COVID Calculator

Team 5: Louise, Emily, Maysa, and Jess K

INTRODUCTION

Until recently it was thought that viral pandemics were a once a century event. Recent events have proven otherwise.

COVID-19 has demonstrated the enormous impact that pandemics have both at an individual level and also on communities and nations across the world.

As COVID-19 is now endemic in the UK, information about it is dropping out of the mainstream media. This means it is becoming harder to access up to date information about covid rates and this makes it more difficult for individuals and organisations to make informed decisions about their behaviour and risk management.

With the end of lockdown, routine vaccination and the changes in laws and attitudes around mask wearing and social distancing, giving people access to the information they need to make their own choices is vital. We therefore decided to create a user friendly application that allows users to search for a UK location to access easily understood COVID rate data.

This application could be used by individuals or organisations to help them make choices about how they manage their day to day COVID risk. This would not only help them avoid becoming unwell, but also could have a positive impact on the public health and wellbeing of the country as a whole. As discussed at the end of this document, the functionality of this application could also be extended in a number of different ways.

BACKGROUND

The aim of this project was to help empower individuals to make choices which could potentially reduce their risk of catching and spreading COVID-19.

The COVID Calculator calculates the rate of positive tests per 100,000 population at the user's choice of location over the latest available 7 days (Figure 1). It also calculates the UK wide rate for the same period for comparison. This is done by prompting the user for their locality: Upper Tier Local Authority (UTLA) or Nation. The programme then gives users information about the number of cases in that area and risk stratification. The predictor also gives the user the most up to date advice for their locality (using national guidance), gives them links to more information and gives them the option to save their data.

SPECIFICATIONS AND DESIGN

For this project we decided to use Python for our backend, MySQL for our databases, and Javascript for our frontend (Figure 2).
Our system has a user interface which allows the user to create a user account, or log-in to an existing account, and to input the area they are interested in. The input request is assisted by a “Did you mean…” functionality, to assist the user in finding their location. This input is processed by the Python backend, which requests the COVID case numbers for that location from the API. Simultaneously, population information for the same area is retrieved from the MySQL database by the Python backend.
The backend calculates the rate of new cases per 100000 population using the latest available 7 day’s case numbers and the population for the search area.
The rate and a risk stratification (based on the CDC community stratification table (which can be found here: https://www.cdc.gov/coronavirus/2019-ncov/science/community-levels.html) is relayed back to the user using the frontend.
Finally, the user is given the options of accessing local advice (based on which nation their search area is in), and if they would like to store their information for future reference. This stored information is given to them next time they perform a search, allowing them to see if there has been an increase or decrease in their local rates, and hence adjust their behaviour as necessary.

IMPLEMENTATION AND EXECUTION

Our team split into 3 sub-teams. Louise worked on the MySQL database and the Python query codes to go with this. Emily and Jess worked on the Python backend including the API connection, predicting user input via the Levenshtein module, and the main file. Maysa designed and implemented the frontend in Javascript.

To keep track of progress we used Trello (https://trello.com/b/vd2ppEVP/covid-calculator) and communicated through a Slack channel
(https://app.slack.com/client/T03E122UDRU/C03QGFYDSLQ.
We also had weekly Zoom meetings which were run with an agenda, and from which outcomes were distributed after each meeting. These techniques allowed us to communicate, keep on track, set timescales and targets, and assign work in a flexible manner. We chose to use an agile approach, and so we would discuss our individual progress and difficulties in these meetings, in a similar way to a daily SCRUM. We also had weekly code-reviews to change files or add new features - An example of this was our decision to build in a log-in and registration functionality, which was initially identified as a “nice-to-have” future extension. This way of working made our project flexible and easier to manage, and also allowed us to support each-other well.

What went well

- We initially came up with more than one idea. We quickly narrowed it down to our final project which stood out due to its relevance to the events of the last 2 years, the ongoing current impact of Covid-19 and the high risk of future viral pandemics.
- Our overarching concern at the start of the project was whether we were able to complete and build what we had agreed, so we decided to start with a small and manageable project and to review and adjust the build as we went along. This flexible approach allowed our project to ultimately contain more functionality that we had planned in the initial stages. Examples of these are the user login/ registration functionality, and the “Did you mean” functionality.
- We decided which team members would be responsible for which aspect of the build based on their experience and interest in different areas. This meant that all team members were fully involved and engaged with the project.
- Despite working remotely and having never met in person, we worked together well, communicated well and all team members gave 100% effort to the build..
- During the build, we had regular meetings which were timetabled from the start of the project and which we all attended and engaged with. These meetings allowed us to discuss difficulties in a prompt manner, and provide any necessary support to each-other.

Implementation challenges

- The ONS database and government API had different data structures and also different UTLA names. Therefore, we adapted the ONS database to cater for the API and adjusted our database accordingly. The following piece of code was used to automatically identify any locations in the database that weren’t compatible with the API, so that we could manually remove them.

```
# Fetches all locations from the database
database_locatons = get_locations()

# Iterates through the locations
for location in database_locatons:
   # Try to use the API with the current location
   try:
       get_cases_by_location(location[0], 'utla')
   # If the API throws an error, location is invalid
   # These locations are printed out, so that we know which must be
removed from the DB
   except Exception:
       print("Not working: {}".format(location[0]))
```

- As people were editing the same Python code on different branches and committing changes, occasionally we had merge conflicts that needed to be resolved.
- The large amount of User inputs required by the program presented problems with preventing any invalid responses. It was essential to handle these correctly to prevent any unexpected behaviour in the program.
- When code was updated, unit tests also needed to be regularly reviewed and updated.
- We had to review and manage bugs that cropped up as the build developed.

