**Criterion C: Development[[1]](#footnote-2)**

The Dentist App is a web application requested by my client Mr. Dwijendra. The application uses Python and the Flask framework. It helps my client manage different textual and visual information of each of his patients which was previously done with Gmail and pen and paper.

**Dependencies**

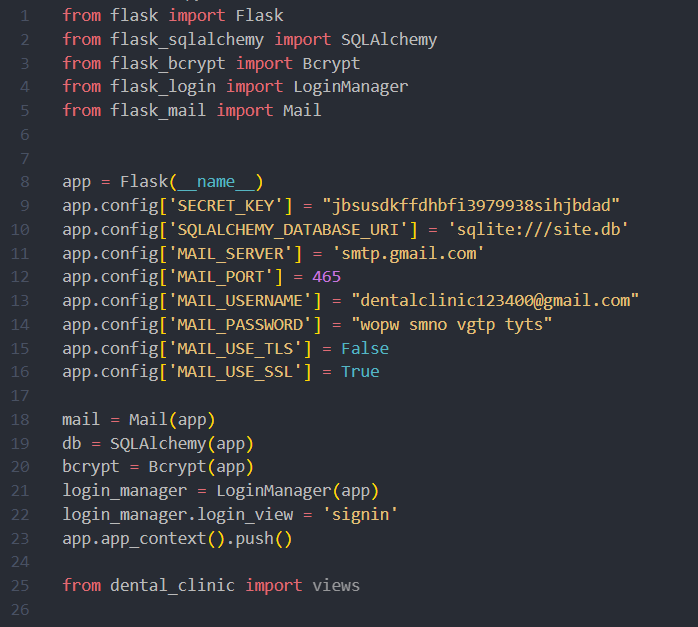
|  |  |
| --- | --- |
| Dependency | Purpose |
| Flask==3.0.2 | Framework to build the web application. |
| Flask\_Login==0.6.3 | Provides user session management and authentication. |
| Flask\_Bcrypt==1.0.1 | Hashing utilities for securely hashing passwords. |
| Flask\_Mail==0.9.1 | Email sending capabilities. |
| flask\_sqlalchemy==3.1.1 | Integrates SQLAlchemy, a Python ORM, for database operations. |
| flask\_wtf==1.2.1 | Provides form handling and validation capabilities. |
| Werkzeug==3.0.1 | A WSGI (Web Server Gateway Interface) utility library for Python. |
| WTForms==3.1.2 | A flexible forms library, utilized by Flask-WTF for form handling. |

**List of Techniques**

1. Flask Framework
2. WTF Forms
3. Server-Side Validation
4. Encryption using Bcrypt.
5. Multi-Factor Authentication
6. Flask-SQLAlchemy
7. Image Handling
8. Use of templating language
9. Use of multiple programming languages

**Flask Framework**

The code in the below snippet initializes a Flask web application along with some extensions like Flask-SQLAlchemy, which integrates SQLAlchemy, a powerful ORM library, Flask-Bcrypt adds bcrypt hashing utilities for password security, Flask-Login manages user sessions and authentication, and Flask-Mail facilitates email sending capabilities, configuring SMTP settings for email delivery.



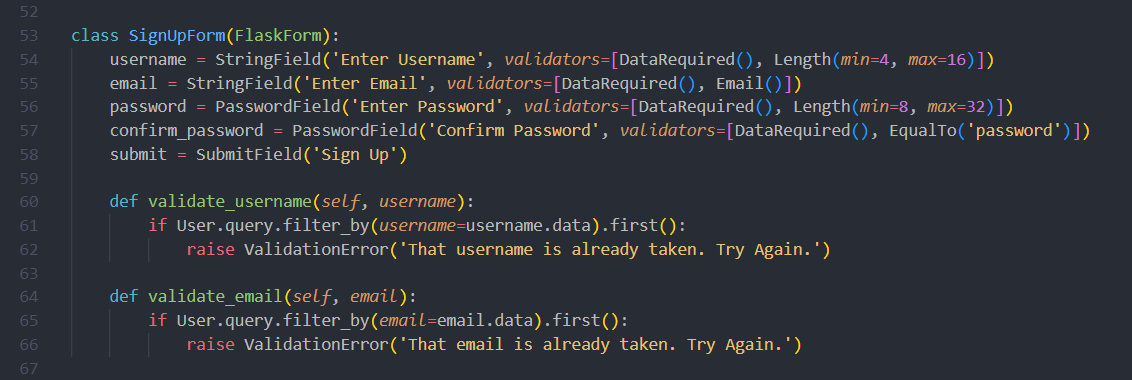
The application's configurations are set using the app.config dictionary, defining parameters such as the secret key for session security, database URI for SQLite, and SMTP server settings for email transmission. Finally, the code imports the views module from where the routes are defined.

