Internship Project Report: *BiteMyRoll*

# This report details the design, development, and implementation of the internship project titled “BiteMyRoll”, a food ordering web application offering a variety of rolls and beverages. The project was inspired by analyzing the design structure and operational flow of reputed restaurant and hotel websites that offer food ordering and room service. Based on this analysis, a theme-driven, user-friendly, and modular application was created, focusing on simplicity, functionality, and future scalability.

# 2. Project Inspiration and Background

The idea for BiteMyRoll was born out of extensive research into the UI/UX and architecture of well-known restaurant and hotel websites. The goal was to understand how such systems allow users to browse menus, customize items, and place orders seamlessly. Drawing inspiration from these professional platforms, the project focused on delivering a simplified version tailored to serve as a standalone food ordering system that can later be scaled or integrated into larger ecosystems.

# 3. Design and Frontend Structure

The project utilizes HTML templates rendered using Flask's Jinja2 templating engine. Pages include the home, menu, contact, about, drink menus, and detail views for each product. Static images for each roll and drink enhance the visual appeal and improve user interaction. The frontend emphasizes user experience with accessible navigation, smooth transitions, and form-based interactions.

# 4. Backend Architecture and Flask Routing

Flask is used as the web framework. Routes are defined for major functionalities such as '/menu', '/drinks', '/detail/<type>', '/cart', '/orders', etc. Each route corresponds to a specific operation and renders templates with appropriate dynamic data from the server.

# 5. Form Handling and Validation

WTForms is used for form handling. Separate forms are created for login, signup, password change, and order creation. Form classes are defined in `forms.py` and leverage validators for secure data entry.

# 6. Data Handling Without a Database

Currently, the app uses a Python dictionary (`database`) to simulate a backend database. Each user entry stores login info, cart contents, orders, and location. This structure is designed to be compatible with future database systems like SQLite or MySQL or PostgreSQL with minimal code refactoring.

# 7. Session and Authentication

Session management is handled using Flask's session object. Upon login/signup, user data is stored in session variables to personalize and secure user access throughout the app.

# 8. Cart and Order Management

Each item (roll or drink) can be added in different sizes. Items are stored per user session inside the dictionary. The '/cart' and '/checkout' routes manage the addition, update, and placement of orders, with order history maintained in the `orders` list per user.

# 9. Location and Delivery Handling

The app includes a location setting feature using GPS integration. This is stored in the `location` key of each user's entry in the dictionary and is designed to be expandable with real maps or delivery APIs.

# 10. Routing Summary

• '/' and '/home': Home Page

• '/menu': Roll menu

• '/drinks': Drinks menu

• '/detail/<type>': Individual roll detail page with quantity selection

• '/drinksDetail/<type>': Similar page for beverages

• '/cart', '/orders', '/checkout': Order processing

• '/login', '/signup', '/logout': User session handling

• '/set\_location': Stores delivery location via JSON

# 11. Readiness for Database Integration

The current dictionary schema uses keys like 'info', 'cart', 'orders', and 'location', mimicking relational tables. This allows for seamless migration to MySQL/SQLite. Only the data access layer would need refactoring using SQLAlchemy or raw queries.

# 12. Challenges Faced

Key challenges included simulating persistent storage with dictionaries, managing form validations for multiple roll sizes, and dynamically rendering product details. Also, handling user sessions and redirection after login was carefully managed.

# 13. Future Scope

• Database integration

• Admin panel for inventory and order management

• Payment gateway

• A more advanced and mobile-responsive Progressive Web App (PWA), improving upon the current implementation.

• Real-time delivery tracking

# 14. Data Flow and Code Walkthrough

The application handles data internally using a global Python dictionary named database. This design simulates the structure of a relational database and is built in a way that can be migrated easily to systems like MySQL or SQLite.

1. Simulated Data Structure

The following dictionary simulates a database entry for a user:

database[user\_email] = {

'info': user\_info,

'cart': {},

'orders': [],

'visited\_drinks': False,

'location': ''

}

* info: Contains user credentials and personal data.
* cart: Holds all items the user adds to their cart.
* orders: List of all completed purchases.
* visited\_drinks: Tracks whether the drinks menu was accessed (used for dynamic UI).
* location: Stores the delivery location set by the user.

1. Adding Items to Cart

When the user submits a form from the roll detail page (/detail/<type>), the following code stores the selected items:

database[session.get("User\_email")]['cart'][type.strip()] = {

'name': roll['name'],

'size': qty,

'note': form.note.data

}

* This structure allows users to order different sizes with custom notes.