Python

With GitHub (https://github.com/jessicakan789/team5) and PyCharm, we used version control to work on code for our backend (Figure 3). The dbconfig file holds information about the user, host, and password for the SQL database which the user is prompted for. The dbconnection file allows the user to access the SQL database via the MySqlConnector library. These functions are imported for population, dbutils, save_search, and area_advice which contribute towards the running of main.

Population gets the population statistics from the table which holds information from the ONS (Population estimates - Office for National Statistics). Save_search acts like a cookie and adds the most recent search information by the user to the database. Area_advice gives the user an option to view government advice about COVID-19 in their nation stored in a text file for each nation. Db_utils allows the database to be updated and requires hashing which hashes the password as a hexadecimal. Save_search and dbutils functions are imported for login which prompts the user to either sign in or create an account. Yes_no_input is a helper function for requesting a Yes/No user input. This function returns True or False depending on the answer received, whilst also ensuring a valid input is received. .

For the API side, we used the Levenshtein module (in file predict) to predict what the user meant if they spell a location wrong. This makes the programme easier for users to use. Then the API file connects to the government API that holds information on COVID-19 rates (Open Data API Documentation | Coronavirus in the UK). The connection requires the json and requests libraries. This returns the rate for the past week to the user and is used to calculate their risk.

To ensure the program's quality, a major focus was ensuring the code is robust in accepting user-inputs. More specifically, this means that the code should be able to appropriately handle any invalid inputs, without running into unexpected behaviour. As a basis, all input prompts were used along with a While loop to continually prompt for another input, so long as a valid input has not yet been received. Input strings were also stripped of white space, and converted to lowercase, so that their values could be safely compared to the expected cases.

The main functionality of the program is encompassed in the user selecting their location from a list of UTLA's and Nations as stored in the database. This presents several problems, in that the user may misunderstand the local-authority names, or simply misspell them, causing their input to be incompatible with the API. The program was designed to overcome this issue, by immediately checking the input against all the valid locations stored in the database. If the input cannot be recognised, the user will be informed, before being prompted to try again.

Whilst this is a viable solution, we tried to further assist the User by also providing a built-in “Did you mean...?” functionality in the predict.py file. This utilises the Levenshtein module to identify stored locations which are *‘closest’* to the input word, as measured by the Levenshtein Distance – essentially the number of single character edits made to one word, to get the other. If any locations are identified as being similar, as decided by the threshold parameter, the user is presented with the options, along with the match probability, to either confirm or reject. This generally makes the program much more user-friendly, and is more reflective of the movement towards Natural Language Processing, and other Machine Learning techniques, in the real-world.

Another issue we had identified was surrounding the yes/no inputs, required several times throughout the program. Whilst these inputs are more straightforward to deal with due to the low number of *accepted* inputs, we found that their high frequency had caused the code to become quite messy and redundant in places. To overcome this, we created a separate yes_no_input() function, which was then reused multiple times within our main code. This makes the code much more readable, reduces repetitions, and generally follows common best-practices much better.

To ensure our code included some object-orientation, the login system is class-based. This allows us to isolate different types of users, and group functionalities that should belong to each, making the code more reusable.

The User base class defines the attributes belonging to a User instance, namely their username and password. Using inheritance, two child-classes have been derived from this base class; the NewUser class and the ExistingUser class. These classes define their own methods and member variables, to perform the actions necessary and unique to themselves.The NewUser class has member functions is_unique, common_password_checker, and password_regex, which essentially ensure that the username and password chosen upon registration of a new account are acceptable. We have defined our own exceptions, which are raised if any problem is found. For example, we have defined a RegexError exception, which is raised when the password entered does not follow the defined REGEX rules. These methods are brought together in the make_new_user function, which interacts with the database to register new users if all validity checks are passed. Note that for an extra layer of security, passwords are hashed before being stored.

In contrast, the ExistingUser class defines a login member function, which fetches the stored account data from the database, and checks that the hashed passwords match, before allowing account access.

By using a Class based login system, the functionalities can easily be extended as the system becomes more complex. For example, in the future we may also wish to define methods for a

NewUser to add a password hint to their account upon registration, or for an ExistingUser to reset their password.

MySQL

The 'Population' database was created using MySQL Workbench.

The data in the database is derived from https://www.ons.gov.uk/peoplepopulationandcommunity/populationandmigration/populationestimates. This data was converted to a csv. The csv was formatted to allow for correct insertion into the database tables and for use by the python part of the program.

The database comprises three tables (See ERD Figure 4).

