Swinburne University of Technology

School of Software and Electrical Engineering

ASSIGNMENT AND PROJECT COVER SHEET

Subject Code: SWE30003 Unit Title: Software Architectures and Design

Assignment number and title: 3 , Design Implementation Due date: 11:59pm, 24th Nov 2024

Project Group: 5

Tutorial Day and time: 13:00 Tuesday

Tutor: Mr. Trung Pham

To be completed as this is a group assignment

We declare that this is a group assignment and that no part of this submission has been copied from any other student's work or from any other source except where due acknowledgment is made explicitly in the text, nor has any part been written for us by another person.

ID Number	Name	Signature
104513735	Thai Duong Bao Tan	Tan
1041171133	Le Huu Nhan	Nhan
103805253	Dang Quynh Chi	Chi

Marker's comments:

Total Mark:

Extension certification:

This assignment has been given an extension and is now due on

Signature of Convener:

Introduction

The report focuses on the detailed design and implementation of a Restaurant Information System tailored for Relaxing Koala, building upon the initial object-oriented design established in Assignment 2. This assignment aims to refine the existing framework, address any shortcomings, and document the evolution of the design process through practical implementation experiences.

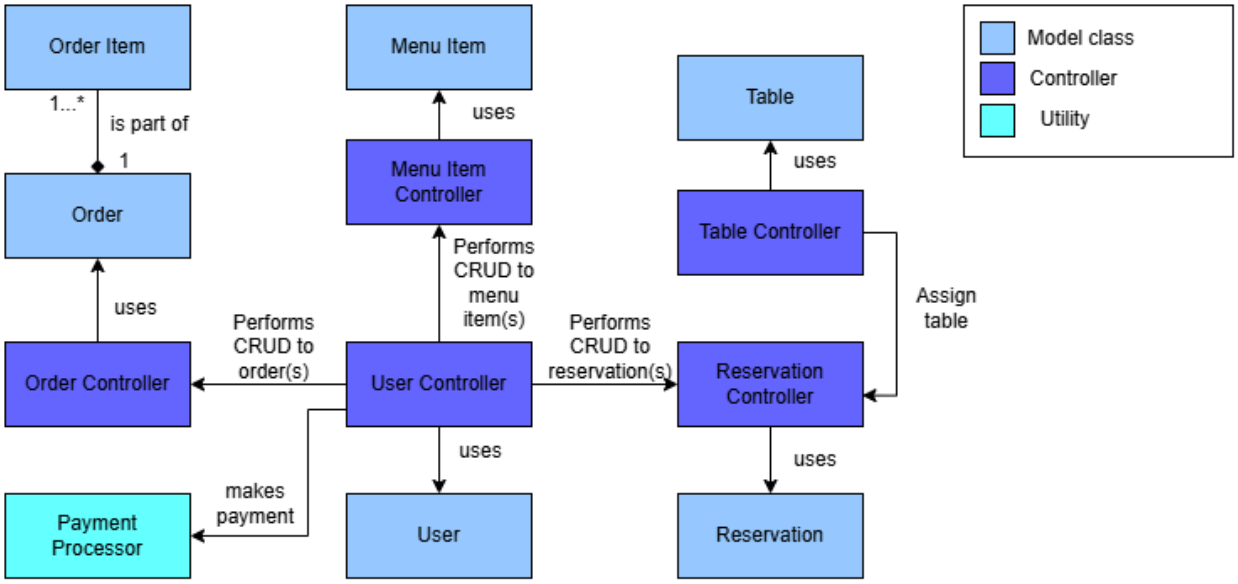
In the initial phase of this project, we established a high-level design that outlined the core functionalities required for the system. This included managing orders, reservations, and menu items, which are essential for the restaurant's operations. However, as we progressed into the detailed design and implementation stages, it became evident that certain aspects of our initial design were either inadequately addressed or entirely missing. Therefore, this report will not only present the refined design but also reflect critically on the original concepts to highlight areas of improvement.

The detailed design process involved a thorough analysis of each component's responsibilities within the system architecture. By adhering to object-oriented principles, we aimed to achieve a clean separation of concerns among models, views, and controllers. This approach is crucial for ensuring that our system remains maintainable and scalable as new features are added in the future. Additionally, we recognized that some classes from our initial design were redundant or unnecessarily complex and made adjustments accordingly to streamline functionality.

Throughout this report, we will document the justifications for changes made during the design process and discuss how these modifications enhance the overall system architecture. Furthermore, we will reflect on our experiences with the initial design to identify lessons learned that will inform our approach to future projects. By analyzing both successful elements and shortcomings of our original framework, we aim to provide insights that can improve future design methodologies in similar contexts.

Detail Design

Final Class Diagram



Justification of Changes and Non-changes

Removal

Several classes from the original design were removed after careful consideration, as they were deemed unnecessary or redundant:

1. **Invoice Class**: Initially, we included an Invoice class to handle invoice-related tasks. However, after more analysis, we noticed that the Order class could handle all the responsibilities initially assigned to the Invoice class, such as summarizing order details and payment information. As a result, the Invoice class provided no significant benefit and was removed to simplify the design and reduce redundancy.
2. **Data Tracker Class**: The original design included a Data Tracker class for managing reports and statistics. After reconsideration, we determined that this functionality would be better integrated into a dedicated reporting or analytics interface within the web application. The actual business need—viewing statistics—could be met more effectively with a specialized webpage or dashboard than with a standalone class.
3. **Delivery Class and Its Business Logic Handler**: Although arranging delivery is one of the system requirements and was simplified in our initial design, we found

that implementing this functionality remained complex. Managing delivery logistics, such as assigning drivers, tracking deliveries, and ensuring timely updates, required significant effort and additional considerations. After careful evaluation, we decided that this functionality could be handled more efficiently by integrating with an existing third-party delivery system. These systems are specifically designed to manage deliveries and provide robust features that would be difficult to replicate within our project scope. By outsourcing delivery management to a third-party service, we simplified our design and focused on core functionalities, ensuring that the system is manageable while still meeting the restaurant's requirements.

Non-Changes

Despite these modifications, the majority of the candidate classes from the original design remained unchanged. We found that most of these classes accurately reflected the core business operations of the Relaxing Koala. For example, classes such as Order, Reservation, and Menu Item remained central to the system design, as they were directly related to key restaurant processes. Their roles were clear and appropriate for the system's functionality, so no major changes were required.

Responsibilities and Collaborators

One of the most significant changes we made during the detailed design and implementation stages was the reallocation of responsibilities for the **User** model. In the original design, the User class was responsible for placing orders, making reservations, and processing payments. Upon further review, we realized that these responsibilities did not correspond to the primary role of a model in the MVC architecture, which is to store and manage data (Rahman 2023). Instead, we restructured the design so that these tasks would be handled by controllers or other dedicated classes. This change was crucial in adhering to the principle that models should only focus on managing the data they represent, which helps maintain a clear separation of concerns and makes the system easier to maintain and extend.

We also refined the responsibilities of the remaining model classes. In the original design, some model classes were responsible for tasks other than data management, such as executing business logic or interacting with other components. For example, the Menu Item class was initially in charge of updating pricing and availability. We revised this to limit the Menu Item's responsibilities to data storage only, while creating separate components to handle tasks such as price updates or availability changes. This change improved cohesion and reduced the possibility of coupling between unrelated parts of the system.

Dynamic Aspects

Bootstrap Process

The new bootstrap process for the Menu Initialization remains largely unchanged. The Menu.cshtml.cs class fetches menu data via the MenuItems context, and the menu is displayed in a grid format for users. While the assignment 2 design specifies that the menu can only be modified by admins, this behavior is indirectly supported in the new process through the admin interface (MenuItemManager in Admin.cshtml). No major structural changes are observed in how the menu is initialized or displayed to users.

The User Initialization process introduces several enhancements. As in the original process, user credentials are validated against the database, and a User class instance is created upon successful login or registration. However, the new implementation adds stricter validation for registration, including a ConfirmPassword field and checks for unique email addresses. The User class has been extended to include properties like Role and PhoneNumber, with default roles assigned as Customer. Cookies are also used to store user session details, enabling role-based access and automatic login functionality. These changes improve the security and usability of the user authentication process.

For the Order Manager, the new process retains the core functionality of managing the order lifecycle, including creation, editing, and cancellation of orders. Users can select order types (dinein or takeaway), specify payment methods, and view their order history if logged in. The Order.cshtml.cs class handles order data validation and storage, linking Order instances to OrderItem instances. Enhanced server-side validations ensure invalid or incomplete data is flagged, streamlining the order processing experience. However, there are no significant changes to the basic flow of order management from the original process.

The Reservation Manager also follows the original process closely, allowing users to create reservations and assign tables. In the new implementation, Reservation.cshtml.cs enforces additional checks, such as ensuring the reservation date and time are not in the past. Logged-in users can view their reservation history, with pending reservations assigned a status of "Pending." These validations and added features enhance the reliability and user experience of reservation handling while staying true to the original design.

Two components from assignment 2 report, the Delivery Manager and the Payment Processor, are now not implemented fully in assignment 3. Delivery management, including takeaway orders and customer notifications, is removed completely to simplify

the implementation process. Payment processing is only partially implemented in the Order.cshtml.cs, with total calculations and item subtotals handled directly in the order creation process, but there is no invoice generation or payment status updates.

Interactive Scenarios

The reservation process in the new interaction maintains the core functionality described in the original design. The ReservationModel class in Reservation.cshtml.cs enables users to select a party size, date, and time, validating that the chosen date and time are not in the past. Unlike the original design, the new implementation explicitly uses cookies to authenticate users and retrieve reservation histories.

The menu interaction in the new system adheres closely to the assignment 2 design's process. When users access the Menu page, the MenuModel class fetches menu data from the database via the OnGetAsync() method, displaying the items in a grid format with availability indicators. Users can view the menu regardless of authentication status, aligning with the original report's design. However, the current implementation adds an interactive interface where unavailable items are disabled, providing better user feedback. The database query and item rendering processes remain unchanged.

For ordering takeaway food, the new interaction diverges slightly from assignment 2. While the original process describes integration with a Delivery Manager class to handle delivery orders, our current project implementation does not include a dedicated delivery mechanism. Instead, the OrderModel class manages order creation and item selection, allowing users to specify "takeaway" as an order type.

The dine-in payment process simplifies some aspects of the original interaction. The Order.cshtml.cs class enables customers to select a payment method (cash or card) during the ordering process. However, there is no dedicated Payment Processor class for handling transactions or generating invoices, as described in assignment 2. Instead, payment handling is rudimentary, with hardcoded card details and a lack of integration with a payment verification system. This simplification reduces complexity but does not fully meet the detailed functionality described in the original interaction.

Discussion of Assignment 2 Design

Good Aspects

1. **Detailed Problem Analysis**: Our team conducted a thorough analysis of the problem, making a strong effort to identify and address potential challenges

faced by The Relaxing Koala restaurant. This effort resulted in nearly 20 clearly defined assumptions, each carefully crafted to reflect the restaurant's operational issues. In addition, we included a "Simplifications" section in our design to reduce complexity and make the design process easier to manage. This proactive approach not only provided us with a better understanding of the problem, but it also ensured that the implementation phase proceeded smoothly, with few unexpected obstacles. By addressing potential issues early on, we established a solid foundation for the entire project.

2. **Candidate Classes, UML Diagrams, and CRC Cards**: The inclusion of candidate classes, UML diagrams, and CRC cards was extremely useful in helping us understand the relationships between classes and their respective responsibilities. These tools gave us a clear picture of how different components of the system interacted and worked together. While there were some cases of over-inclusion or omission in the design, these items greatly aided the implementation process. They provided a structured framework that helped us reduce ambiguity and better align the design with the system requirements. This clarity ultimately improved our team's efficiency and confidence during the implementation phase.
3. **Consideration of Design Patterns**: Our team recognized the importance of incorporating design patterns into the system. While many of the design patterns we explored proved to be unnecessary for the project, the process of considering them demonstrated a thoughtful and deliberate approach to design. In the end, we primarily used the MVC (Model-View-Controller) design pattern, which provided a well-organized structure for the system. This decision ensured that our design was maintainable and scalable, even if new features were added in the future.
4. **Processes Verification**: The verification processes we included in our design were critical in guiding the implementation phase. By outlining how the system's functions should work and how users would interact with them, the verification sections provided a clear roadmap for development. These detailed verifications enabled us to anticipate and address potential edge cases and usability concerns, ensuring that the system worked as intended. Furthermore, they helped our team visualize the application's workflow, bridging the gap between theoretical design and practical implementation.

Missing from Original Design

1. **View Component**: The "View" component in the MVC architecture was not taken into account in our original design, which was a significant oversight. While we worked hard to design the backend system, focusing primarily on the "Model" and "Controller" (we called it “Manager” at that time) aspects, we neglected the

"View," which is essential for displaying the interface to users. At the time, we assumed that the frontend did not require specific design considerations and failed to define it as a separate class or component. While this omission was not particularly difficult to address, because Razor Pages handle View and Controller integration well (a Razor Page can essentially serve as both), it was still not ideal. This oversight reflects a failure to fully embrace the principles of the MVC architecture during the design phase, which could have improved the clarity and structure of our system.

2. **Viewing History for Reservations and Orders**: Another feature that was missing from the original design was the functionality to view the history of reservations and orders. Initially, we underestimated its importance and did not include it as a requirement. However, as the project progressed, we realized near the end of the implementation phase that this feature was critical for both users and staff. While we were able to add this functionality, doing so late in the process introduced unnecessary complexity and time constraints. If we had identified this requirement during the design phase, we could have built it into the system from the start, making the implementation process more seamless and cohesive. This experience demonstrated the importance of anticipating user needs and considering additional features that may not appear immediately necessary but could improve the system's usability and functionality.

Flawed Aspects

1. **Misunderstanding of Model Responsibility**: One of the major flaws in our original design was a misunderstanding of the responsibilities of the "Model" component in the MVC architecture. In our initial design, we assigned the Model more responsibilities than it should have had, especially when it came to updating menu item prices and availability. As stated above, according to MVC principles, the Model should be responsible solely for storing and managing data, with no direct interaction with the business logic or other components (Rahman 2023). However, we mistakenly bundled actions such as price and availability updates with the data itself. This not only blurred the distinction between the Model and the rest of the system, but it also made the Model more complex and difficult to maintain. By misallocating responsibilities in this way, we created unnecessary dependencies and reduced design clarity. A clearer division of responsibilities would have allowed the system to be more modular and adaptable in the future.
2. **Overuse of the "Manager" Label**: Another flaw in our original design was the overuse of the term "Manager" when naming certain classes. While the term "Manager" may appear intuitive and simple, it resulted in an undesirable concentration of responsibilities. For example, we had classes labeled as "Order

Manager" or "Table Manager" that were responsible for a variety of tasks. In the case of the "Order Manager," this class was expected to handle the creation, updating, and deletion of orders. However, this resulted in a violation of the Single Responsibility Principle, which suggests that a class should be responsible for only one part of the functionality of the system (Martin 2017). Instead of having a single "Manager" class responsible for all CRUD operations, each operation (Create, Read, Update, Delete) should have been assigned its own class. This would have not only adhered to SRP but also made the system more flexible and easier to scale. The overuse of "Manager" also created ambiguity about what each class was actually responsible for, making the design less transparent.

3. **Uncertainty about the Need for the Invoice Class** (already mentioned above in *Responsibilities and Collaborators* section): Lastly, we struggled with whether to include an "Invoice" class in our design. At first, we were unsure whether this class would add value because, after implementing the basic responsibilities of the "Invoice," it appeared to be very similar to the "Order" class. This led us to consider whether the Invoice class was redundant. Finally, the decision to include it was made hastily, without fully considering the long-term consequences of having it as a separate class.

