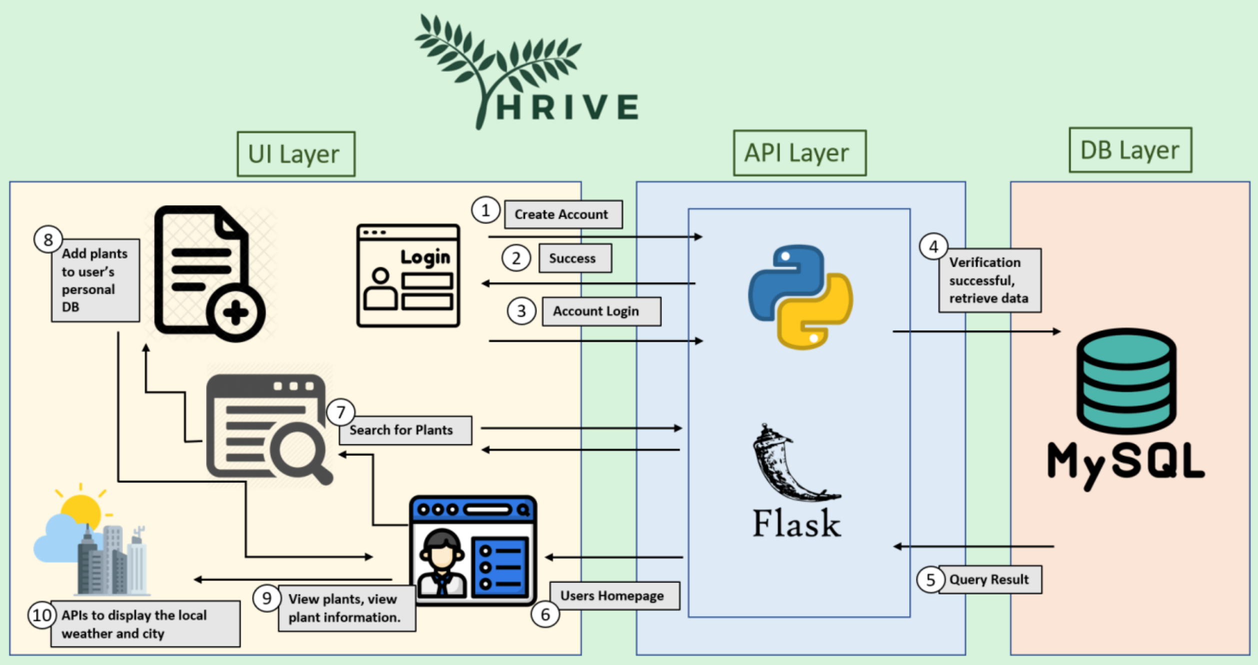
- **Search function:**

A function that enables the user to search through the database is also important. We also wanted the function to have the ability to display all the plants as well searching for specific houseplants. This would be linked to the API.

- **Web application:**

We decided early on that our project was going to be a web application. Flask was our first choice due to how flexible and how much more accessible the framework is. It also provided a lot of useful tools to access our database in MySQL. We briefly touched on Django as well however decided against it as there is more of a learning curve compared to flask.

DESIGN & ARCHITECTURE


HRIVE
UI Layer
API Layer
DB Layer
1 Create Account
2 Success
3 Account Login
4 Verification successful, retrieve data
5 Query Result
6 Users Homepage
7 Search for Plants
8 Add plants to user's personal DB
9 View plants, view plant information.
10 APIs to display the local weather and city
Login
Flask
MySQL

IMPLEMENTATION & EXECUTION

Development approach & Team member roles.

The approach we decided to follow was the waterfall method. As we are all working remotely, and are available at different times of the day, this approach was more favourable than agile, as we would not be able to commit to regular progress meetings.