- cutdownpopulation - contains the populations for the areas which are searchable on the API.
- user_info - contains the username and encoded password for each user.
- user_area_data - contains the last searched area and calculated rate for each user

Table Keys:

- cutdownpopulation : Primary Key *'Area'* referenced as a Foreign Key by the *last_area* column in *user_area_data* table
- User_info : Primary Key '*username*' referenced as a Foreign Key by the *username* column in *user_area_data* table

Once the database had been created and populated, MySqlConnector was used to create a connection between the python backend and the database. This allows the backend to communicate with the database in the following ways:

- To find out if a *username* is already in the database before an account is created(and therefore unavailable for use, avoiding the duplication of usernames. This is necessary both for security and also as username is a primary key in one of the tables).
- To insert a new users *username* and encoded *password* into the *user_info* table
- Simultaneously to insert a new users *username* into the *user_area_data* table
- To pull the user's encoded *password* back from the *user_info* table when they log in to allow for checking
- To update the *user_area_data* table when a user wants to save information from their searches
- Returning the user's last search results from the *user_area_data* table
- To identify from the *cutdownpopulation* table which nation users search is within so they can be given the correct advice for that country.

To store the user's password when they register, the MD5 function from the Hashlib module is used. This function creates a repeatable 128 bit hash value. The user's password is saved in the database as an MD5 hash of the original password. When a user logs in, their password attempt is encoded using the MD5 function. The MySQL connector retrieves the user's saved password hash and compares the two. If they match, the user is allowed access to the system.

Web-Application

We created a small web-application version of our programme so that users could directly interact with it. This application only exists on the front-end branch in Github. Currently, the application accepts a location and returns the rate (population/risk) and the risk-level. The application was created using FastAPI, Uvicorn and React for the user interface. FastAPI is a web framework for building API's with Python(v3.6). Uvicorn, an ASGI, HTTP web server program is required to run this.

Uvicorn handles requests from the browser and processes them. It hands this over to FastAPI which has a defined call-back function for that specific route. It then executes it and returns information from the database to Uvicorn. Uvicorn responds to the request and sends the response over to the browser in json form (main_server.py) (Figure 6).

Main_server.py imports fastAPI and HTTPException for error handling. Get_rate_by_location and calculate_risk are both imported from API.py to calculate the CDC Covid-19 community level. CORSMiddleware (cross origin resource sharing) is also imported to overcome same origin policies and the wildcard(*) is used to declare that all requests are allowed (Figure 7).

The front end consists of a React application (app.js) which is a Javascript library that allows you to easily build user interfaces (Figure 8). Nodejs is used to run it in development but wouldn't be required in production. It uses a MUI, a react component library which helps with pre-made components and styling to overcome the time constraints we were facing. In this simplified application, the box, textfield and button components were used. UseState and useCallback are hooks; functions that allow you 'hook' onto functions. UseState allows you to save and update states between re-renders.

Limitations

The main limitations of this application were caused by time constraints. This means that there is currently limited functionality as not all the functions have been imported. Additionally, there is a lack of focus on accessibility and this application isn't suitable for screen-readers or users with any accessibility issues.

Another major limitation is that due to the use of two API's, changes in one would mean having to make changes in the other which requires maintenance and results in a significant amount of technical debt.

TESTING AND EVALUATION

- During the coding process we used unit testing (Figure 5) because it was easy to implement and check functions. Modifications were made to our python code to cater for unit testing and to make sure all tests passed. For example, we added in try and except clauses, and return statements.
- As some of the tests involved the database, one of the tests included Mock.
- Throughout the build, we regularly conducted user-based testing to help identify sporadic bugs which we then fixed.
- Once the application was integrated together, we systematically tested the whole application going through every combination of actions that a user could take to check for bugs that hadn't already been picked up.

System limitations

- Availability of area information. We would have liked to be able to take the areas offered down to more local regions, but this was not possible with the information available to us.
- Case rates are about 7 days old.
- As with any API based application, this app relies on the API continuing to be updated, maintained and available. This is one of the reasons we chose the API we did as it is an official UK government API.
- The population database is based on the most recent Office of National Statistics information from 2020. The database will therefore need updating each year as more recent statistics are made available.
- The advice given in the textfiles is likely to change with time and this will also need updating.
- Accessibility limitations
- Due to the short time we had to design and build, there have been limits on the way we have been able to explain risk and rates to the user. With more time, it would be possible to extend and build on this.
- Testing of the system ideally would need to include a mocked database.
- The log-in system has no functionality to change the password on the account, or other common functionalities such as 'get a hint'. This should be extended as the application is rolled out.

CONCLUSION

We have created an application which allows a user to find the up to date Covid-19 rates in their local area, a risk stratification and by extrapolation, their risk of catching COVID-19.

The user can interact with the application through an interface powered by Javascript. Then the Python backend retrieves the information on COVID-19 rates and population size from the government API and SQL database, respectively. Then, the rate is calculated and returned to the user with an option to view COVID-19 health recommendations for their nation and to save their search results.

Throughout this project our team worked together focussing on different aspects of the application and reviewing progress weekly through various communication channels. An agile approach was taken for flexibility and unit tests were conducted to ensure that the application ran smoothly.

In future, this application could be developed further in a number of different ways. For example:

- Cater for users globally by using different area data and languages
- Improve accessibility for visually impaired users.
- Extend its use into providing information about future viral pandemics.
- Allow different levels of access and advice to cater for different audiences - for example employers or event organisers making risk management decisions, different age groups or those in different types of employment.
- Provide more granular information about local areas.
- Provide more detailed information about hospital admission rates and death rates.
- Use the application to calculate and provide more detailed information - for example the actual risk of a user catching covid based on local rates and their personal risk profile.
- Increase the security of user accounts by using a hash function that allows the use of a salt and to prevent users from creating commonly guessed passwords
- Storing more detailed user and search data to allow for future epidemiological, public health and educational use.