Lessons Learnt

Importance of Clear Responsibilities in MVC Architecture

One of the most significant lessons learned was the necessity of clearly defining the responsibilities of each component within the MVC (Model-View-Controller) architecture. Initially, our design misallocated responsibilities, particularly concerning the User model, which was burdened with tasks such as placing orders and processing payments. This misunderstanding led to a convoluted structure that complicated maintenance and scalability. By redefining the User model's role to focus solely on data management, we adhered more closely to MVC principles, ensuring that models serve their intended purpose without overstepping into business logic or user interaction responsibilities. This experience highlighted the need for a clear separation of concerns to enhance system coherence and maintainability.

Value of Comprehensive Design Analysis

The project underscored the importance of conducting a thorough design analysis at the outset. Our team invested significant effort in identifying potential challenges and defining assumptions related to the restaurant's operations. This proactive approach not

only facilitated a smoother implementation phase but also minimized unexpected obstacles. By establishing a solid foundation through detailed problem analysis, including simplifications to reduce complexity, we were able to navigate the development process more effectively. Future projects should prioritize this level of analysis to ensure that all aspects of the design are well-considered from the beginning.

Need for User-Centric Features

Another critical lesson was the necessity of anticipating user needs during the design phase. The omission of features such as viewing reservation and order histories initially seemed minor but ultimately proved to be a significant oversight. Recognizing user requirements early would have allowed us to integrate these functionalities seamlessly into the system, avoiding last-minute complications during implementation. This experience serves as a reminder that understanding user interactions and expectations is essential for creating a functional and user-friendly application.

Avoiding Overcomplication in Class Responsibilities

The project also revealed pitfalls associated with overcomplicating class responsibilities, particularly through the overuse of generic terms like "Manager." Classes such as "Order Manager" were assigned multiple responsibilities, violating principles like the Single Responsibility Principle (SRP). This not only made classes harder to manage but also introduced unnecessary dependencies within the system. In hindsight, each operation—Create, Read, Update, Delete—should have been encapsulated within its own class to promote modularity and flexibility. Future designs should prioritize clarity in class naming and responsibility allocation to foster a more organized architecture.

Emphasizing Design Patterns Appropriately

While exploring various design patterns proved beneficial, it became clear that not all patterns are necessary for every project. Our primary reliance on the MVC pattern provided a structured framework that supported maintainability and scalability. However, recognizing when a design pattern is appropriate is crucial; unnecessary complexity can arise from attempting to implement patterns that do not align with project needs. Future projects should focus on selecting design patterns judiciously based on specific requirements rather than adhering rigidly to theoretical ideals.

Implementation

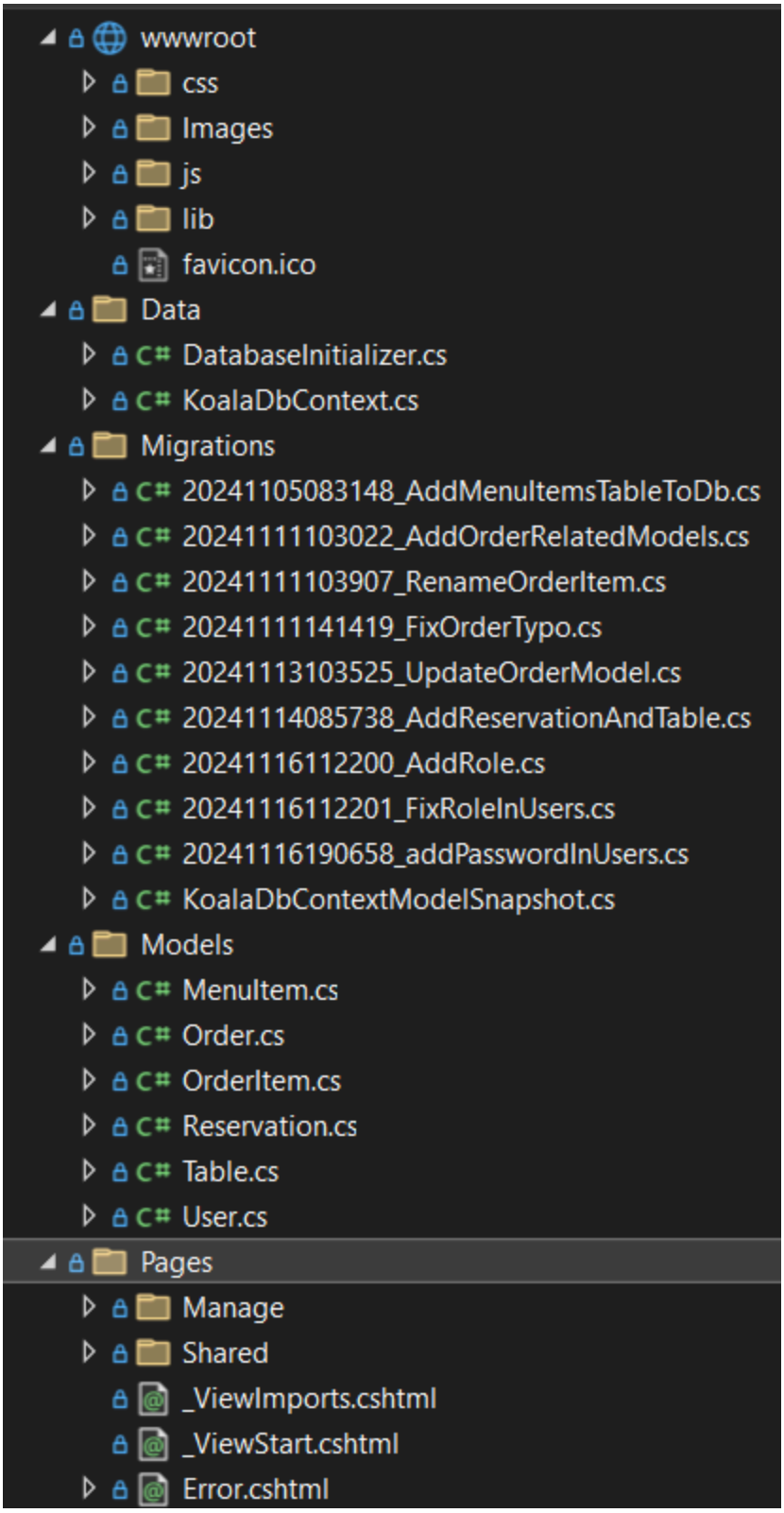
Coding Standards

Naming Convention

To maintain readability and uniformity throughout the codebase, we adhered to a consistent naming convention standard while developing our web application. For classes, methods, and properties, we consistently used PascalCase to ensure clarity and alignment with .NET naming guidelines (Cwalina & Abrams 2008). For example, we named a class MenuItems, a property Type, and a method GetUserIDFromCookie. Meanwhile, variables and parameters were named using camelCase to distinguish them from class members while emphasizing their scope and purpose (e.g., userCookie, userID). This structured approach to naming conventions not only improved the project's overall maintainability but also facilitated seamless collaboration among team members, reducing confusion and potential code errors.

File Organization

Throughout the development of our web application, we adhered to a clear and consistent file organization standard to improve code readability and maintenance. Our project structure is logically organized, with separate folders for Models, Razor Pages (which can be treated as paired view-controller setups), and Database-related classes, so that each component is easily identifiable and accessible. Additionally, we followed meaningful and descriptive file naming conventions by naming files based on their content and purpose. For example, a Razor Page for placing orders is named Order.cshtml, which reflects its functionality and role within the application. This structured approach has streamlined development, made debugging easier, and facilitated smooth collaboration among team members.



A snippet of our project file organization

Best Practices

Security

During the development of our web application, we have prioritized security by following best practices that protect both user data and the system. We ensure that input validation is implemented at all appropriate points to prevent vulnerabilities such as SQL injection and protect the application from malicious data manipulation.

For authentication and authorization, they have been designed with a middleware approach to ensure secure access control. The system validates user roles by deserializing the user cookie to retrieve the Role and Email, determining access permissions. For example, if a user who is not an Admin or Staff attempts to access any path under /Manage, they are redirected to the UnauthorizedAccess page, and the attempt is recorded with their email or IP address. Similarly, access to /Manage/UsersManager is restricted to Admins, explicitly preventing Staff from unauthorized access. This granular control, reinforced by context.Response.Redirect, ensures robust role-based security and secure navigation within the application (Microsoft n.d.).

Additionally, we have prioritized data protection by encrypting all sensitive information and requiring HTTPS for all communications, ensuring data integrity and confidentiality throughout the application. Razor Pages supports these security measures seamlessly, allowing us to concentrate on developing a stable and maintainable codebase.

Error Handling

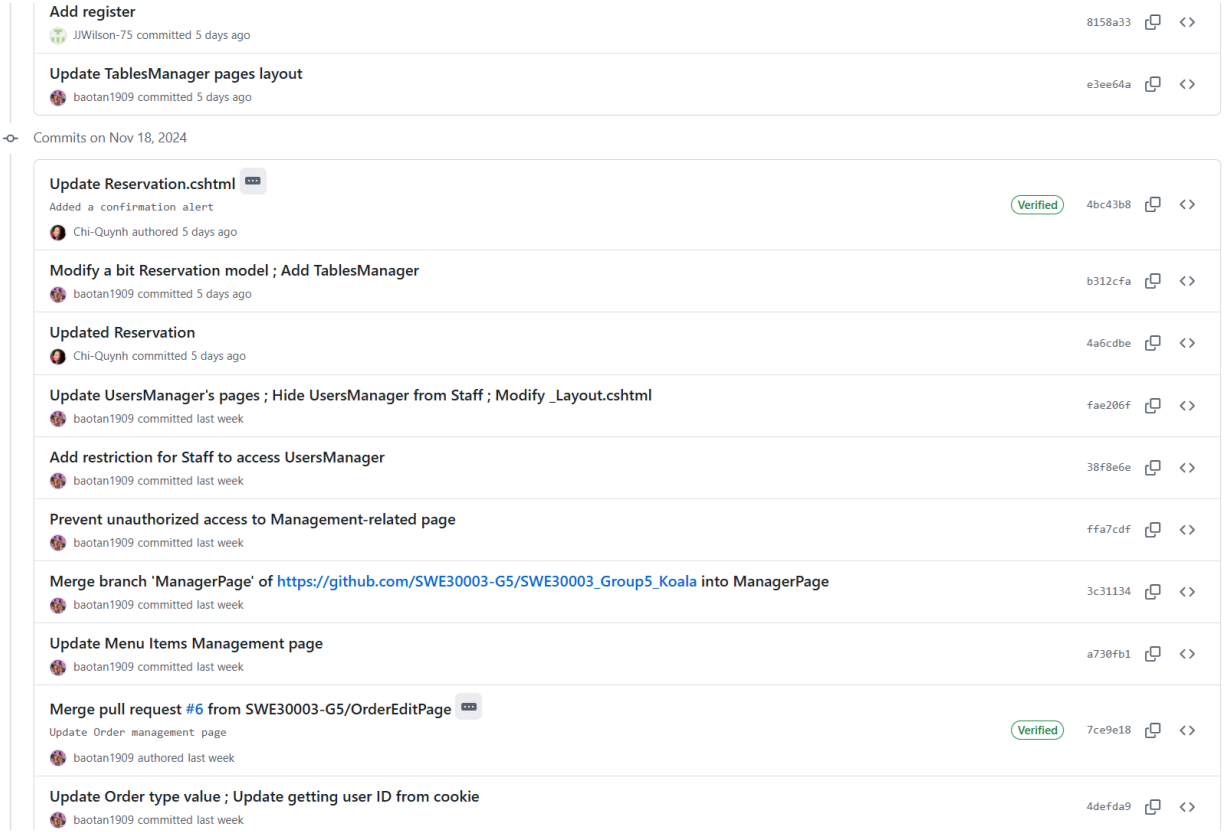
In addition to strong security measures, we have prioritized effective error handling to further improve the reliability and user experience of our web application.

To handle server-side errors in C#, we consistently use try...catch blocks to gracefully manage exceptions, thus preventing unexpected application crashes (Warren 2024). Detailed error information is carefully logged using the ILogger interface, allowing us to efficiently track, analyze, and resolve issues (Krishnareddy 2024).

On the client side, we also use try...catch and console.log() to detect and debug JavaScript errors during runtime, ensuring smooth interaction and functionality within the browser (Mozilla Developer Network 2024). By combining these practices, we ensure robust error tracking across both server and client environments, which contributes to our application's overall stability and usability.

Version Control

We used GitHub as a centralized repository for our codebase to follow best practices for version control. To maintain a clear workflow, we implemented a branching strategy in which feature-specific branches are created for new functionalities or fixes, and only merged back into the master branch after thorough testing to ensure they are free of errors (Adesoji1 2023; Twumasi 2024). This approach reduces conflicts and improves code stability. Additionally, we established clear guidelines for naming commit messages, requiring them to be descriptive and transparent about the changes made, such as adding new functions, fixing bugs, or removing outdated code (Ayodeji 2019; Manzoor 2024). This structured use of GitHub has improved collaboration, made change tracking easier, and ensured that the team's development process runs smoothly.



A snippet of commit history on our Github repo

Execution and Operation

Development and Testing Platform

Our implementation was developed and tested on **Windows 11** as the primary operating system, with **Visual Studio 2022** as the integrated development environment

(IDE). The application was built with **ASP.NET Core 8.0** and the **Razor Pages** framework, and the database layer was managed with **SQLite** integrated with **Microsoft.EntityFrameworkCore.SQLite** and **Microsoft.EntityFrameworkCore.Tool** frameworks. These tools allowed us to efficiently manage database migrations, seed data, and handle database schema updates during development.

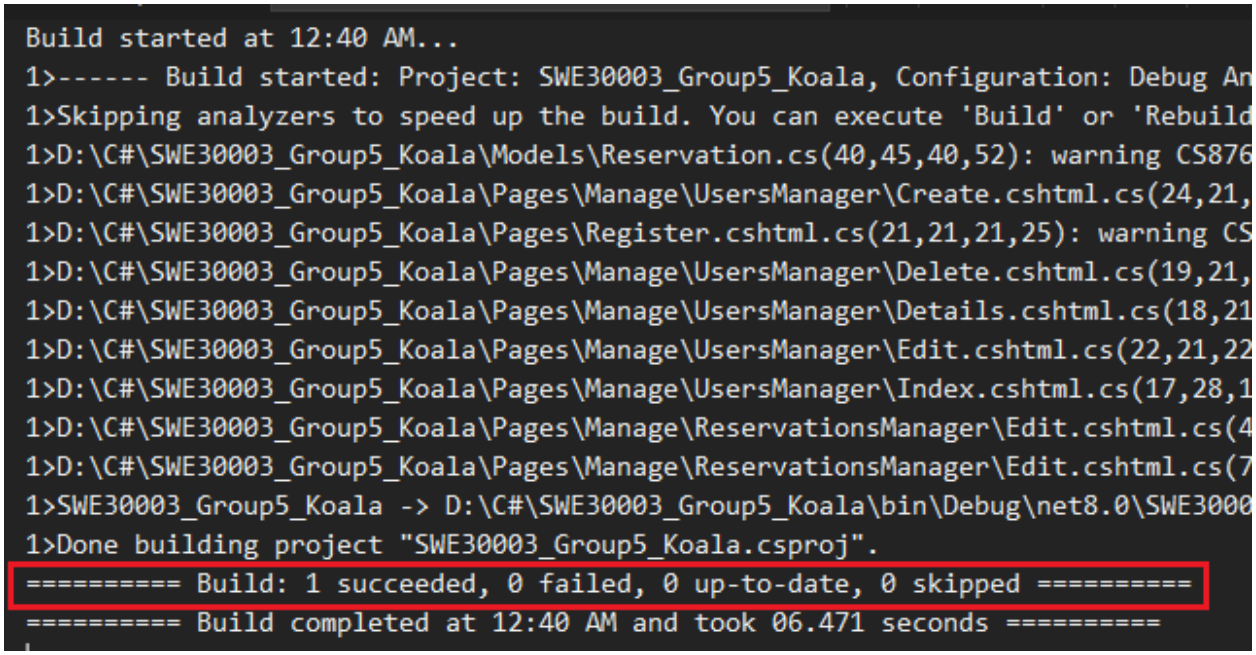
Additionally, we recommend installing **DB Browser for SQLite** to easily view and manage the SQLite database created by the application. This tool offers a simple interface for inspecting and editing the SQLite database, making it easier to debug and visualize the data.