1. Checkout and Order Placement

During checkout, all items in the cart are copied into a new order entry, timestamped, and stored:

order\_data = {

'items': database[session.get("User\_email")]['cart'].copy(),

'total': total,

'timestamp': datetime.now(),

}

database[session.get("User\_email")]['orders'].append(order\_data)

database[session.get("User\_email")]['cart'].clear()

* This ensures the user’s order history is retained and the cart is emptied after placement.

1. Accessing Orders

To show past orders on the /orders route, the app retrieves the list like this:

user\_orders = database.get(email, {}).get('orders', [])

* This provides a safe way to access user-specific data without throwing errors if missing.

# 15. ROUTE BREAKDOWN

1. /menu and /drinks

These routes render the **roll** and **drink menus** respectively:

@app.route('/menu')

@app.route('/drinks')

* **Purpose**: Show available food and drink options.
* **Data passed**: ROLLS\_DATA, BEVERAGE\_DATA, and database for user context.
* **No data modification occurs** here — just displays products.

1. /detail/<type> and /drinksDetail/<type>

These handle adding items to the cart for a specific **roll** or **drink**.

@app.route('/detail/<type>', methods=["GET", "POST"])

@app.route('/drinksDetail/<type>', methods=["GET", "POST"])

* **Conditions:**
* If no size quantity is selected (<= 0), flash a warning.
* If user is not logged in, redirect to /login with next=request.path.
* If logged in, update database[user]["cart"] with selected item details.

**Example Snippet**:

if qty\_regular <= 0 and qty\_large <= 0 and qty\_mega <= 0:

flash("Please select at least one item to add to cart.", "warning")

* + **Data structure updated**:

database[user\_email]["cart"][type] = {

"name": roll["name"],

"size": qty,

"note": form.note.data

}

1. /cart

Displays the contents of the user’s cart.

@app.route('/cart')

* **Reads** data from database[session.get("User\_email")]["cart"].
* **Used for:** Viewing, updating, or checking out items.
* **Data flow**: Read-only, unless updatecart or checkout routes are used.

1. /updatecart

Handles **incrementing or decrementing** item quantity.

@app.route('/updatecart', methods=["POST"])

* **Query parameters**: roll\_key, size, op (inc/dec)
* **Conditions handled**:
  + If roll or size doesn't exist: flash error.
  + If operation is inc, ensure qty ≤ 10.
  + If dec results in qty = 0, remove that size.
  + If no sizes remain, remove the entire item.

**Code Example**:

if op == "inc" and current\_qty < 10:

database[user]["cart"][roll\_key]["size"][size] += 1

elif op == "dec":

if current\_qty > 1:

...

else:

del database[user]["cart"][roll\_key]["size"][size]

1. /checkout

Places an order using the current cart.

@app.route('/checkout', methods=["POST"])

* **Conditions**:
  + If not logged in → redirect to /login
  + If cart is empty → redirect back to /cart
* **Actions**:
  + Copy cart to a new order entry in orders
  + Clear the cart
  + Flash success message

**Order Structure**:

order\_data = {

'items': cart.copy(),

'total': total,

'timestamp': datetime.now(),

}

database[user\_email]["orders"].append(order\_data)

database[user\_email]["cart"].clear()

1. /orders

Displays past orders.

@app.route('/orders')

* **Read-only**
* Requires login
* Renders all entries in database[user]["orders"]

1. /cancel\_order/<int:index>

Cancels an order based on list index.

@app.route('/cancel\_order/<int:index>', methods=["POST"])

* **Condition**:
  + If index is valid, remove the order.
  + Else, flash an error.

1. signup and /login

Handles authentication and initializes user data:

@app.route("/signup", methods=["GET", "POST"])

@app.route("/login", methods=["GET", "POST"])

* **Signup**:
  + Validates form.
  + Rejects duplicate emails.
  + Initializes a new dictionary entry.

1. /changePassword

Allows users to change their password securely:

@app.route('/changePassword', methods=["GET", "POST"])

* Checks that current password matches session-stored one.
* Updates session and flashes success or error.

1. /logout

Clears the session (only email is cleared):

@app.route('/logout')

* Data remains in memory (database not flushed).
* User sees default view (logged-out state).

1. /set\_location

Saves user delivery location (e.g. GPS or text-based):

@app.route('/set\_location', methods=["POST"])

* Accepts JSON data
* Stores it under:

database[user\_email]["location"] = location\_data

* Returns JSON success message

# 16. Conclusion

BiteMyRoll represents a practical solution that reflects real-world food ordering systems. It demonstrates skills in Python, Flask, session handling, form processing, and scalable code structuring. This project has high potential for real deployment with minimal modifications.

# 17. Appendix and Screenshots