Overall, the application functions as we intended, in a robust and reliable manner. All of the key-functionalities have been successfully delivered, and we have identified future areas for development as the application is rolled out on a larger scale. Despite working on different features, we were able to manage ourselves successfully as a team, and communicate any difficulties encountered promptly. This enabled the project to stay on track, and we were able to flexibly alter the project outcomes as circumstances changed.

APPENDIX

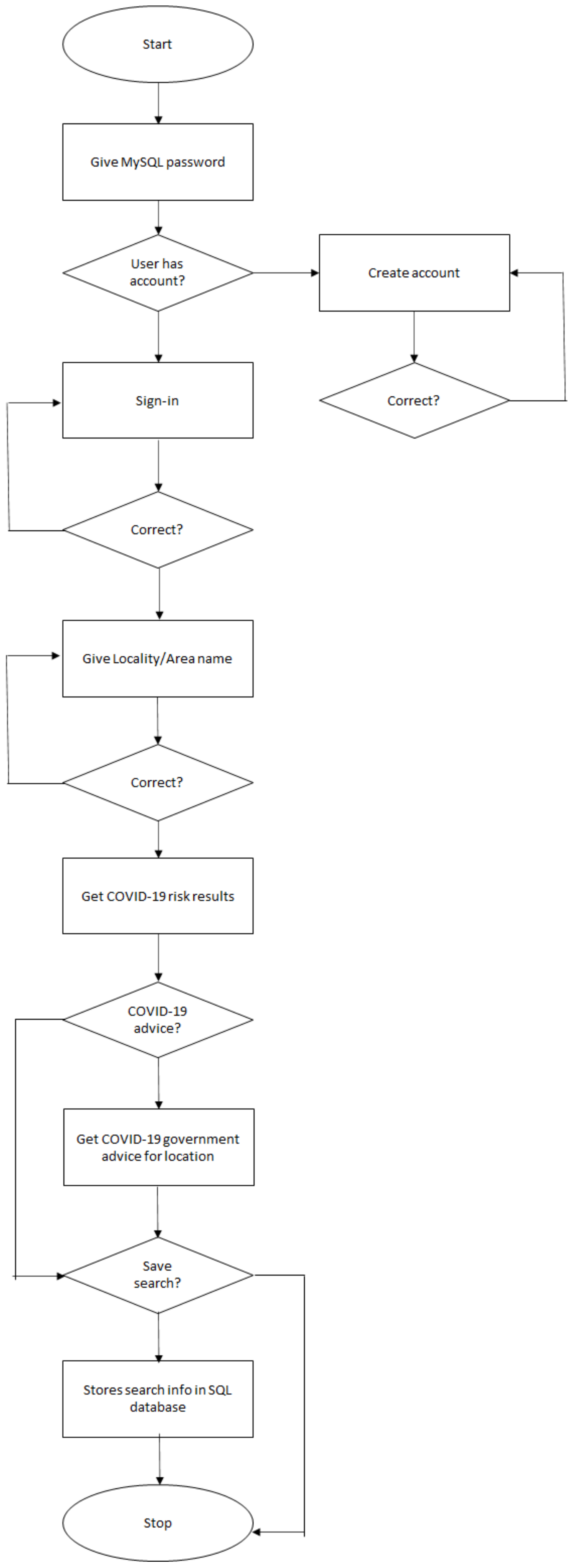

Start
Give MySQL password
User has account?
Create account
Correct?
Sign-in
Correct?
Give Locality/Area name
Correct?
Get COVID-19 risk results
COVID-19 advice?
Get COVID-19 government advice for location
Save search?
Stores search info in SQL database
Stop

Figure 1: Flow diagram of COVID-19 Calculator application.