Moreover, we tested the application on three browsers: **Firefox** (our primary testing browser), **Microsoft Edge**, and **Google Chrome**, to ensure cross-browser compatibility and consistency across major platforms.

To deploy the application, users need to clone the repository, install the necessary NuGet packages (alternatively, download the .zip file we provided), and run the project directly from Visual Studio using either the IIS Express or Kestrel server. When the application starts, the database will automatically initialize, taking advantage of Entity Framework Core's migration capabilities to ensure that the database schema matches the application's requirements.

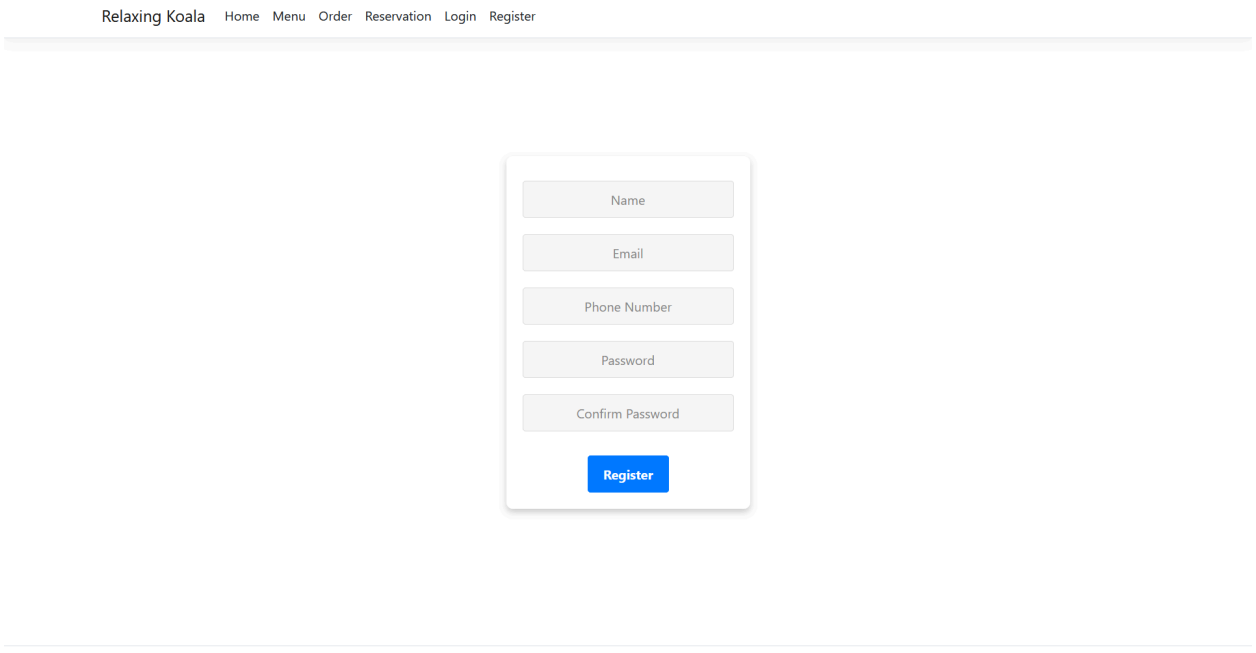
Evidence of Compilation

The screenshot below shows a successful compilation of the application.

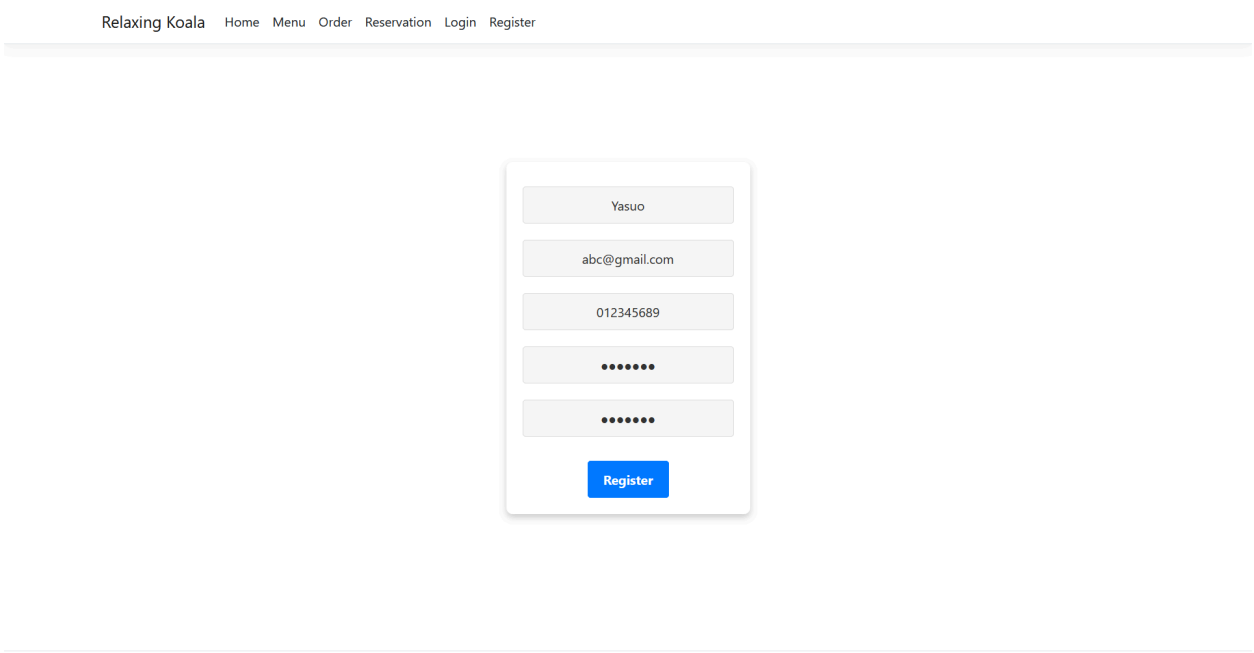


Scenario 1: User Register

1. Starting state: The Register page is empty

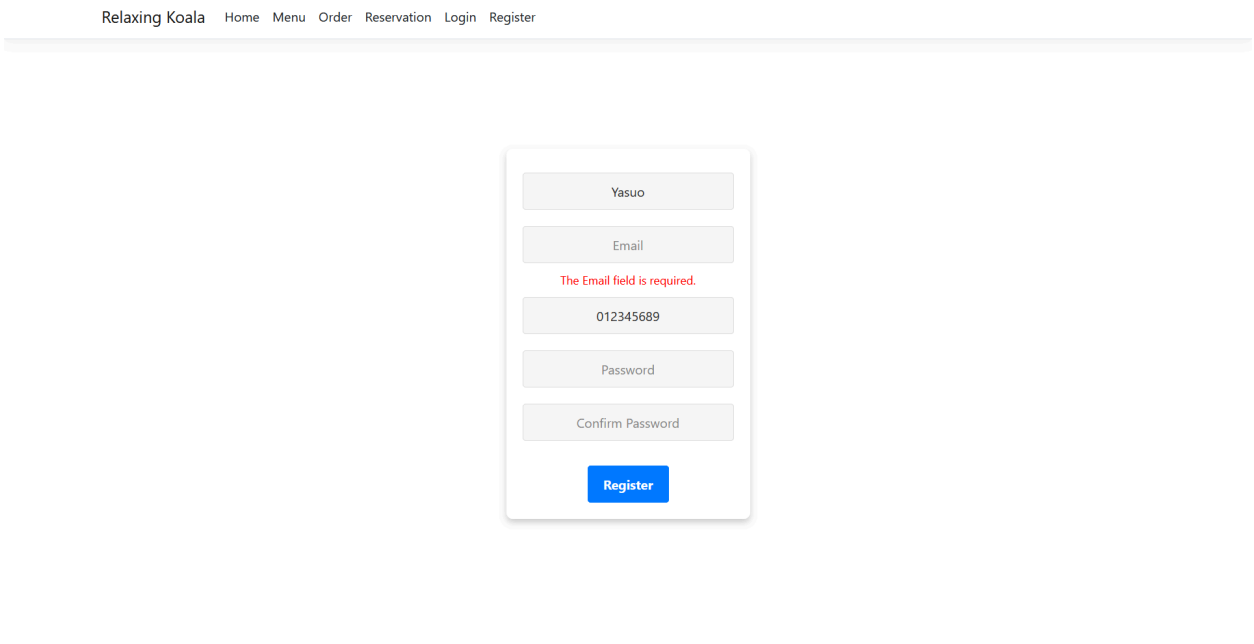


2. Inputting data: User enter the information for registering new account

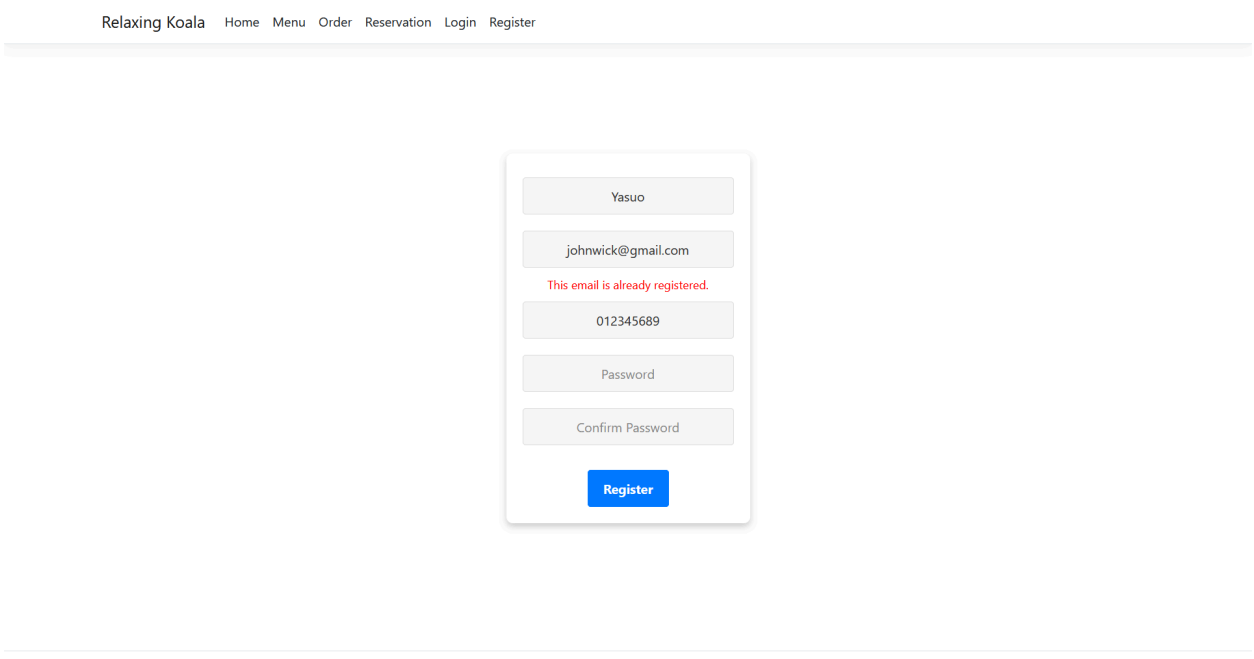


3. Input validation:

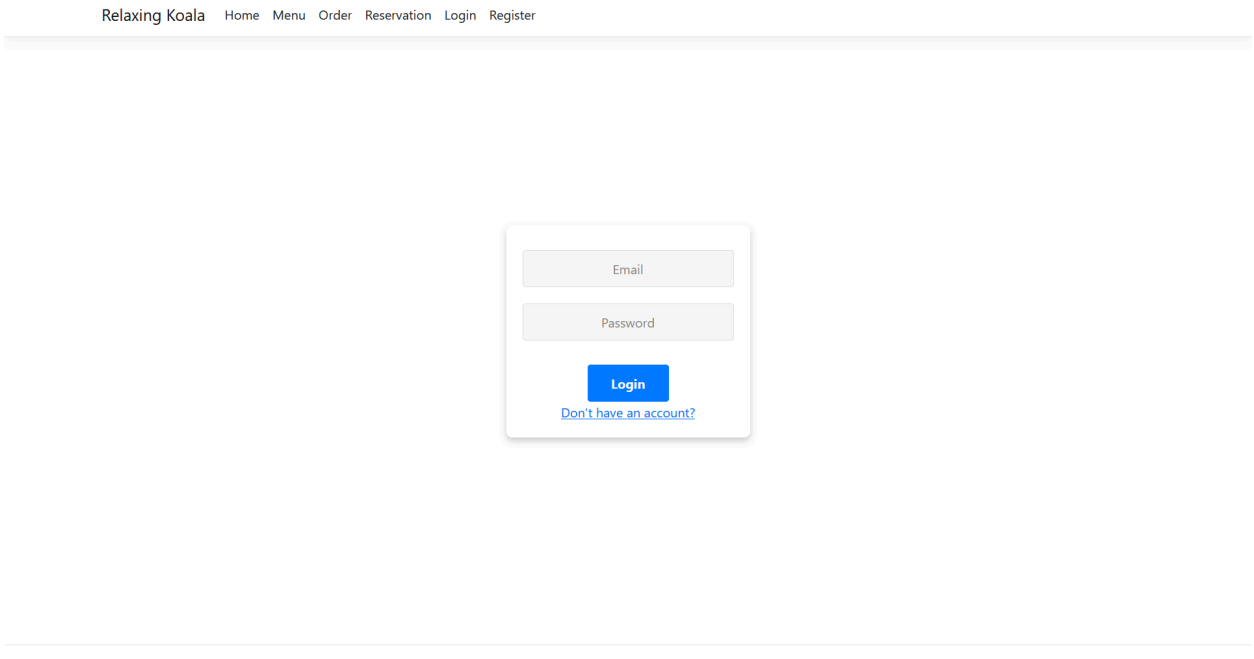
a. User didn't fill in a required field:



b. User input an email which is already registered previously

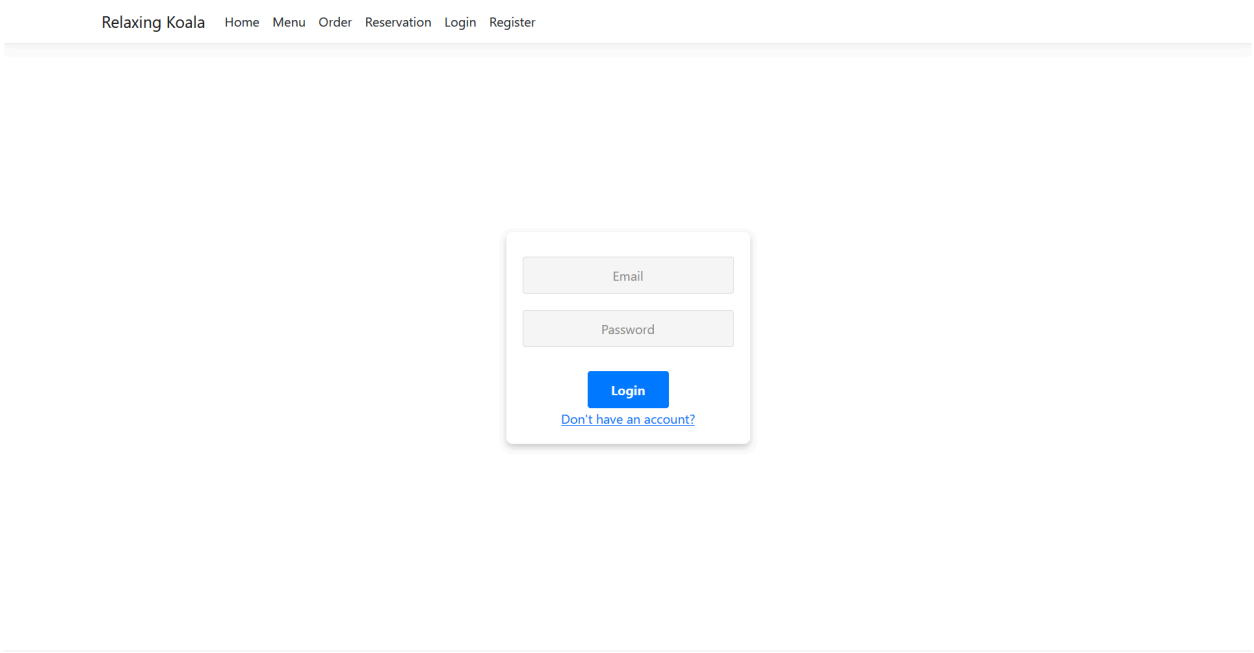


4. Correct input: When users enter all the correct and non-duplicate input, they will be redirected to the Login page.

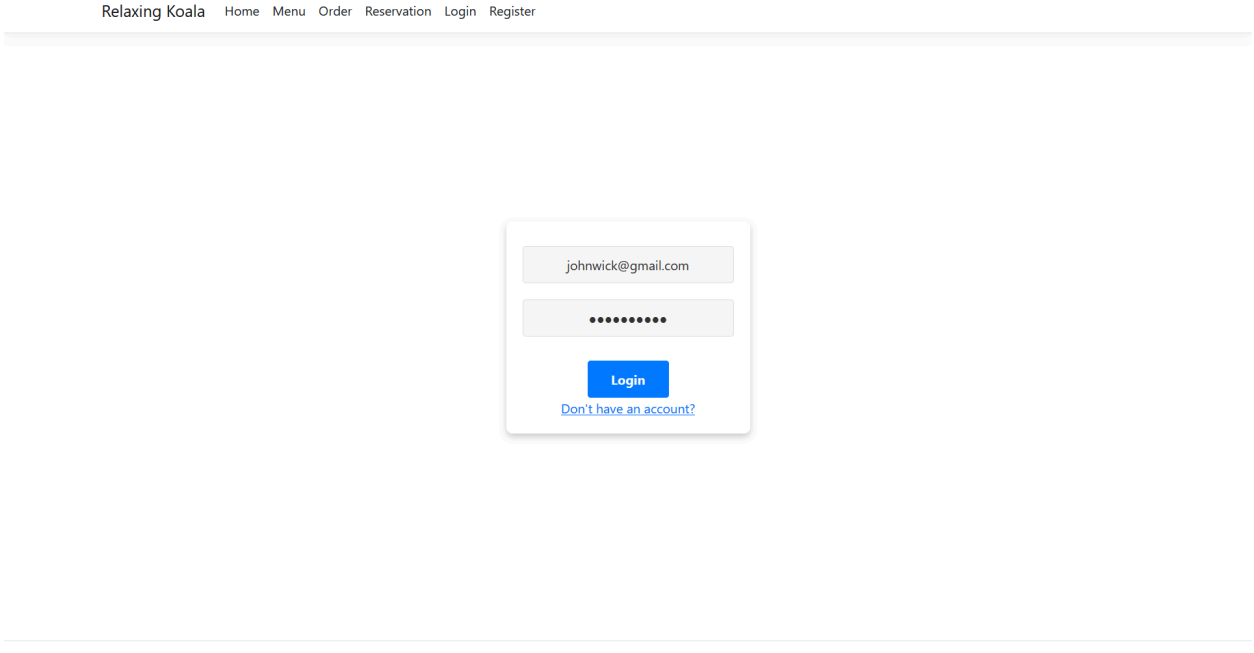


Scenario 2: User Login

1. Starting state: Login page is empty.

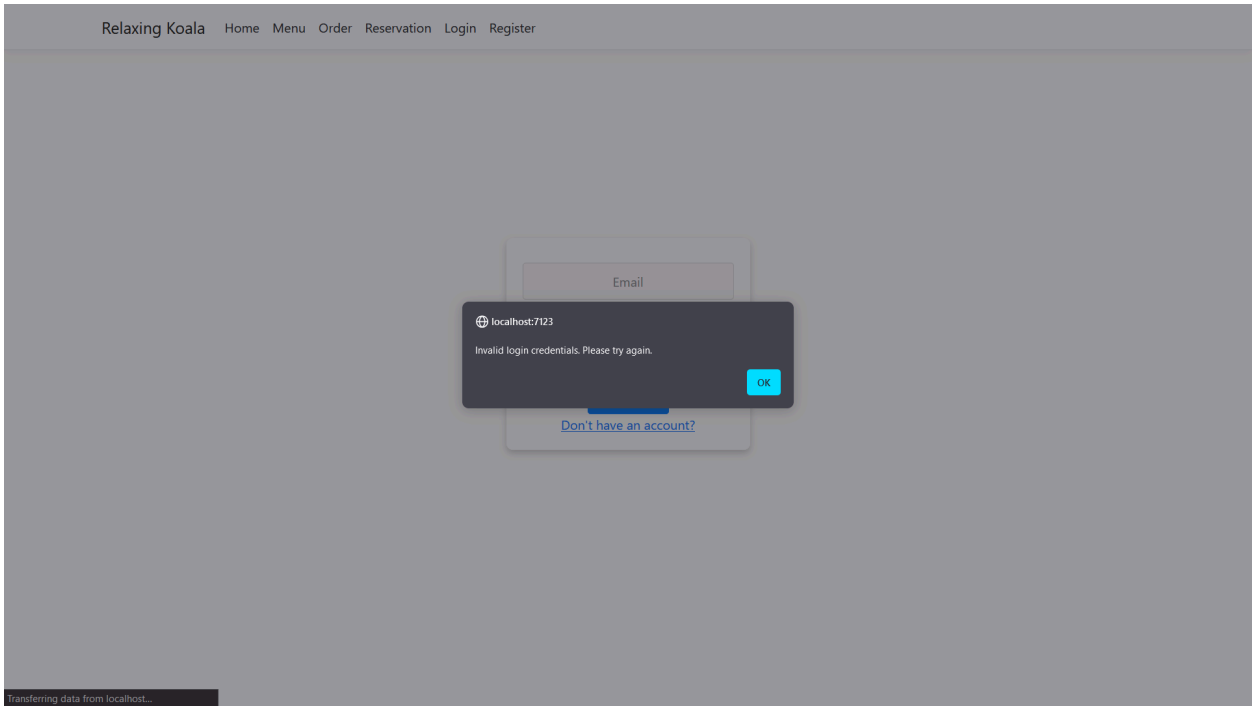


2. Inputting data: User enters the information to log in.

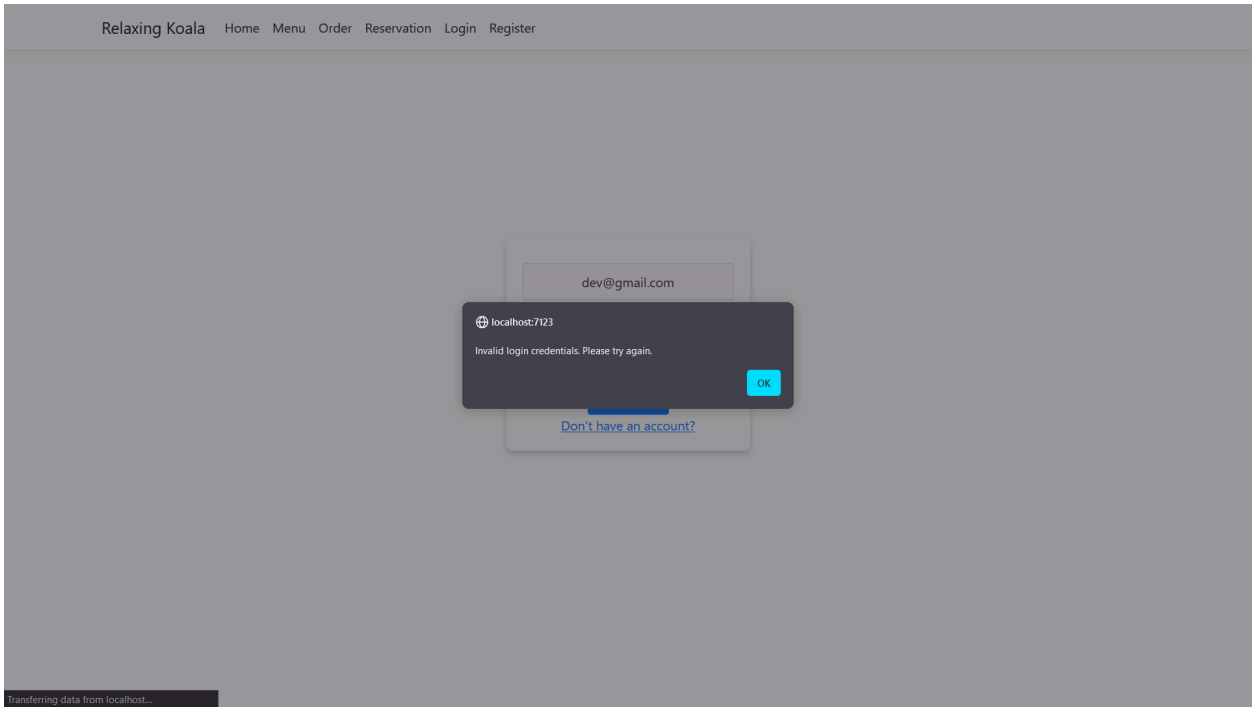


3. Input validation:

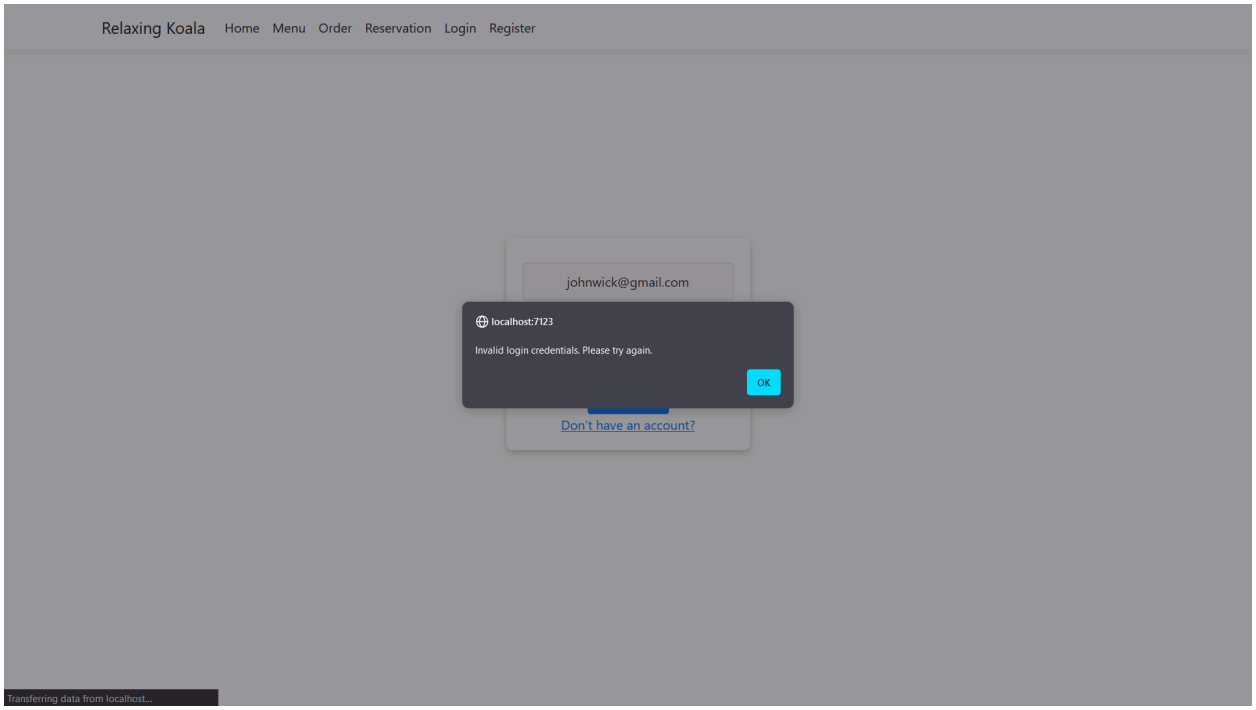
a. User didn’t fill in a required field:



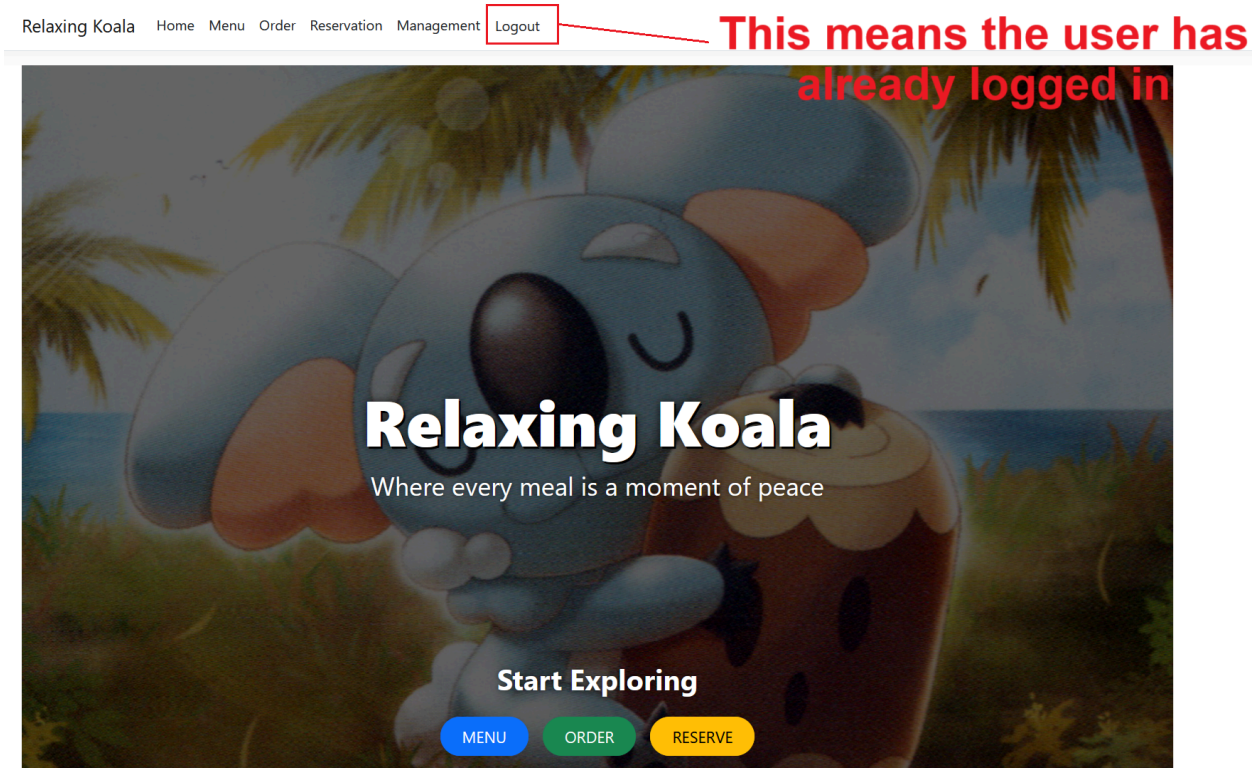
b. User input an email which is not existing in database



c. User enter correct email but incorrect password

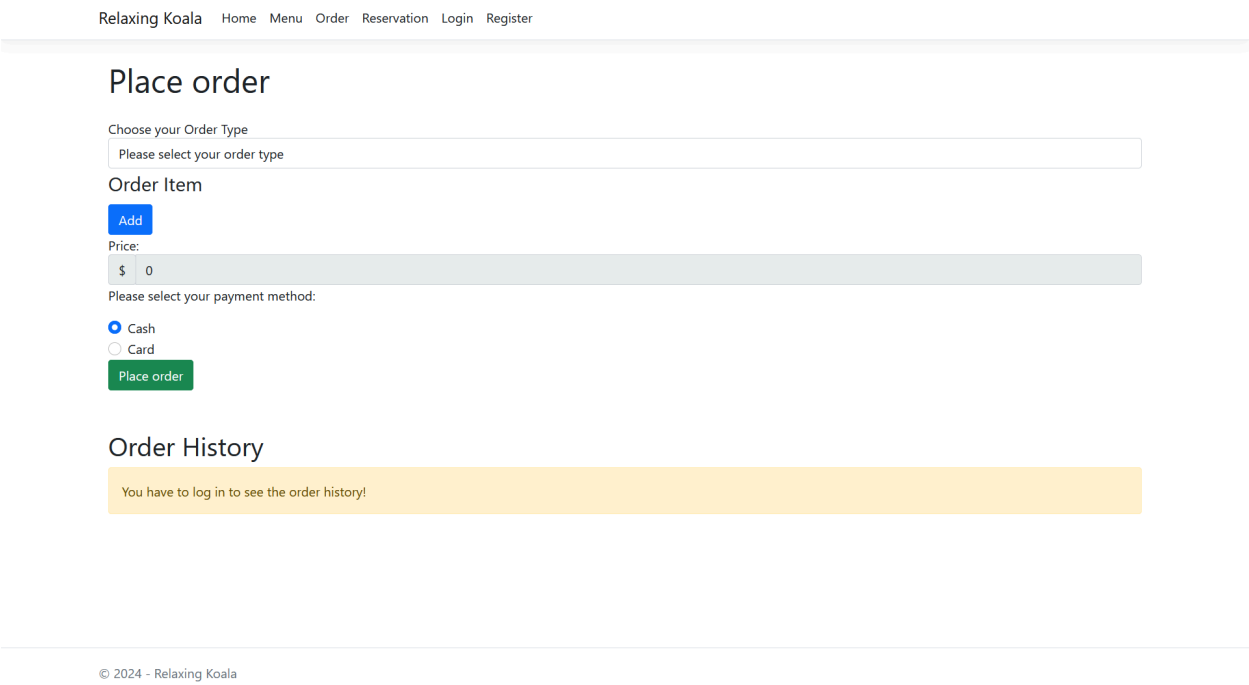


4. Correct input: Once a user logs in successfully, they will be redirected to the Homepage. The navigation bar will then display "Logout" instead of "Login" and "Register."