We firstly evaluated the project requirements by researching similar apps and websites that already existed. We took note of what functionality and featured these other websites offered, and thought about what was happening behind the UI. We used this to build a list of our own requirements, and decided on which ones we definitely wanted to implement, and some that would be nice to implement if there is time to do so. Moving onto the design, we drew up a map of the web app, and all the pages that would be needed. We would need a registration page, log in, homepage, a page to view different plants to add to your own database, and a page to view your own details and your own plants.

To begin the implementation process, we began by creating a 'thrive' database and our functions. We began with a simple sign up, login form and homepage. Once this was implemented, we built on it and added a profile page, where the user could view their details, and update them if they wanted. All this information came from the 'user' table, through mysql queries in the python functions. A search function was added, which searched the table 'plants' we had created. We would have liked to have used an API for this which contained info on different plants, however we were unable to find one which was suitable. Another page was added that enabled the user to add plants to their own user_plants table to keep track of what plants they had. The aim was to combine both the search function and the add plants function to make a 'search and add' function, where the user could search the plants table, select a plant, and add it to their own user_plants table.

However, it was more user friendly and visually pleasing to get rid of the search function, and to simply display all of the plants from the plants table on one page, with an option to click and add them. The profile page was further updated to display the users' plants, and the update detail's function was removed due to security concerns.

Having this function as it was meant that anyone who gained access to the users account would be able to update their email address, username or password without any secondary verification. While it is a useful feature, it felt best to remove the risk at this stage. This could be added back on in future developments, once a second layer of security is added, such as MFA.

The signup and login were further developed, with a password hashing function being added. Storing the passwords in the database in plain text is a huge security vulnerability. Any cyber attacker who is able to gain access to the database would have instant access to all users emails and passwords. Data leaks caused by poor security measures have resulted in companies facing reputational damage, financial loss through needing to compensate individuals and pay fines, as well as the risk of the attacker being able to gain access to the entire system and corrupt databases and files, which would have a devastating impact. Using a hashing algorithm will 'scramble' the plain text password that the user inputs into a hash value, making it indecipherable to read. However, simply using a hash is not enough, as many hashing algorithms have been cryptographically broken.

Cyber attackers could easily find users passwords by running them through any number of free online password decoders. This is where the 'salt' comes in - which is where a string of random characters is added to the password before it is hashed. Unless the attacker knows what the salt value is, their job of trying to crack the password is now a lot more difficult. Some decide to use the same salt for each user password, however to further secure our database we set our secret key to 'os.urandom(16)' , which randomly generates a unique string of mixed bytes for each password.

As an additional feature, we wanted the user to be able to see the weather on their home page. This could help with caring for the plants, and ensure they are getting enough warmth/light. It also meant we could use an API, as we were unable to find one for the plants.

We used a weather API which displayed the temperature and weather conditions for the specified city. In order for us to obtain the user city, we added a 'city' field to the registration page, and a corresponding row to the user table. We manually added various major UK cities into the drop-down list for the user to select. We did not like this idea, as it was a lot of code and maintenance, and would likely mean that the user wouldn't be able to find their local city. To overcome this, we explored using the users IP address to find their location. We first tried using the python socket import, which would find the IP address, then we fed this into an IP address API which would extract the user's city from their IP address.

This city would then be used to find the weather from the weather API. We decided to get rid of the IP address for various reasons. The first is that it may worry users who have privacy concerns – many people don't like sharing their location unless they're specifically asked and agree to do so.

The other reason is that it isn't a reliable way of seeing where a user is – if they are using a remote web server, or VPN, it would give them the wrong location. The user needs to see the weather for where their plants are stored, i.e., at home – so if they are travelling, this feature would be rendered useless! We opted for using a city API to populate the drop-down list on the registration page, which would then populate the field on the weather API to display the users' weather for their home city.

To improve the user experience, and to begin building our brand, we used bootstrap to bring the website to life and make it look great! A green colour scheme complimented the plants' theme and the logo that was created. Images of plants were added to the plants table, meaning the user could see a picture of each plant to help them identify ones that they owned, but did not know the name of.