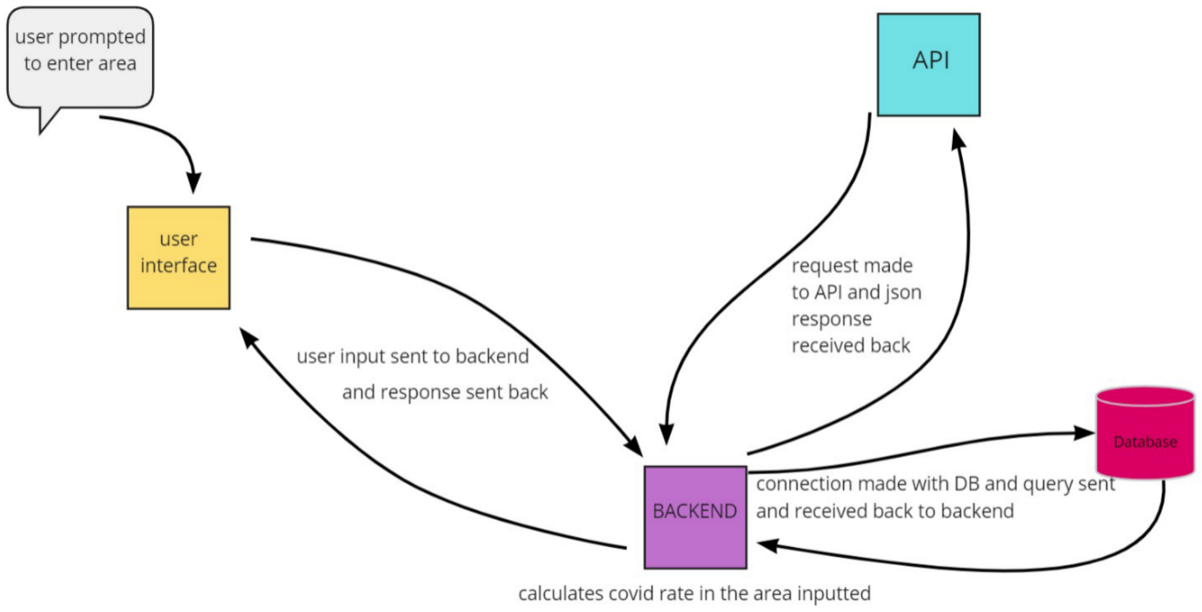

user prompted to enter area
user interface
user input sent to backend and response sent back
API
request made to API and json response received back
Database
connection made with DB and query sent and received back to backend
BACKEND
calculates covid rate in the area inputted

Figure 2: System architecture.

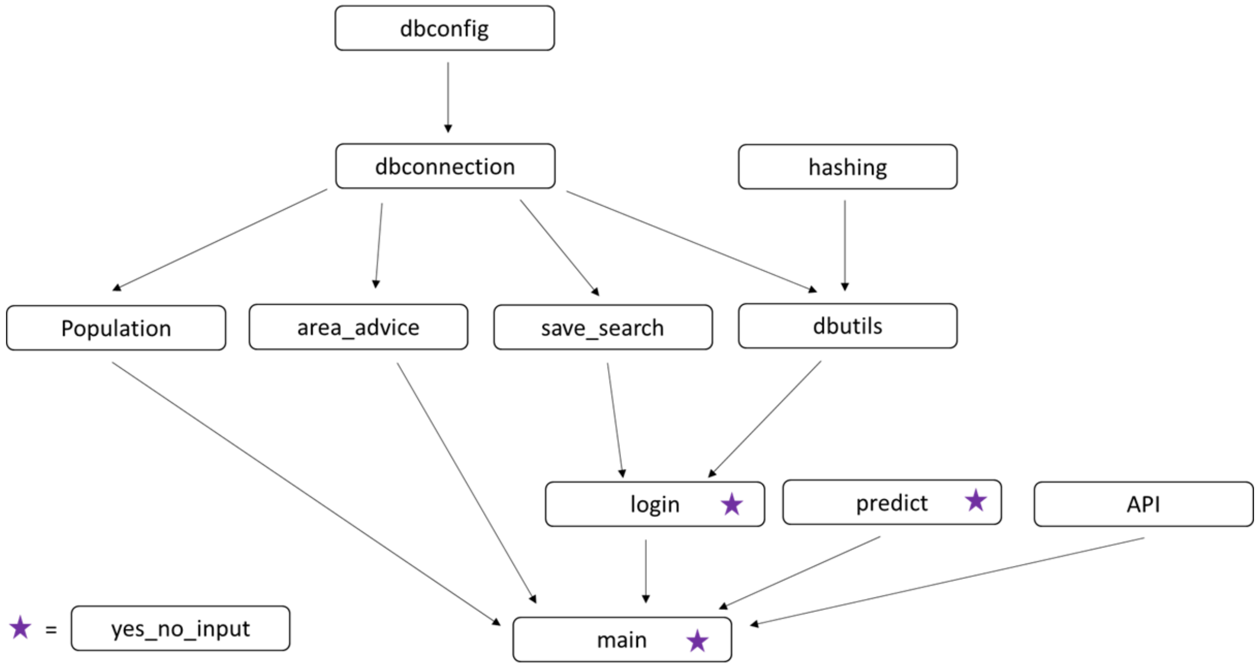


Figure 3: Python code architecture.