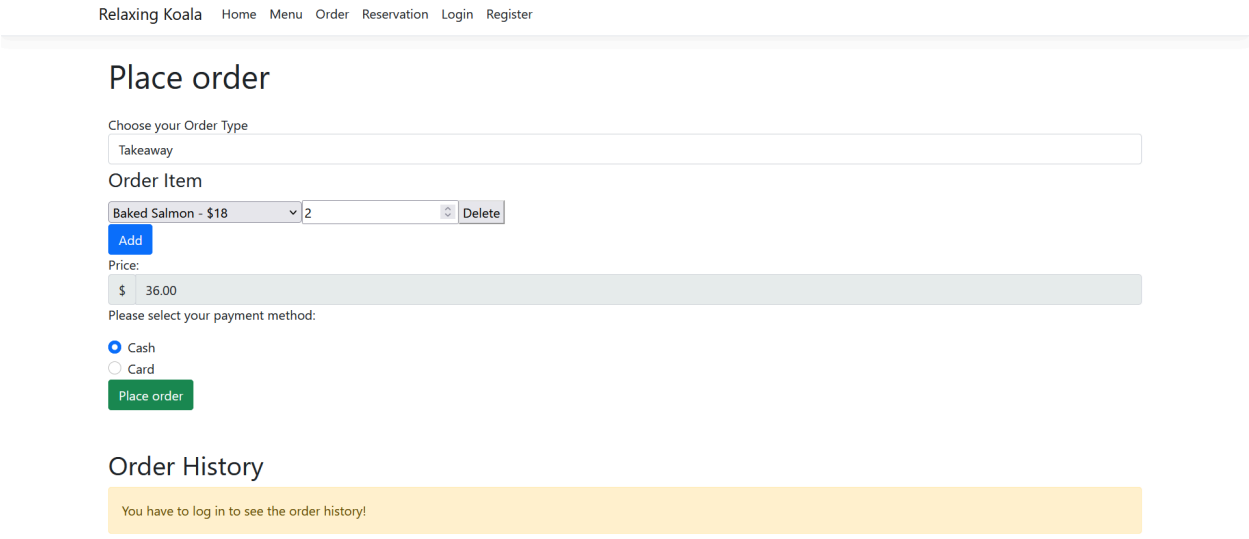


Scenario 3: Place Order

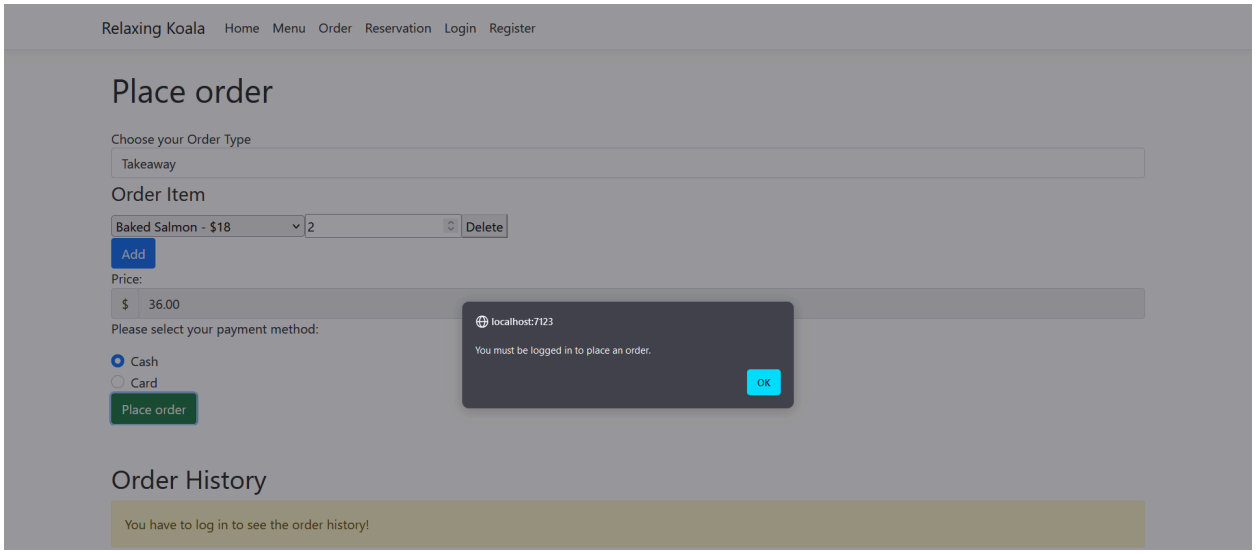
1. Starting state: The order page is displayed, allowing users to add items, select an order type, and choose a payment method.



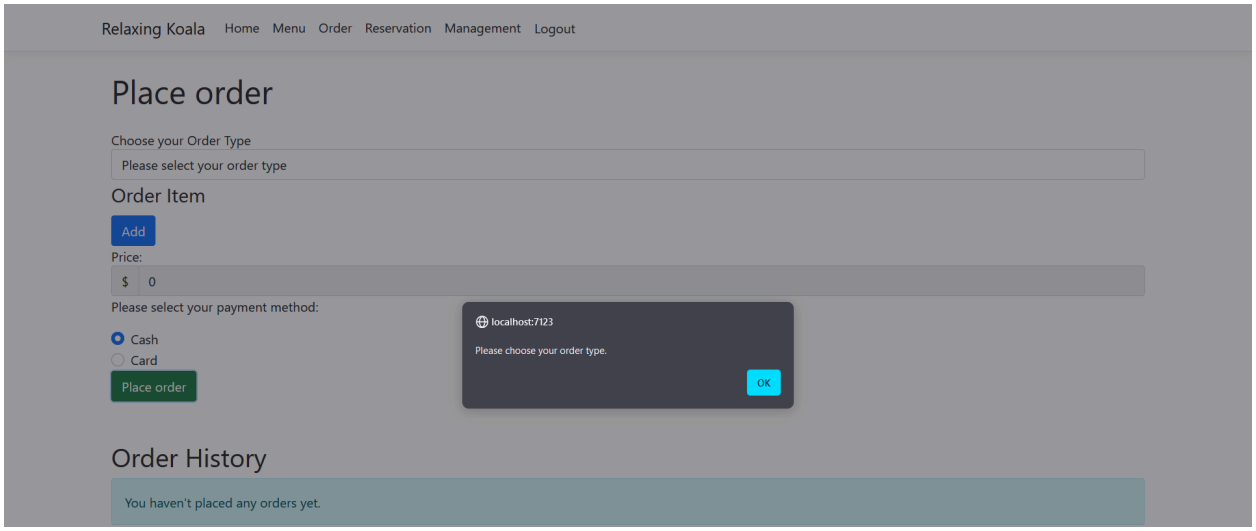
2. Inputting data:
a. User selects "Takeaway" as the order type.
b. User adds a new order item: Baked Salmon (Quantity: 2).
c. User chooses "Cash" as the payment method.



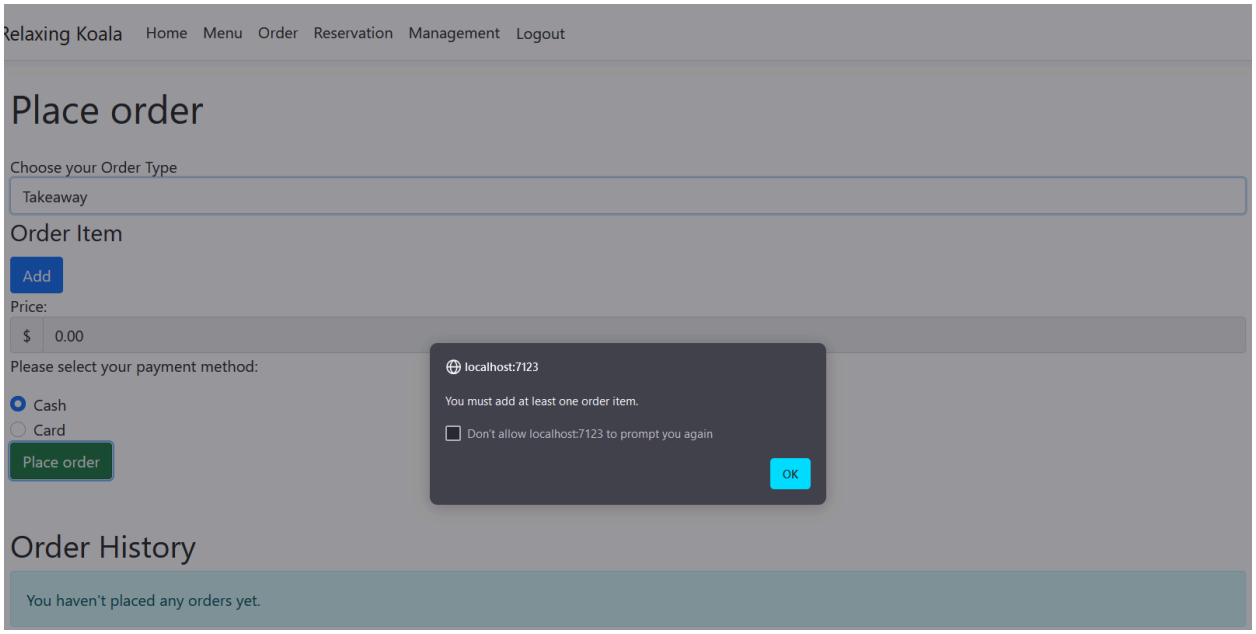
3. Input validation:
a. User has not logged in.



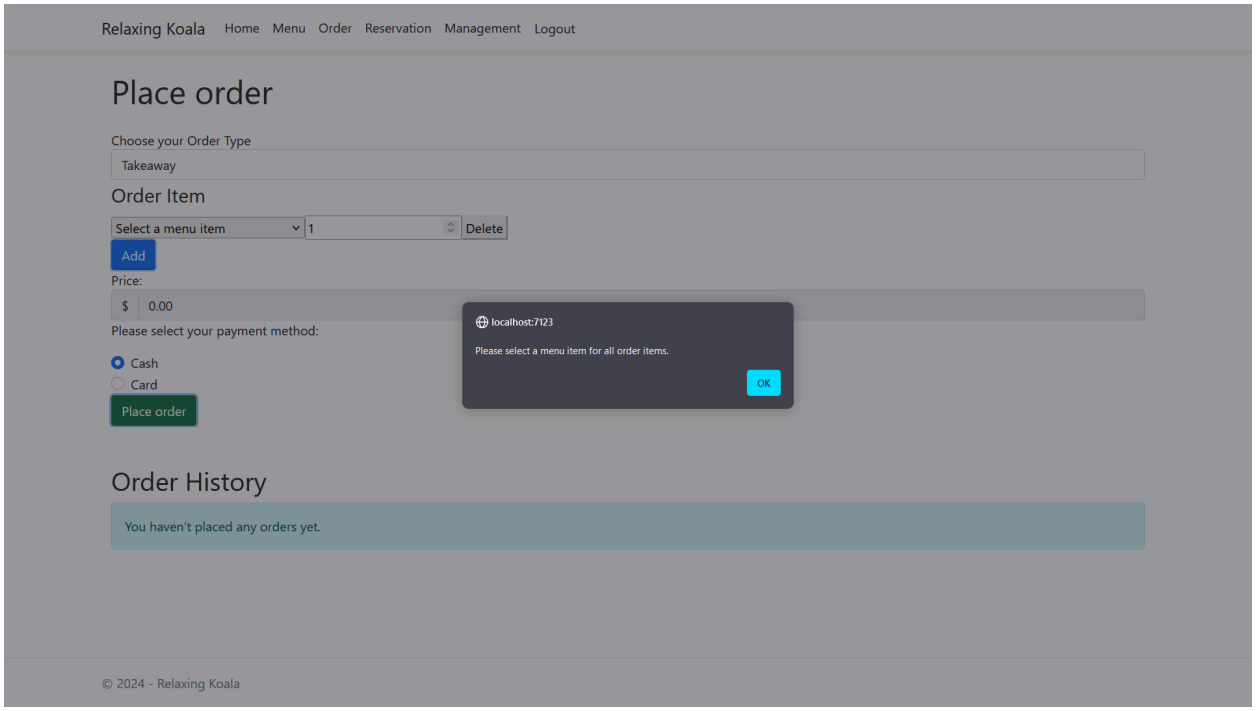
b. User has not selected the order type.