Finally, the functions were tidied up as many were very long and difficult to read. We used OOP to remove the repeated code, and split it into its their own classes or functions that could be called where needed, such as the redirects, or functions that fetch users accounts information.

This helped to significantly improve the appearance of the code and to make it much more readable.

The final product consists of:

- sign up page containing username, email, password and city fields.
- login screen
- homepage containing a welcome message which pulls the local weather information from an API, and the users selected city
- search page displaying all plants from the plants table, along with pictures, more info button, and an option to either add or remove the plant from the users profile
- profile page displaying current users email, username, and their own plants
- option to log out

Tools and Libraries

The main tools that make up the backbone of our project is Python and MySQL workbench. Pythons' vast library support makes it ideal as it eliminates the need for writing codes from scratch. The libraries used to run Thrive are:

- Hashlib: To encrypt and generate a hash for the passwords used in Thrive.
- Requests: used to convert JavaScript Objection Notation into Python lists and dictionaries.
- Mysql-connector: to provide connectivity to the MySQL server and access the database.
- data into fixed length hash values.
- re - regular expressions - helps python to ensure that inputs match with what sort of data we are expecting, for example in the email field, it should contain an @ and a full stop
- uuid - unique universal identifier, was used as part of our password hashing function. uuid generates random objects, and ensures that cryptography is secure
- mysql workbench - mysql connector
- flask

We used MySQL for its scalability and flexibility as well as it's the ease of use when integrated into Python. For the front-end portion, we opted with going for Bootstrap and HTML

It’s high-performance query engine makes it perfect for high-traffic web and data warehouse storage.

Implementation process (Achievements, challenges, decision to change something)

Getting the user interface working was definitely a big achievement, it was important for us due to it being the first thing the user sees when interacting with Thrive. We went through several versions of the HTML frontend before settling on a final look. Another big achievement was putting together all the various features like the general login page, the search, add to own database, view database. combining them all felt like a huge milestone because up until then each task had been completed separately from each other. Over the course of the project Thrive had to go through many iterations. We had planned for more features, making it more of a plant care orientated website, but we had to amend that due to time and lack of experience. We had to discard our very first idea, Retailer for similar reasons. Retailer was meant to be an app or program that lets you receive paperless receipts when shopping without having to give out your email to the merchant.

Agile development (did team use any agile elements like iterative approach, refactoring, code reviews)

Even though we mainly used the Waterfall approach we did touch on some agile elements. One of them being the iterative approach, we used it a lot since we continuously had to revise and cycle through project ideas, plans and development phases as we progressed with the project. Old ideas that were no longer doable were set aside after analysing. Features had to be re-designed and changed, the lighting function and watering reminders we first planned didn’t fit into our final design.

Through trial and errors, we realised we preferred bootstrap to CSS and adjusted our code according. We also refactored a lot of our code via decorators in order to make it neater and cleaner looking. To do this, we used inspiration from the Instabook project from our class, and created decorators which called the functions “should_be_signed_in”, and “should_be_signed_out” making it easier to ensure the users are only accessing the correct pages when signed in/signed out.

Implementation challenges

One of the first implementations challenges we encountered was the lack of suitable APIs. Initially we planned to use an API to fetch plant data to fit the API requirement but we found that all the ones available didn’t fit our needs. We required one specifically for houseplants however the only ones we could find didn’t have information on houseplants or it would fetch data we couldn’t use. So, we decided to create our own database on MySQL Workbench and populate it with the information we needed to get around that constraint. Another challenge was also the IP logger, as we had to remove this as it was posing a cybersecurity risk. Thrive runs on a local network, so we couldn’t implement publicly routed IP addresses that can be mapped to cloud servers.