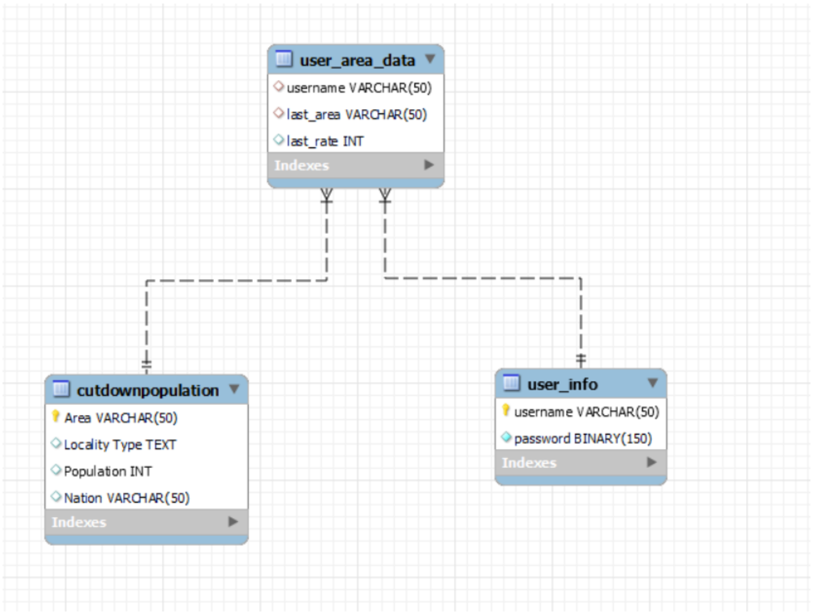
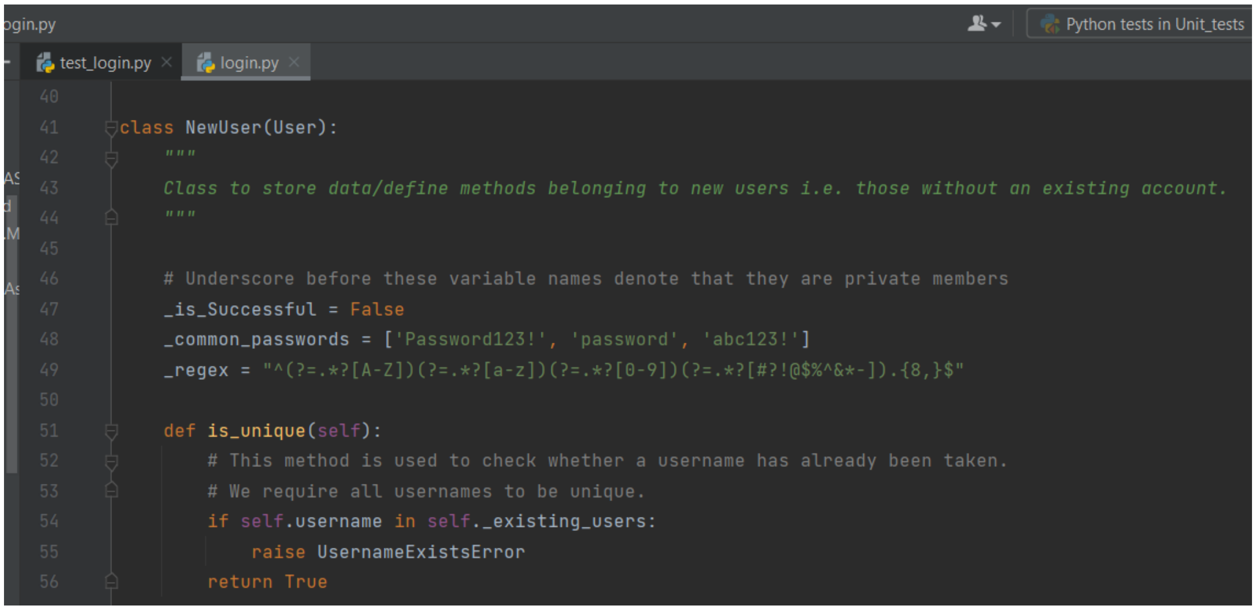


Figure 4: Entity Relationship Diagram for the Population database used in our application. The tables user_info and user_area_data are linked by the key 'username' which is the primary key of user_info and a foreign key in user_area_data.

A

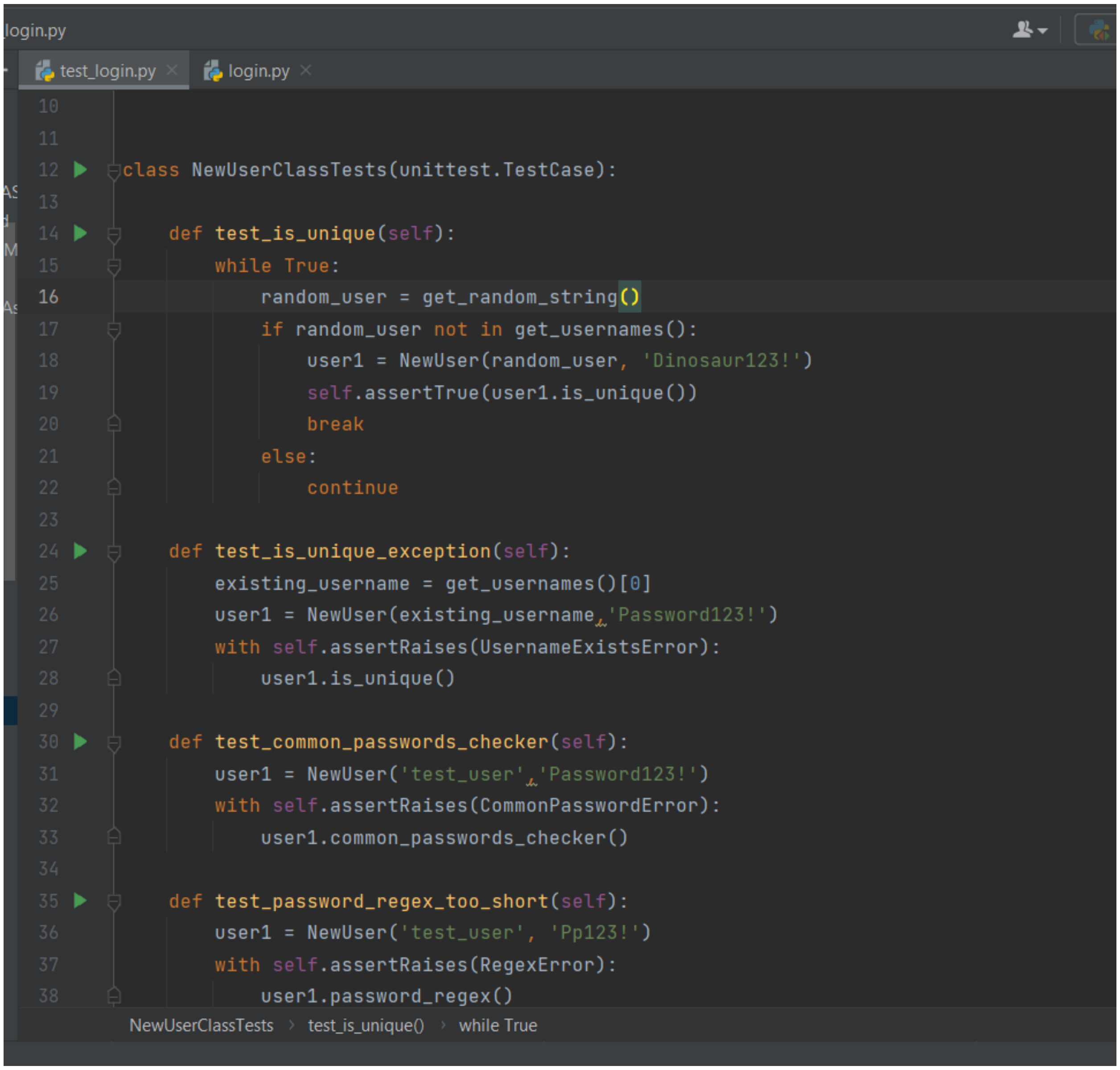


```python
class NewUser(User):
    """
    Class to store data/define methods belonging to new users i.e. those without an existing account.
    """

    # Underscore before these variable names denote that they are private members
    _is_Successful = False
    _common_passwords = ['Password123!', 'password', 'abc123!']
    _regex = "^(?=.*?[A-Z])(?=.*?[a-z])(?=.*?[0-9])(?=.*?[#?!@$%^&*-]).{8,}$"

    def is_unique(self):
        # This method is used to check whether a username has already been taken.
        # We require all usernames to be unique.
        if self.username in self._existing_users:
            raise UsernameExistsError
        return True
```