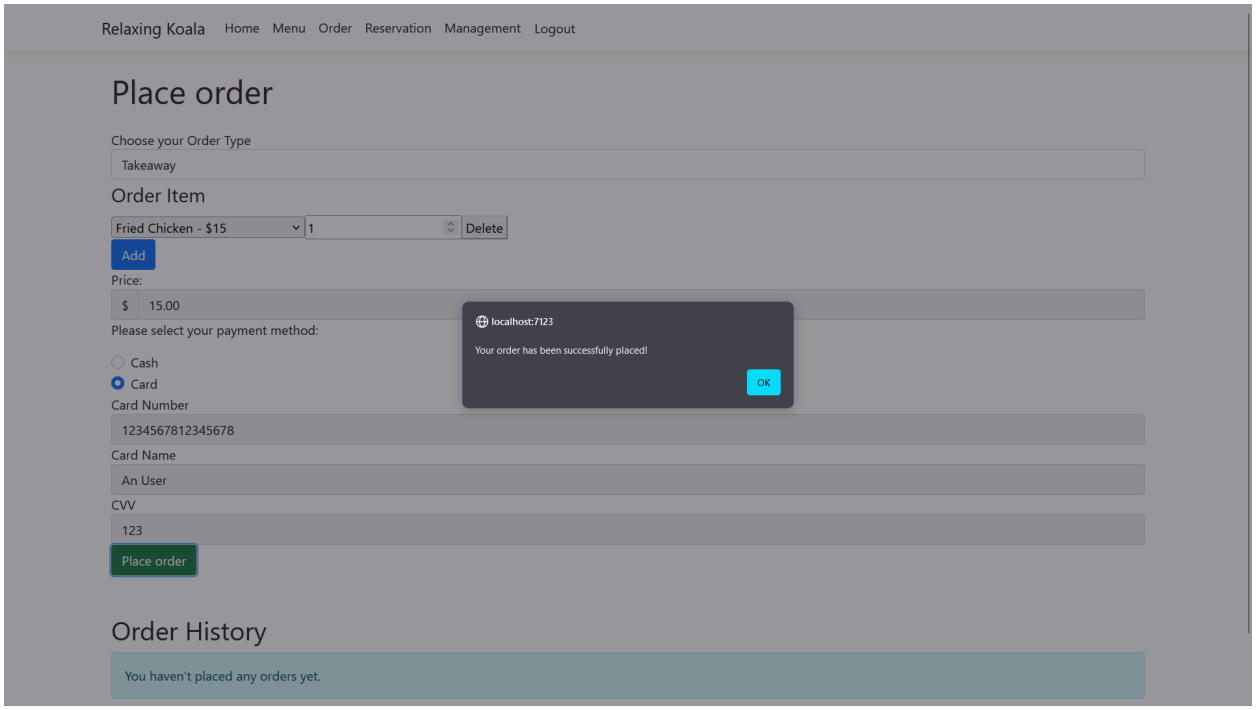
c. User has not added order item



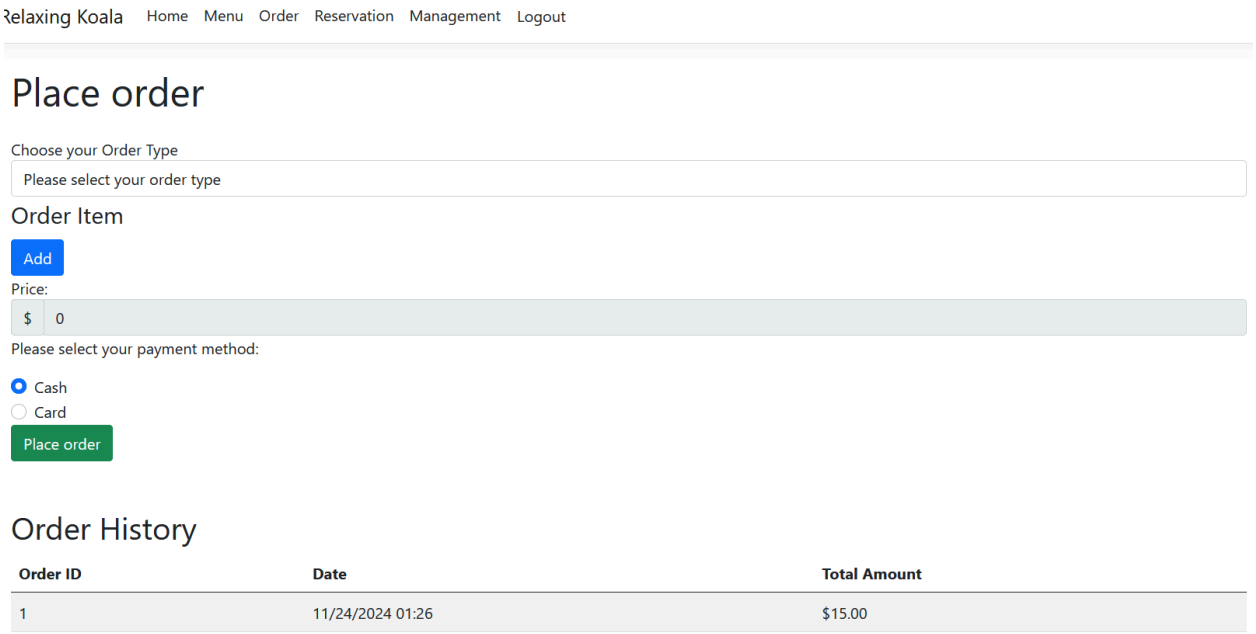
d. User added an order item but has not chosen the specific item.



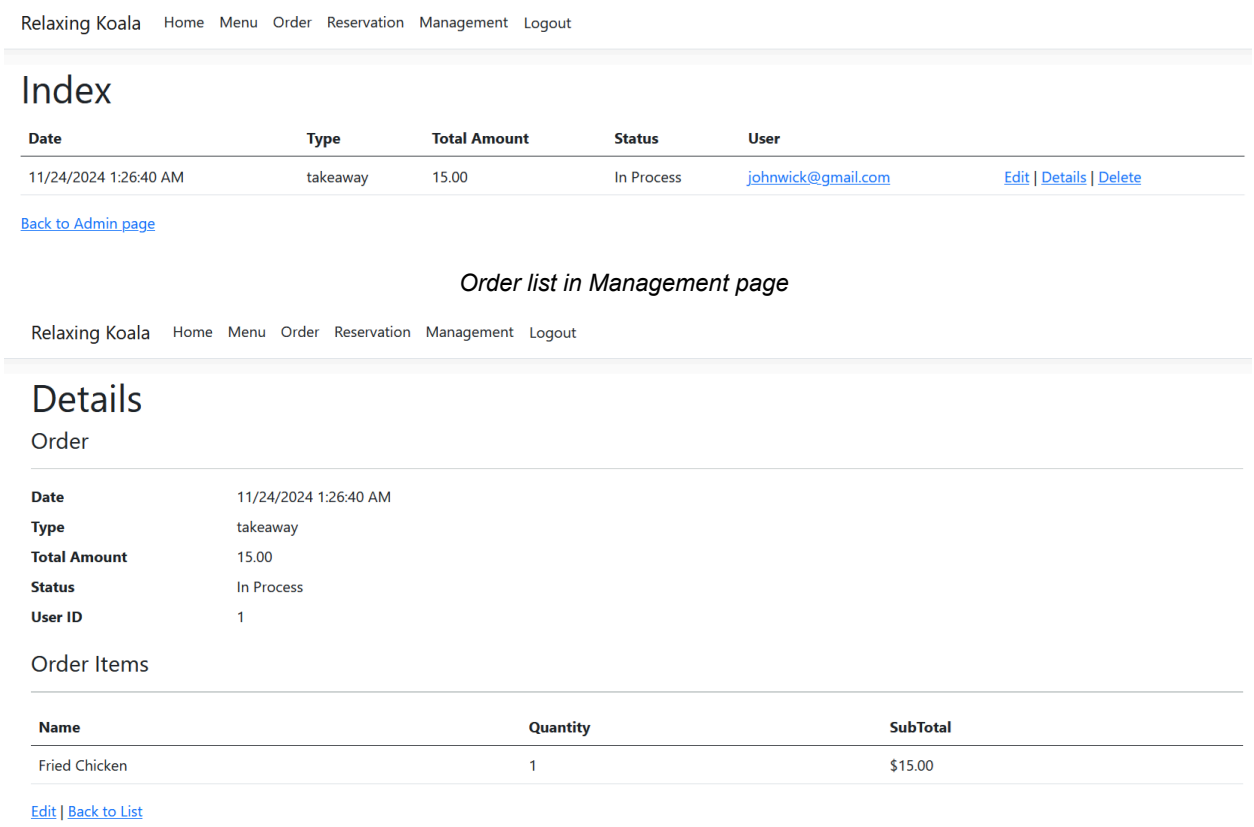
4. Correct input: User confirmed the order after providing all required information.



5. Users can view their past orders in the Order History.



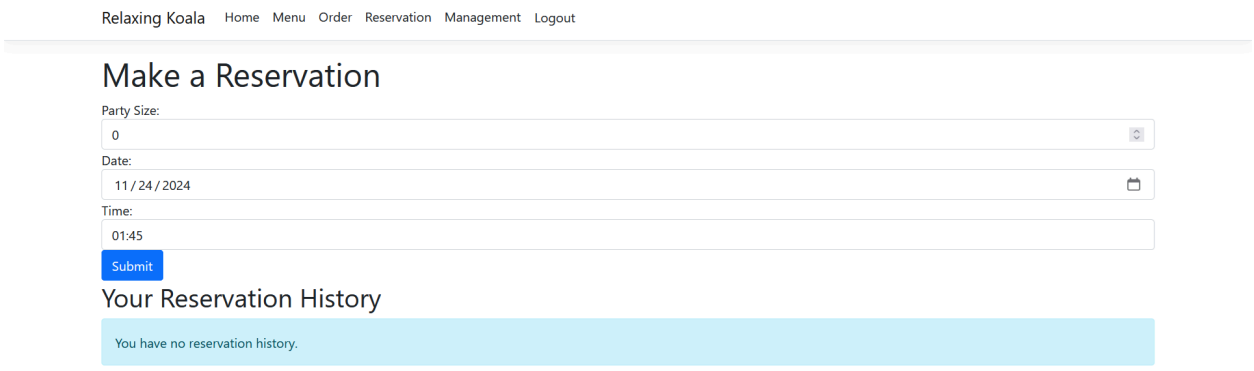
6. Order details are also visible in the management page.



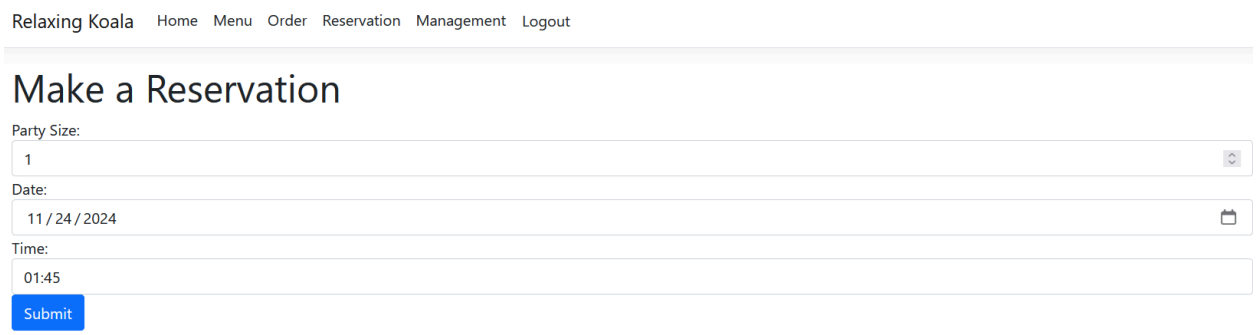
Order list in Management page

Scenario 4: Make Reservation

1. Starting state: Reservation page is displayed, pre-populated with today's date and the nearest 15-minute time slot (:00, :15, :30, :45).

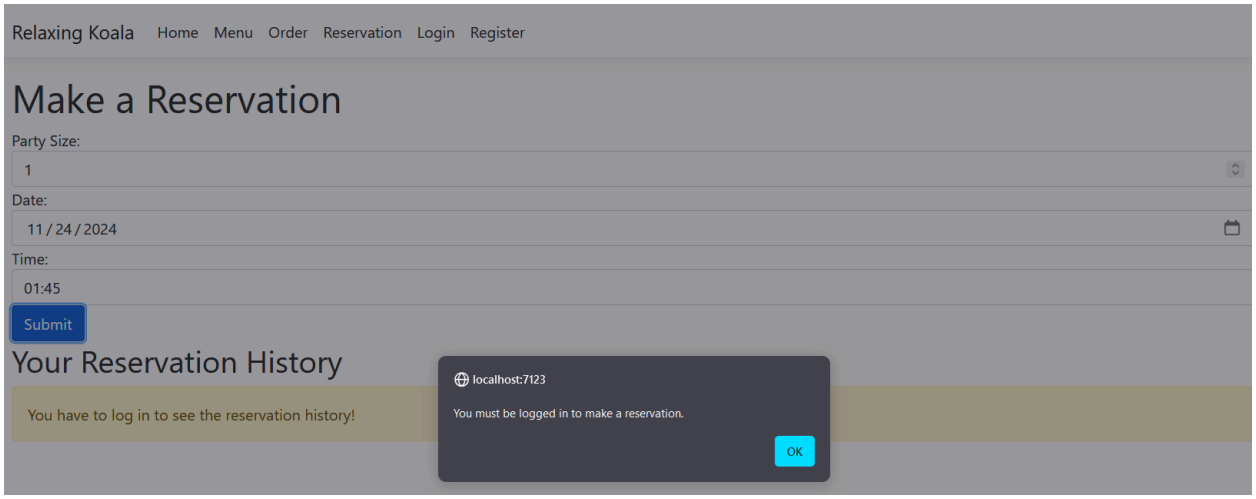


2. Inputting data: User provides necessary information to complete the reservation.

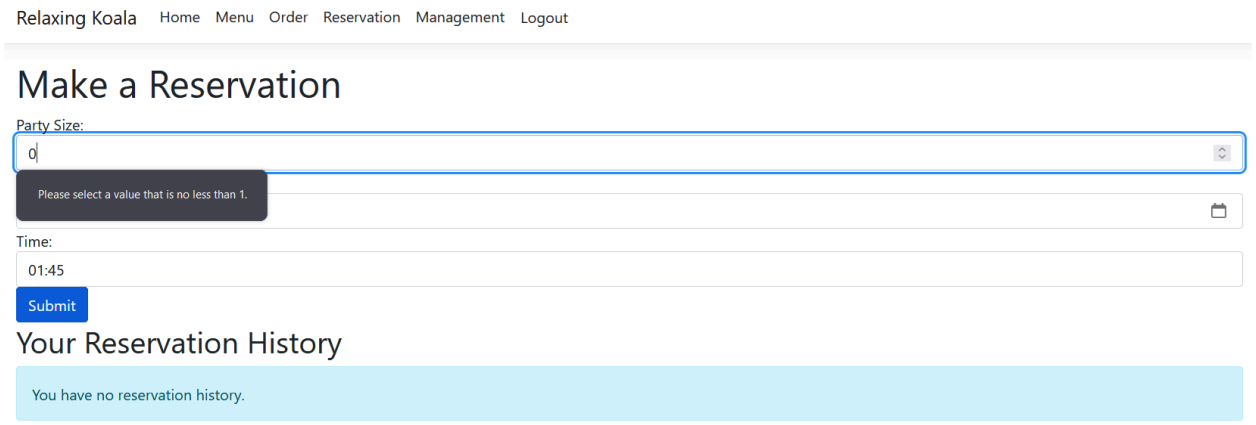


3. Input validation:

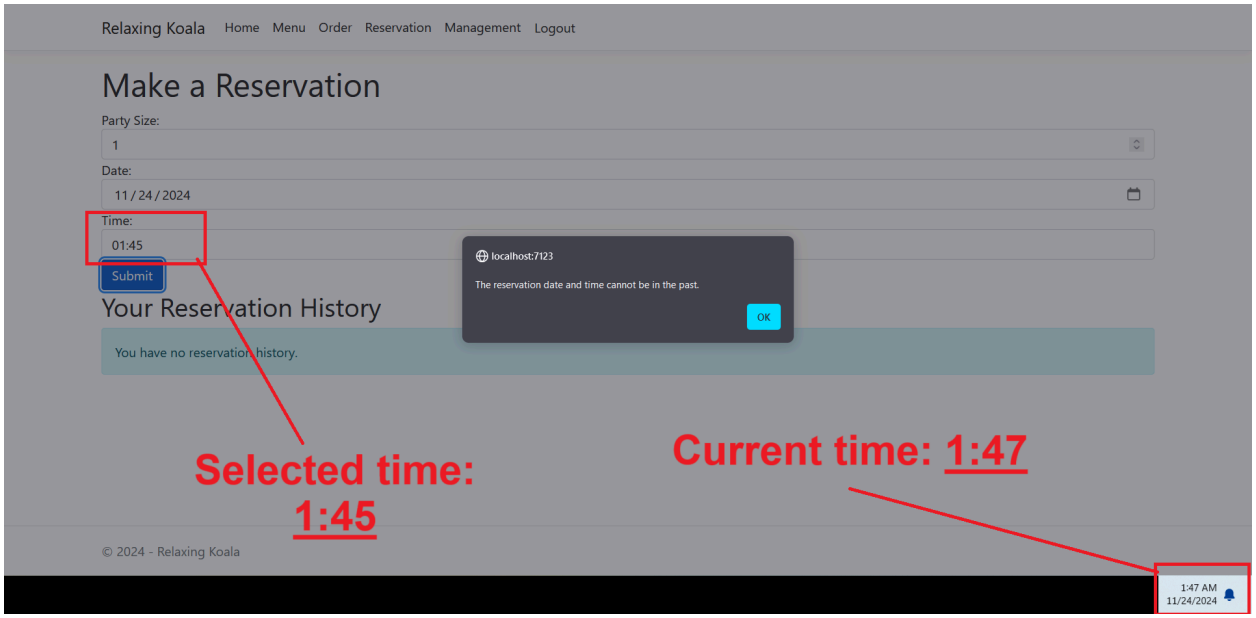
1. User attempts to make a reservation without logging in.