**WTF Forms**

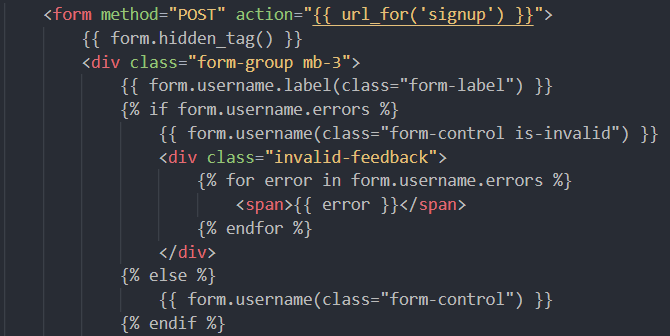
This code imports necessary modules and classes from Flask-WTF and WTForms libraries to facilitate form handling. Flask-WTF is an extension for Flask that integrates WTForms, a flexible form handling library for Python. The imported classes include FlaskForm, the base class for forms in Flask-WTF, and various field classes like StringField, PasswordField, FileField, SelectField, DateField, TimeField, and TextAreaField, which represent different types of form input fields. Validators such as DataRequired, Length, Email, EqualTo, NumberRange, etc., are also imported from wtforms.validators to enforce data validation rules on form fields.

A screen shot of a computer

Description automatically generated

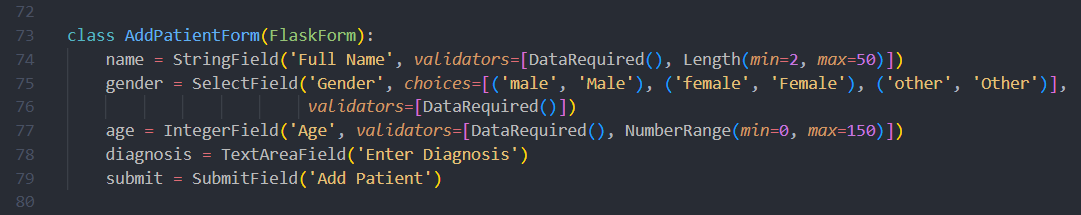


The above snippet is an example of how a form is implemented into the app. For instance, the code defines a SignUpForm class, inheriting from FlaskForm. It consists of fields for username, email, password, and confirm password, along with a submit button. An object of this class is instantiated and passed into the corresponding html file and is used like this:



The above code also specifies that the form data should be sent using the POST method to the URL associated with the signup endpoint. The form includes fields for username, etc. [[2]](#footnote-3), and if there are any validation errors for the username field, they will be displayed to the user. Otherwise, the form renders the username and other input fields normally.

**Server-Side Validation**



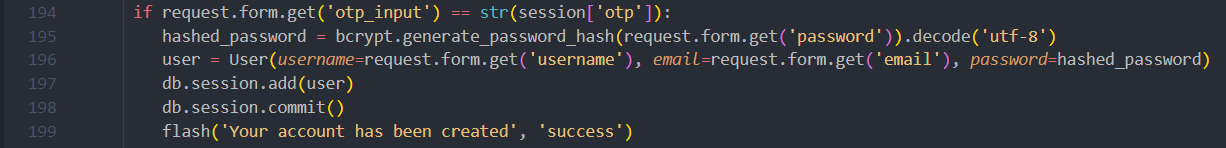
This code is an example of how server-side validation is carried out in the application. It defines an AddPatientForm class, inheriting from FlaskForm, intended for adding patient information to the database. The form includes fields for the patient’s full name, gender, age, and diagnosis, along with a submit button. Validators are applied to each field to enforce data requirements:

1. name: Validates that the field is not empty [DataRequired()] and enforces a length constraint between 2 and 50 characters [Length(min=2, max=50)].
2. gender: Requires that a gender option is selected from the provided choices [DataRequired()].
3. age: Ensures the age provided is a non-negative integer within the range of 0 to 150 [DataRequired(), NumberRange(min=0, max=150)].

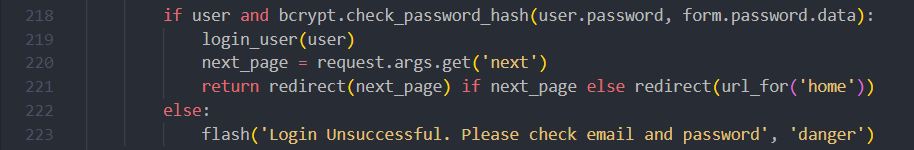
**Encryption using Bcrypt**

In the provided code snippets, bcrypt is used for hashing passwords in the sign-up route and for verifying passwords during sign-in. Hashing is a cryptographic technique that transforms input data (in this case, passwords) into a fixed-size string of characters.