B



login.py

test_login.py login.py

```
class NewUserClassTests(unittest.TestCase):

    def test_is_unique(self):
        while True:
            random_user = get_random_string()
            if random_user not in get_usernames():
                user1 = NewUser(random_user, 'Dinosaur123!')
                self.assertTrue(user1.is_unique())
                break
            else:
                continue

    def test_is_unique_exception(self):
        existing_username = get_usernames()[0]
        user1 = NewUser(existing_username,'Password123!')
        with self.assertRaises(UsernameExistsError):
            user1.is_unique()

    def test_common_passwords_checker(self):
        user1 = NewUser('test_user','Password123!')
        with self.assertRaises(CommonPasswordError):
            user1.common_passwords_checker()

    def test_password_regex_too_short(self):
        user1 = NewUser('test_user', 'Pp123!')
        with self.assertRaises(RegexError):
            user1.password_regex()
```

Figure 5: An example of unit testing on a function from login.py. A) Functional code from login.py. B) Unit test from test_login.py.

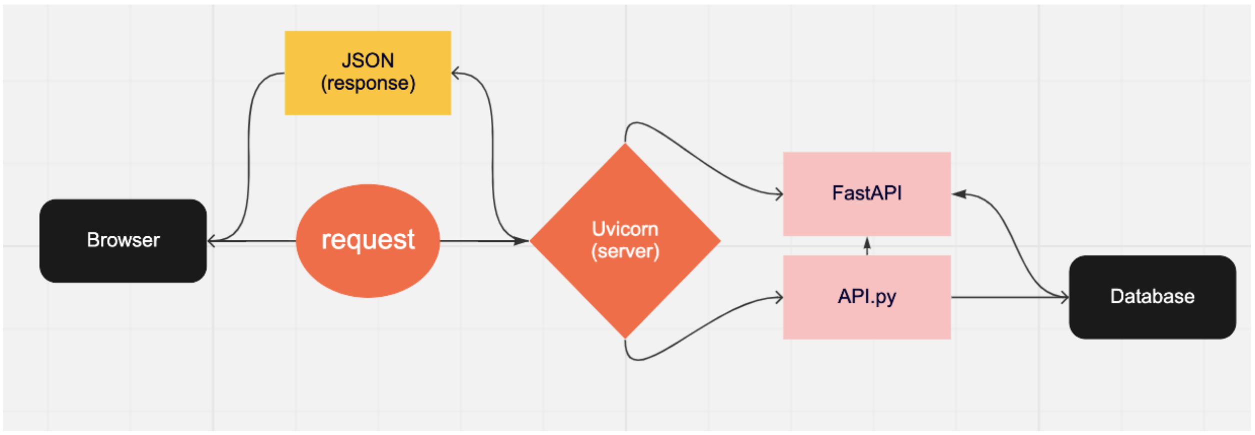
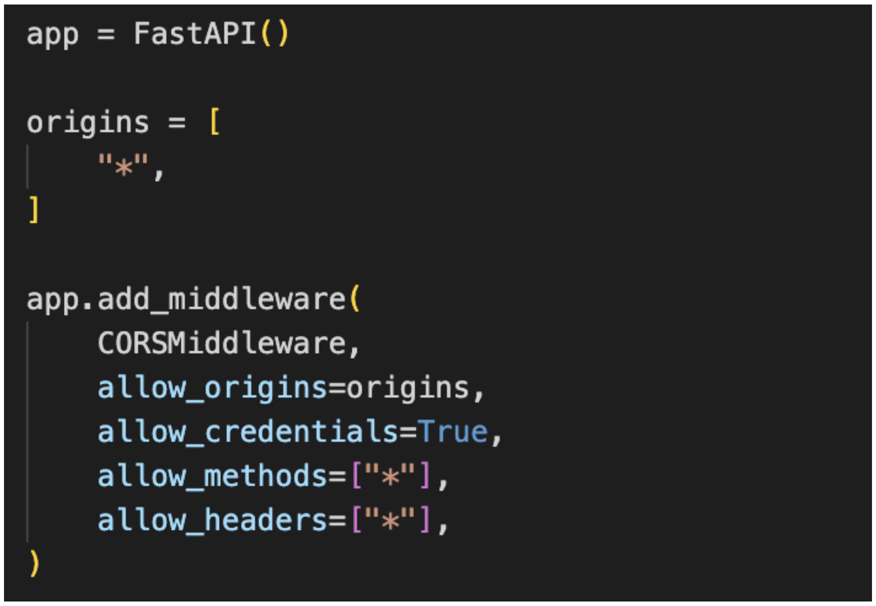