Building a frontend interface was challenging as all our classes and learning material mainly centred around backend development. Those of us with no or little experience with HTML and JavaScript struggled a bit which made the different abilities and confidence levels with coding stand out. We also ran into problems on the backend side trying to add plants to a user’s own database due to our inexperience with Python. We got around this issue by consolidating all of our resources and time into tools and frameworks we were already familiar with instead of overstretching ourselves. The tools and packages we touched on briefly were discarded when they didn’t work for us. One of them being PyGames, the interface and implementation was really easy and intuitive but we realised it wasn’t made to run as a web application so we wouldn’t be able to implement it with the rest of the web site. We tried using SQLAlchemy for the login portion of the project as it had a built-in tool for implementing logins but we realised that learning entirely new queries would take too long. Time was a huge constraint to consider. We did plan for our idea to have more features but they were too ambitious to fit into the timeline we had set out. We had to refer to the deadline a lot when working on this project, accounting for things like homework and assignments that needed to be completed alongside the project.

Another issue we faced is less of a technical issue and more to do with the problem of working together remotely and not being able to see each other in person. We had different team members working different hours which meant we couldn't always meet up at the same time. We also had to consider the team members that were working full time who didn't necessarily have as much time to work on the project. One of the ways in which we got around that issue was creating a private server for the team in order to keep each other posted on issues and current progress as well as plan the next meeting dates. We also set up a notion page with sections divided into to-do lists, resources, progress trackers and any notes we made throughout the project. We found this super helpful as the layout really lends itself for updating progress, we could leave screenshots and bits of code updates as well as detailed instructions via comments.

Since we deal with personal information like user credentials and emails, we had to figure out a way to keep the credentials secure within the database. We had to consider the inherent vulnerability that comes with using MySQL as it's easily exploitable via network attacks that can compromise the server. We used query parameters to protect our database against SQL injection attacks. One of the ways in which we mitigated the risk was figuring out a way to hash the passwords in order to store them securely.

TESTING AND EVALUATION:

In order to test our application, we created a directory named "tests", where we implemented unit and integration tests:

The first test we ran was an integration test for our CityApiManager. With this, we were testing two things:

- If the structure of our function worked as expected.
- If the connection to the external service (City API) was running correctly.

In addition, we ran Unit Tests against our

PlantManager:

- We created a mock database to run these tests, we created a setup using "enter", and a teardown using "exit", whilst also setting up the connection to the cursor. Within get_mock_db_cursor, we also created a set up using "enter", and a tear down using "exit". We also included execute with query and parameters, and specified fetchone and fetchall which uses the mock db cursor to feed the data back that we specify. We used this to test the following functions: get_all_plant_data, get_plant_data_for_user, get_plant_names_for_user. These tests tested whether the functions transforms the data correctly and returns the correct data.

All of our tests ran successfully.

System Limitations.

Within our project, our main limitation is that we are running the application from a local network, and Flask-based systems have the potential for security risks, meaning there was a limit to how much mitigation we could do against security concerns.

Had time constraints not been an issue, we would have liked to have developed this project further. Ideally, the profile page would have shown the plant care information as this makes the most sense for the user. At the moment, they are having to return to the search page to see this info. Building on this, it would have been good to have reminders on when to water each plant to pop up on the homepage. We would have also liked to have integrated the functionality with the weather API, such as telling you to move plants into the shade if the weather was sunny for example.

A 'forgotten password' option would have been very useful to implement, which would enable the user to reset their password via a link sent to their email address.

This would mean that we could have re-integrated a more secure version of the update details function for the user to update their credentials while they are signed in, as it could sent a confirmation email to confirm it is the user trying to make those changes. We could also allow the user to update their other details including their city, as they may move house, along with their plants!

CONCLUSION

We aimed to create a web application, Thrive, that enabled users to access, add and update information on their plants via their own user profile. We intended to include a plant watering reminder and a feature for users to view plant health instructions from their profiles. Although not everything we planned to create was able to make the final result, we have made a functioning web application that runs the main features that we wanted.

Our aims were closely aligned with the project specification requirements because we have incorporated a database, backend Python code and OOP principles and libraries. We initially intended to use an API instead of a database of plants created from scratch. However, circumstances led to us integrating a weather API which we think suits our project theme very well.