Sign-up route: When a user signs up, their password is hashed using bcrypt’s generate\_password\_hash function before being stored in the database. This ensures that the original password is not stored directly, enhancing security.



Sign-in Route: When a user attempts to sign in, bcrypt’s check\_password\_hash function is used to compare the hash of the provided password with the hash stored in the database. If the hashes match, it means the provided password is correct, and the user is authenticated



**Multi-Factor Authentication**

The code snippets below implement a basic form of multi-factor authentication (MFA) using a one-time password (OTP) sent via email during the sign-up process:

**Sign-up route [/signup]**:

1. When a user attempts to sign up, The SignupForm is displayed.
2. Upon submitting the form an OTP is generated using generate\_otp() function and stored in the session.
3. The OTP is then sent to the user's email address using send\_otp\_email() function.
4. A message is flashed to the user instructing them to check their email for the OTP.

A computer screen shot of a program code

Description automatically generated

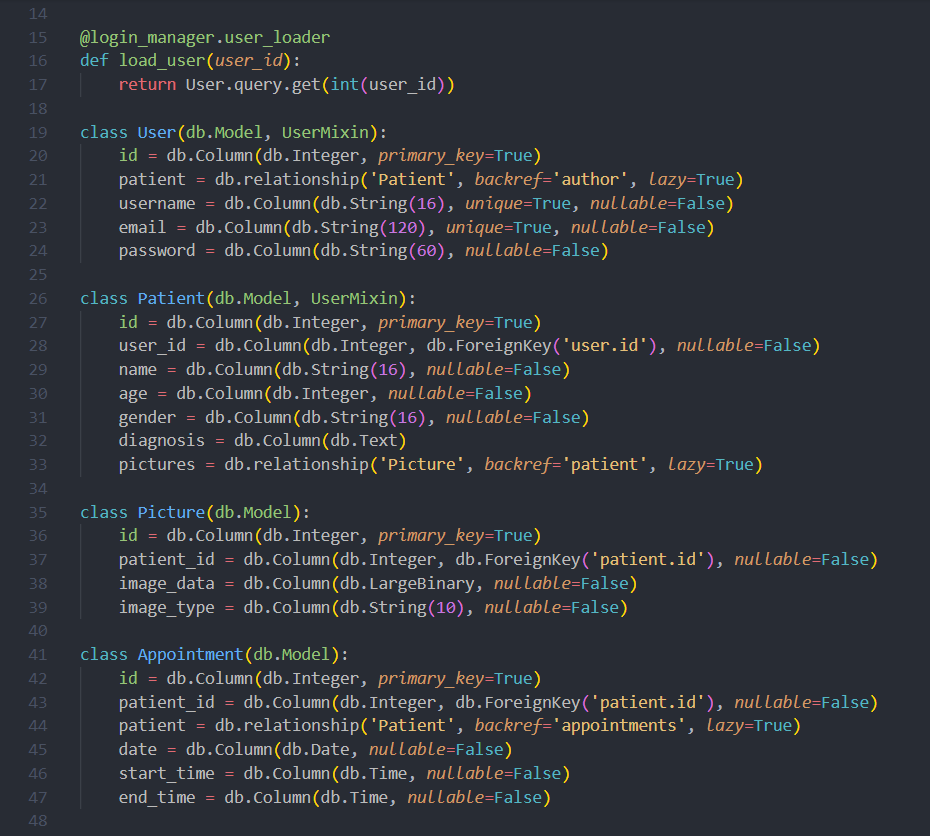
**OTP verification route [/signup/otp]**:

1. After receiving the OTP from the user, it is compared with the OTP stored in the session.
2. If the OTP matches, the user’s password is hashed using bcrypt, and the user is added to the database.
3. A success message is flashed, indicating that the account has been created, and the user is redirected to the sign-in page.
4. If the OTP does not match, an error message is flashed, instructing the user to try again.



**Flask-SQLAlchemy**

The code snippet below defines SQLAlchemy models within a Flask application, facilitating object-relational mapping (ORM) for data management. Models such as User, Patient, Picture, and Appointment are defined with attributes and relationships, encapsulating database tables and their interactions. Flask-SQLAlchemy automates session handling and abstracts away SQL queries, enabling developers to work with Python objects for database operations. This approach enhances modularity and scalability, simplifying data retrieval and manipulation while maintaining data integrity. By leveraging SQLAlchemy's ORM capabilities, the application achieves improved development efficiency and maintainability.



Example Usage:

A screen shot of a computer code

Description automatically generated

It simplifies database operations in Flask applications by providing an ORM layer, allowing developers to interact with databases using Python objects. For example, Patient.query.all() retrieves all the patient records from the Patient table returned in an array. This abstraction reduces the need for writing raw SQL queries, enhancing code readability and maintainability while improving security through parameterized queries.