Figure 6: Flowchart demonstrating the web-application, requests and responses from server to API to database.

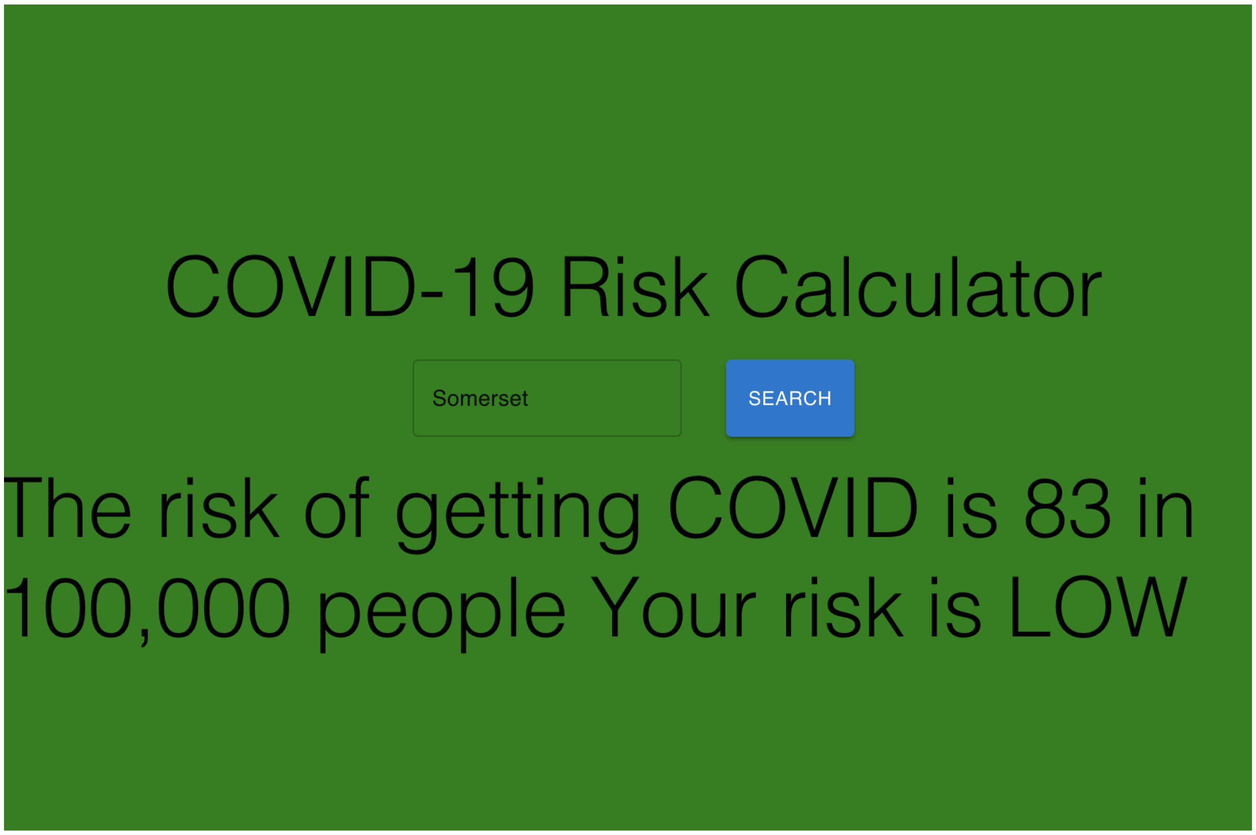


```
app = FastAPI()

origins = [
    "*",
]

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Figure 7: Code to allow Cross Origin Resource Sharing to overcome same origin policy limitations.


COVID-19 Risk Calculator

The risk of getting COVID is 83 in 100,000 people Your risk is LOW

Figure 8: Screenshot of web-application return risk rate and outcome when 'Somerset' is searched in location.