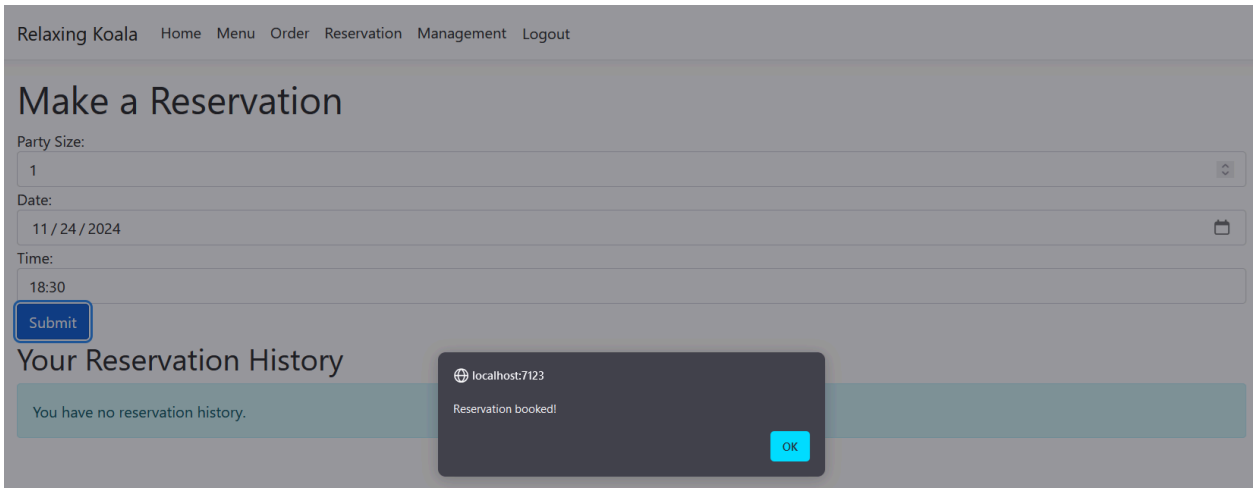
2. User fails to specify the number of guests.



3. User selects a past time slot.

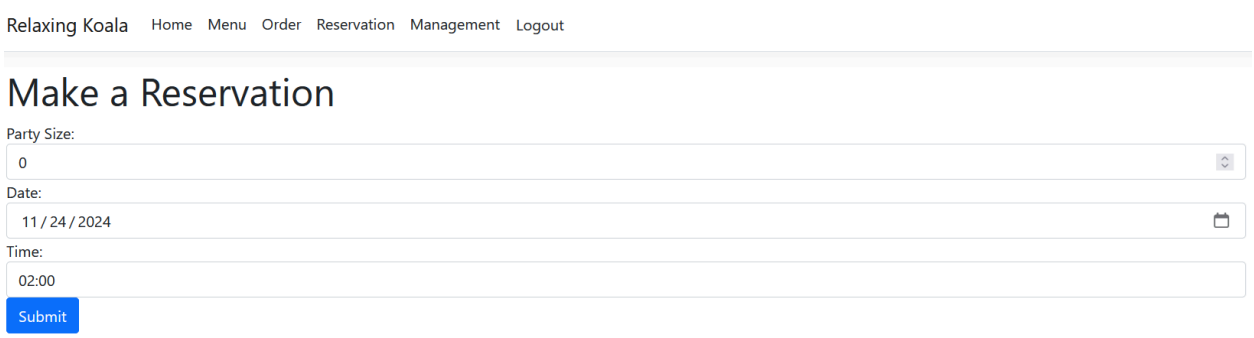


4. Correct input: User has entered all valid input.

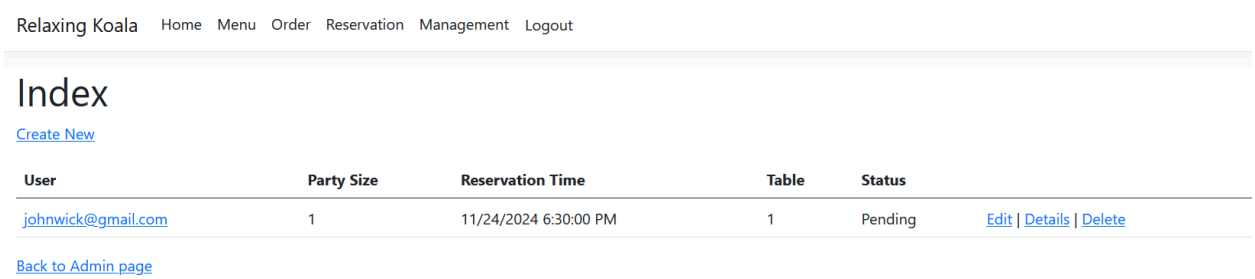


Similar to placing an order, reservations are recorded in both the user's history and the staff's management page.