**Image Handling**



These routes handle operations related to patient pictures within the application:

**View Pictures Route [/patients/<int:patient\_id>/pictures/view]**:

1. Retrieves the patient object based on the provided patient\_id.
2. Renders a template to display the pictures associated with the patient.

**Add Picture Route [/patients/pictures/add]**:

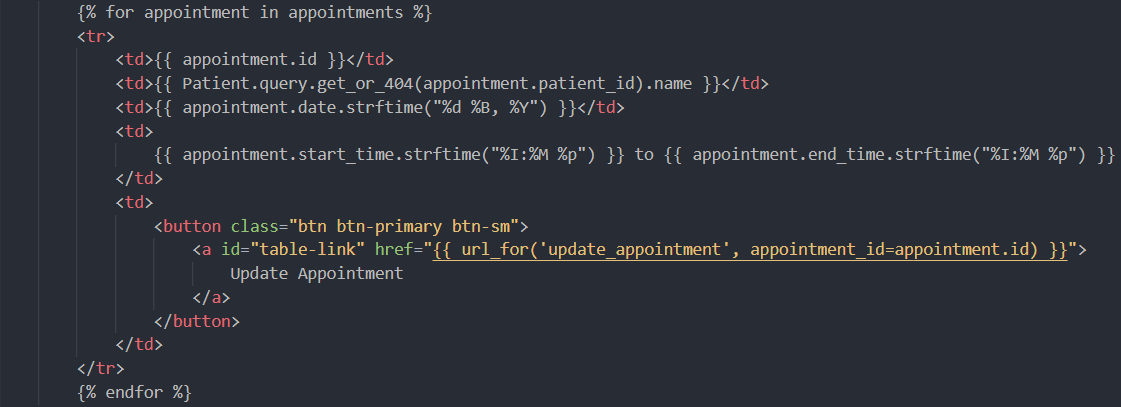
1. Renders a form for adding a new picture to a patient's record.
2. Upon form submission, process the uploaded picture file.
3. Creates a new Picture object with the uploaded image data and patient ID.
4. Adds the new picture to the database and redirects to the view pictures route for the corresponding patient.

**Retrieve Picture Route [/pictures/<int:picture\_id>]**:

1. Retrieves the picture object based on the provided picture\_id.
2. Sends the image data as a file in the appropriate format (e.g., jpeg, png) to the client for viewing.

These routes demonstrate image-related operations, including uploading and viewing pictures associated with patient records. SQLAlchemy is utilized for database operations, while Flask’s file handling capabilities are leveraged for image uploads and retrieval.

**Jinga2 Templating Language**



Jinja2 templating is a powerful and flexible templating engine that was used in the application. It allowed embedding Python code directly into HTML templates, enabling dynamic content generation. Jinja2 provided features such as template inheritance, macros, filters, and loops, making it easy to create reusable and maintainable templates. By separating presentation logic from application logic, Jinja2 enhanced code organization and readability. Additionally, Jinja2 templates support context variables, allowing data to be passed from Flask routes to HTML files.

**Use of multiple programming languages**

JavaScript was used in the front-end to carry out tasks such as remembering navbar position across page-reloads making the application more user-friendly and easier to work with.



**Local Storage for Submenu State**:

1. JavaScript is used to manage the state of submenu visibility using local storage.
2. When a submenu toggle is clicked [submenuToggles.forEach], the corresponding submenu's visibility state is toggled [submenu.classList.toggle('show')].
3. The submenu's visibility state ('show' or 'hide') is stored in the browser's local storage.
4. When the page is loaded [DOMContentLoaded event], JavaScript retrieves the submenu's visibility state from local storage [localStorage.getItem], and if the submenu was previously shown, it remains visible.

Integration of multiple programming languages into web development allows for enhanced functionality and interactivity. HTML provides the structure of the webpage, CSS styles the elements, Python (through Flask) handles server-side logic and dynamic content generation, and JavaScript adds client-side interactivity and behavior. This approach enabled leveraging the strengths of each language to create dynamic features with improved user experience and functionality.

Word Count: 1304

**References**

“Welcome to Flask¶.” *Welcome to Flask - Flask Documentation (3.0.x)*, flask.palletsprojects.com/en/3.0.x/. Accessed 28 Mar. 2024.

“WTF¶.” *Flask*, flask-wtf.readthedocs.io/en/1.2.x/. Accessed 28 Mar. 2024.

“Flask-Mail¶.” *Flask-Mail - Flask-Mail 0.9.1 Documentation*, pythonhosted.org/Flask-Mail/. Accessed 28 Mar. 2024.

1. See Appendix C [↑](#footnote-ref-2)
2. See pages 21 to 24 (Appendix C) [↑](#footnote-ref-3)