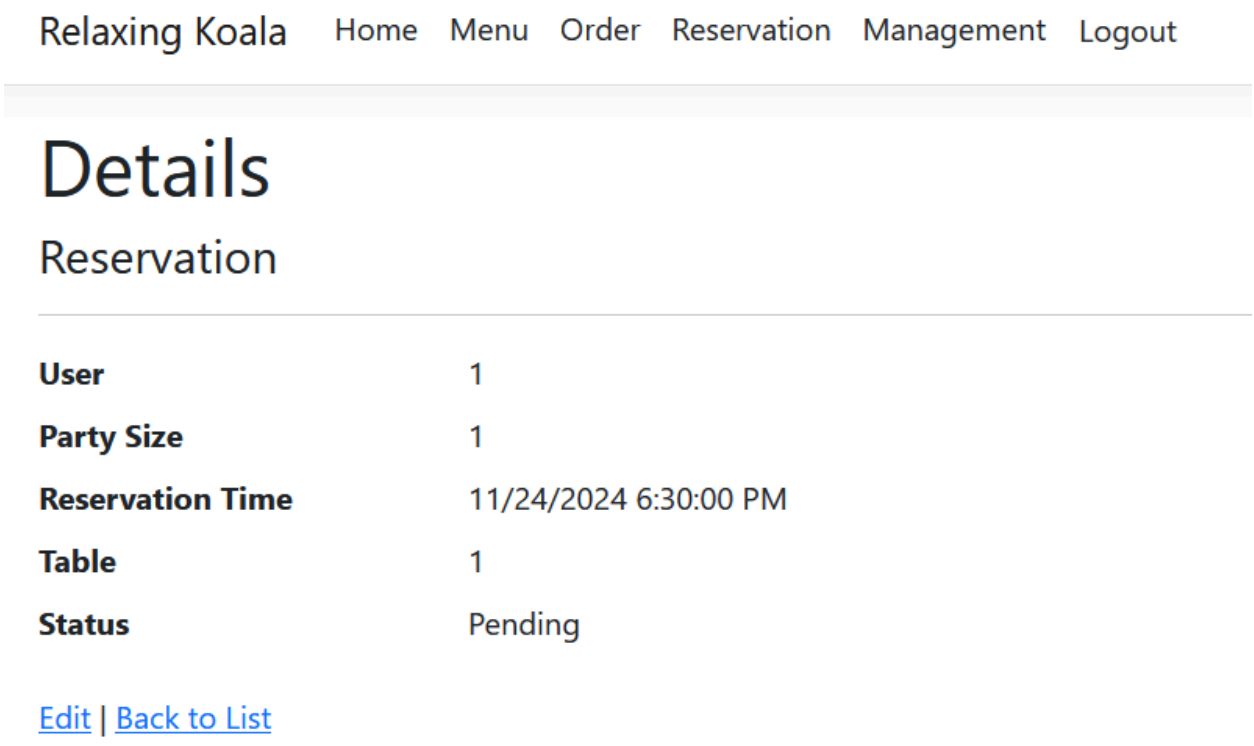
a. Reservation history



b. Reservation list in Management page

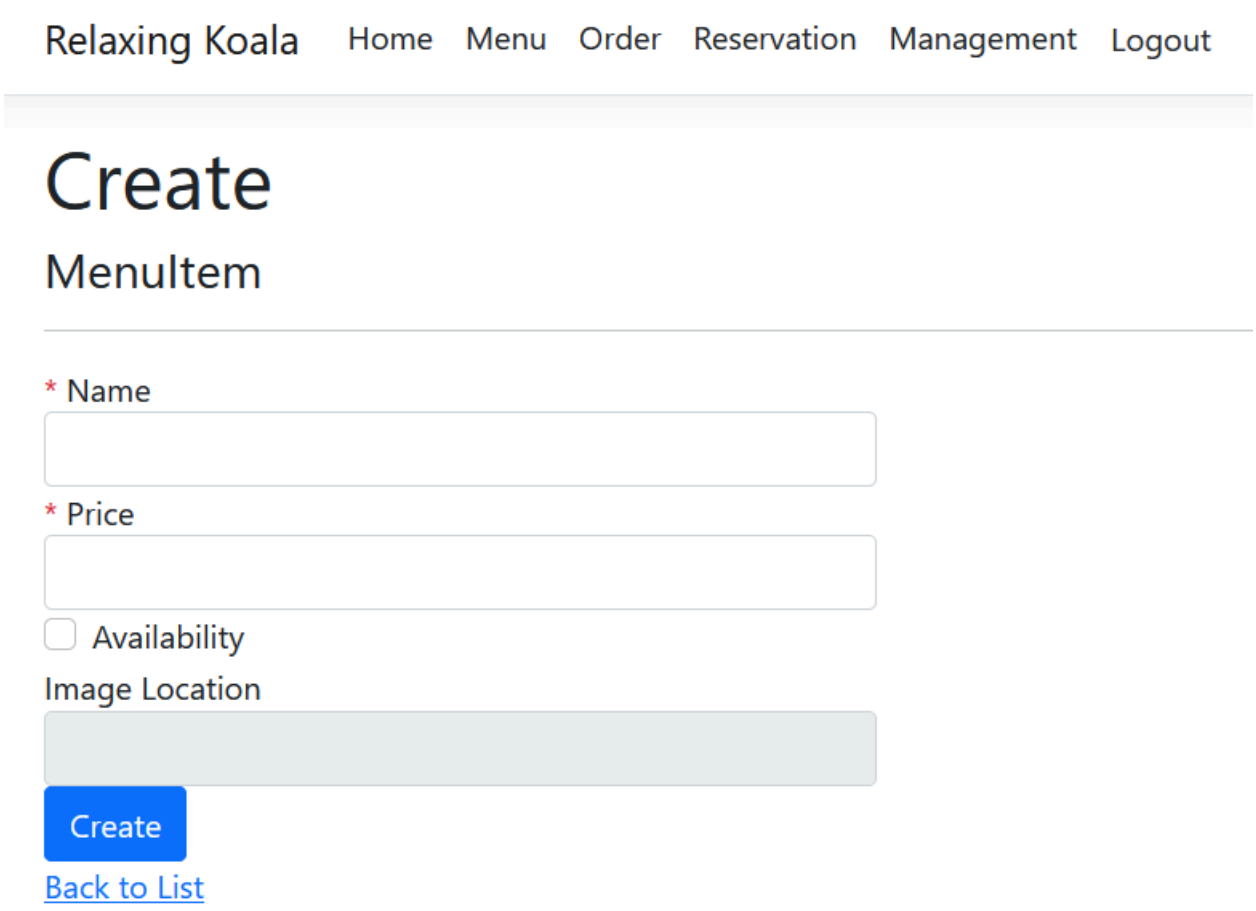


c. Reservation details in Management page

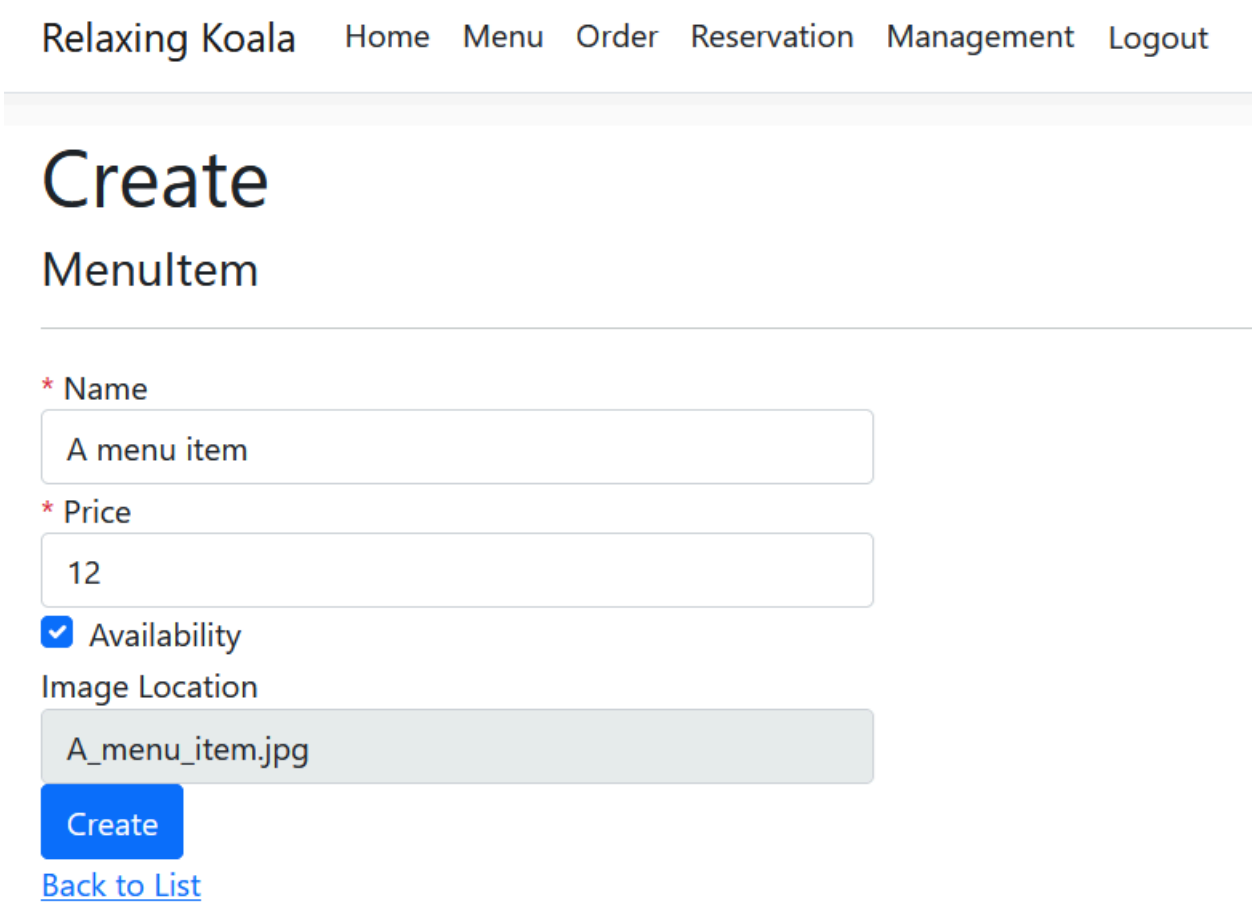


Scenario 5: Add new Menu item

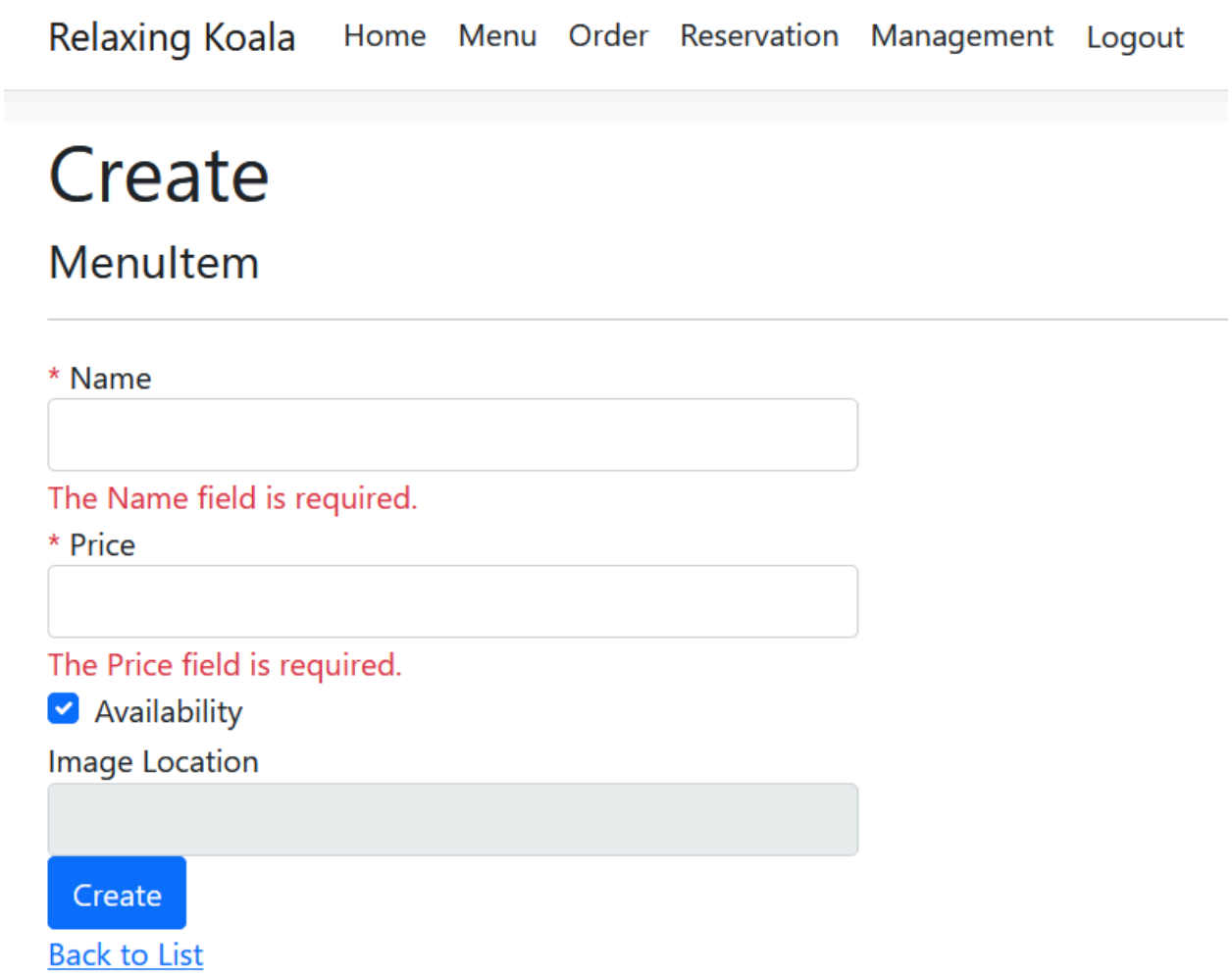
1. Starting state: MenuItemsManager's Create page is empty



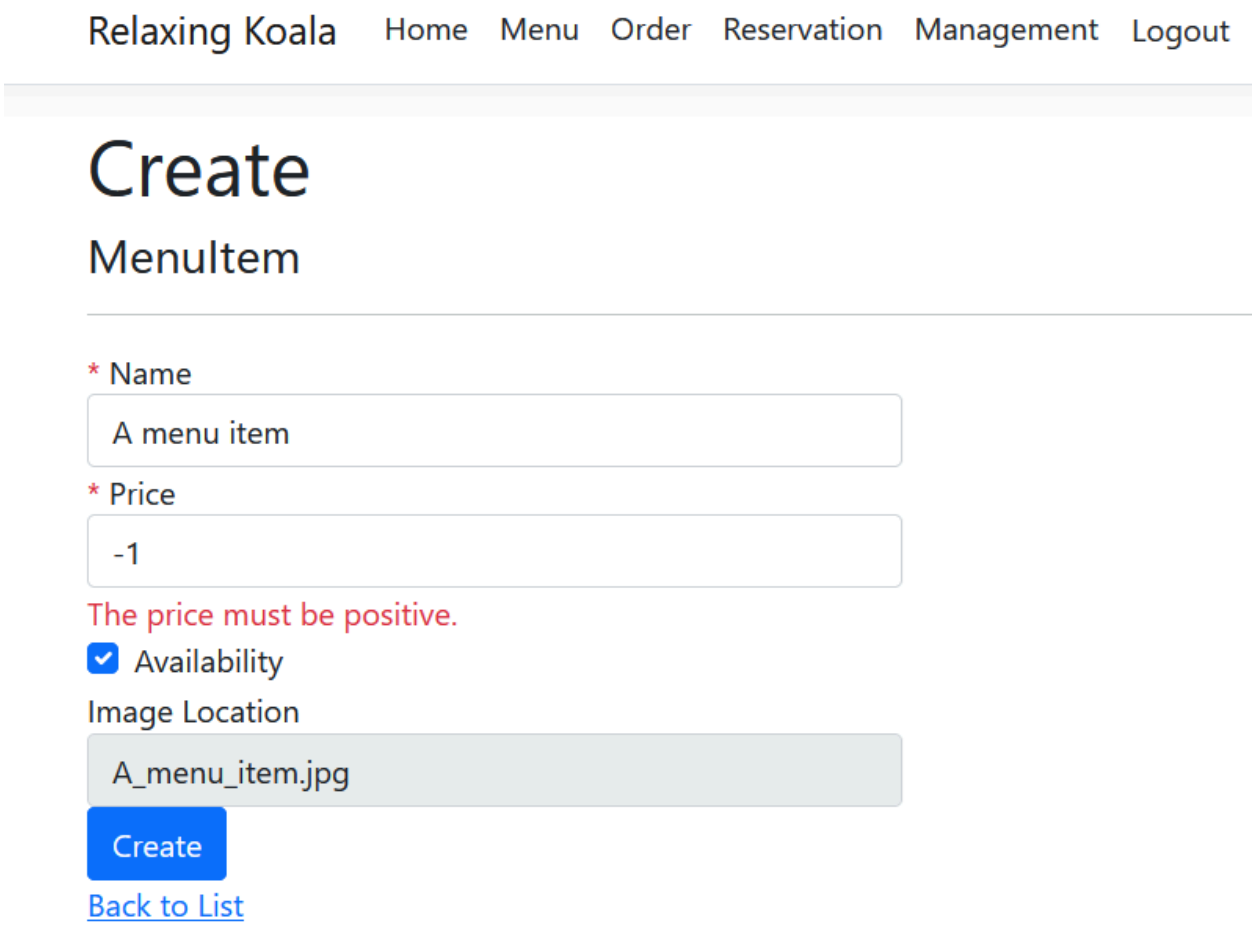
2. Inputting data: User enters the data of Menu Item.



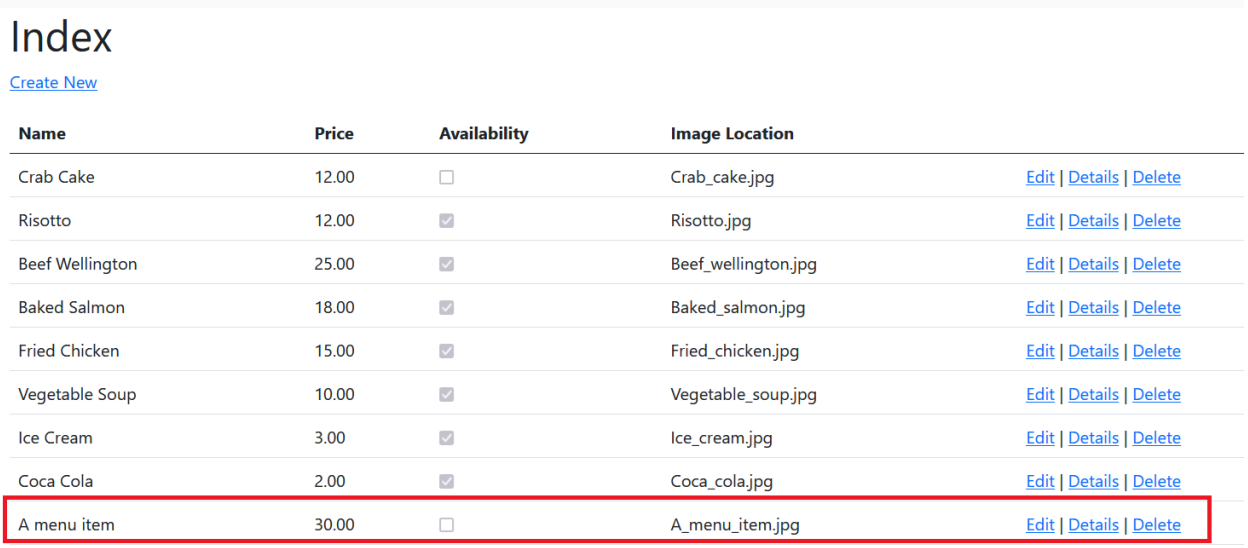
3. Input validation:
a. User leaves the required fields blank

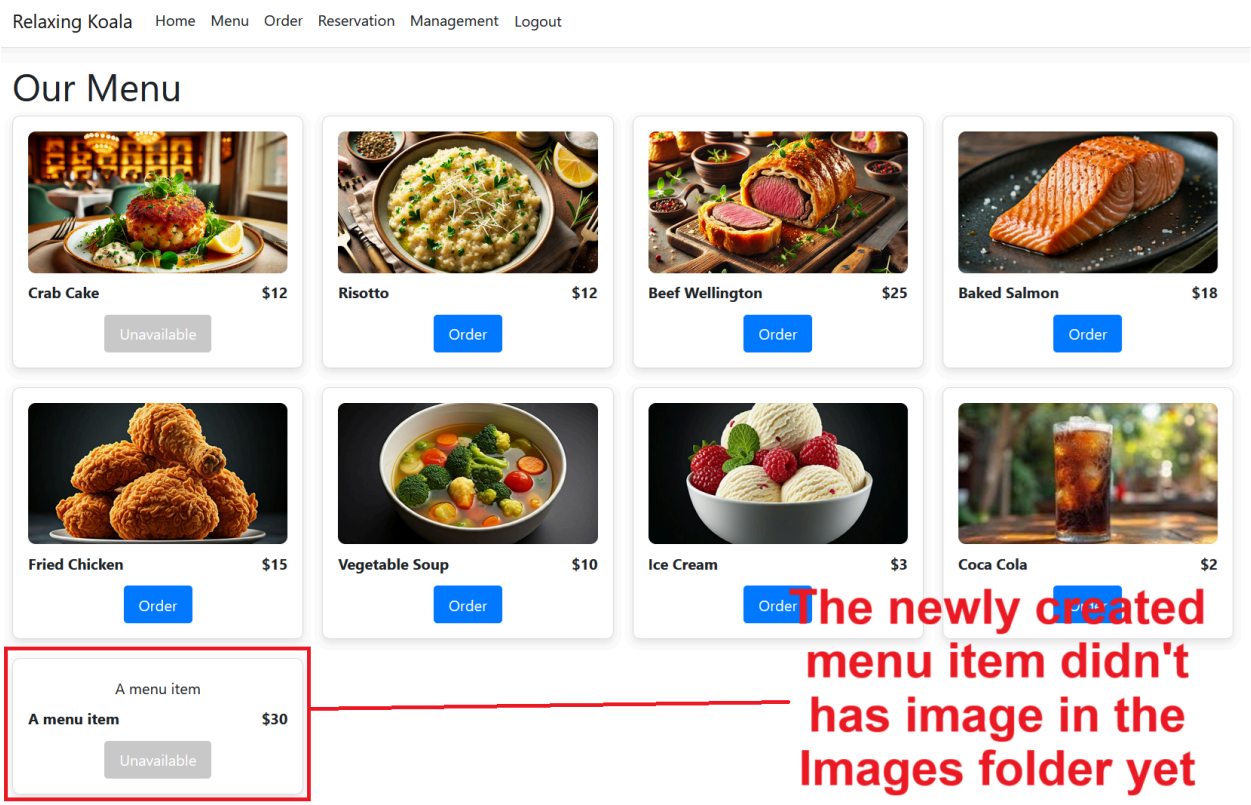


b. User enters negative price



4. Correct input: Once the input is confirmed as valid, the new menu items are added to the database. These items will then be visible on both the Management page, for staff reference, and the Menu page, which is accessible to customers.




Relaxing Koala
Home
Menu
Order
Reservation
Management
Logout
Our Menu
Crab Cake
$12
Unavailable
Risotto
$12
Order
Beef Wellington
$25
Order
Baked Salmon
$18
Order
Fried Chicken
$15
Order
Vegetable Soup
$10
Order
Ice Cream
$3
Order
Coca Cola
$2
A menu item
A menu item
$30
Unavailable
The newly created menu item didn't has image in the Images folder yet

References

1. Adesoji1 2023, *Choosing the right Git branching strategy for your organization*, Dev.to, viewed 24 November 2024, <https://dev.to/adesoji1/choosing-the-right-git-branching-strategy-for-your-organization-3o9p>.
2. Assignment 2 - Object Design | SWE30003_Assignment2_Group5_v3.pdf
3. Ayodeji, B 2019, *Writing good commit messages: a practical guide*, freeCodeCamp, viewed 24 November 2024, <https://www.freecodecamp.org/news/writing-good-commit-messages-a-practical-guide/>.
4. Cwalina, K & Abrams, B 2023, *Capitalization Conventions*, Microsoft Learn, viewed 23 November 2024, <https://learn.microsoft.com/en-us/dotnet/standard/design-guidelines/capitalization-conventions>
5. Krishnareddy, J. 2024, *Best practices for API error handling in .NET*, C# Corner, viewed 23 November 2024, <https://www.c-sharpcorner.com/article/best-practices-for-api-error-handling-in-net/>.
6. Manzoor, S 2024, *Good commit vs bad commit: best practices for Git*, Dev.to, viewed 24 November 2024, <https://dev.to/sheraz4194/good-commit-vs-bad-commit-best-practices-for-git-1plc>.
7. Martin, RC 2017, *Clean architecture: a craftsman's guide to software structure and design*, Prentice Hall.
8. Microsoft n.d., *HttpResponse.Redirect Method*, Microsoft Learn, viewed 23 November 2024, <https://learn.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http.httpresponse.redirect?view=aspnetcore-9.0>.
9. Mozilla Developer Network 2024, *Control flow and error handling*, MDN Web Docs, viewed 23 November 2024, <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling>.
10. Rahman, A 2023, *Understanding Model-View-Controller (MVC) architecture*, LinkedIn, viewed 24 November 2024, <https://www.linkedin.com/pulse/understanding-model-view-controller-mvc-architecture-askur-rahman>.
11. Twumasi, A.O.-T. 2024, *Git branching strategies for DevOps: best practices for collaboration*, Dev.to, 10 Jun, viewed 24 November 2024, <https://dev.to/angelotheman/git-branching-strategies-for-devops-best-practices-for-collaboration-35l8>.
12. Warren, G. 2024, *Best practices for exceptions*, Microsoft Learn, viewed 23 November 2024, <https://learn.microsoft.com/en-us/dotnet/standard/exceptions/best-practices-for-exceptions>.