EatMeOut

Tap it And Wrap It

SEEPP Solutions

Samy Wanas M00949455| Przemek Wasinski M00931085 |

Alex-Petrisor M00861387 | Emanuel Vochitoiu M00942608 |

Edwart Ianovici M00946886

Introduction

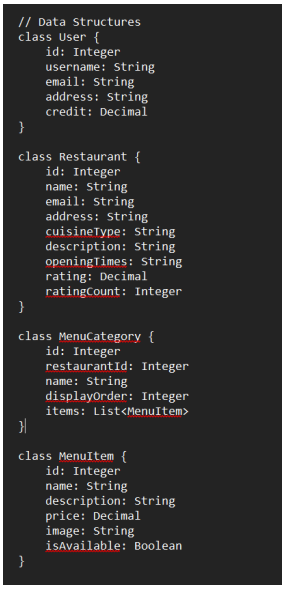
This report contains the development of “EatMeOut”, a restaurant takeaway system designed as a web application that enables customers to order food from partnered restaurants. After researching existing food delivery platforms, we identified an opportunity to develop a streamlined and user-friendly system tailored to both customers and restaurant owners. “EatMeOut” allows users to browse restaurant menus, select items, and place orders seamlessly. Once an order is placed, the restaurant receives and prepares it before sending it out for delivery via dedicated drivers. The system aims to provide a smooth ordering experience while ensuring efficient restaurant order management.

The report will first touch on the design of the project, discussing our justification of data structure(s) and algorithms, along with an analysis of algorithms which provide the key functionality. This will use pseudocode in order to demonstrate our design idea. We will then move onto testing whilst discussing the approach used and a table of test cases. At the end, we will provide a summary of the work done, limitations and critical reflection and how we would change the approach towards a future task. The report will then end with any references.

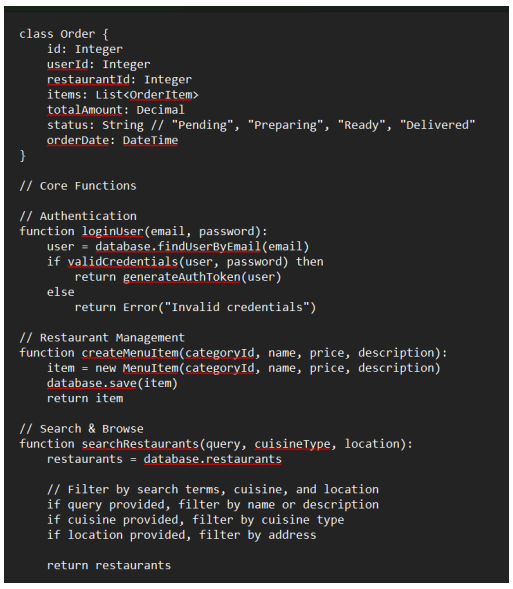
Design

EatMeOut's architecture is built on carefully selected data structures and algorithms to optimize performance and maintainability. The system employs relational database tables for the four primary entities (Restaurants, Users, Orders, and Menu Items), with hash-based indexing enabling O(1) lookups for frequent ID-based retrievals. Key algorithms include a Dynamic Rating System utilizing running averages for O(1) performance, a Smart Pricing Tier algorithm (O(n * m)) that leverages array-based categorization for restaurant price brackets, and an Intelligent Search function (O(R * (C * I * G))) implemented with multi-level hash maps to efficiently locate restaurants across various criteria. For transaction handling, the Order Processing system uses queue data structures for asynchronous processing with linear O(i) complexity, while Revenue Management implements priority queues for the progressive withdrawal system, maintaining O(n) complexity.

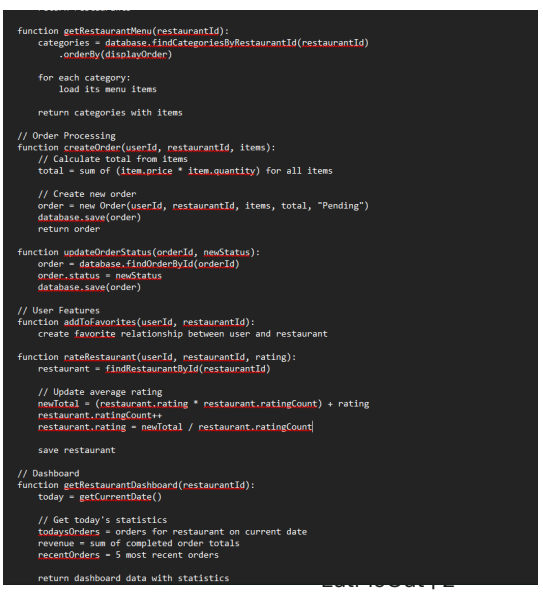
The implementation balances security and performance through strategic data structure choices: BCrypt's adaptive hashing algorithms for password security, JWT token storage in hash tables for O(1) authentication verification, and transaction logs in append-only data structures ensuring data integrity. While the current structure prioritizes maintainability, future optimizations could include inverted indexes (trie-based data structures) for search functionality, LRU cache implementations for frequently accessed data, cursor-based pagination utilizing linked lists for large datasets, and B-tree indexes for range-based queries. These data structures and algorithms provide a solid foundation for the current scale while offering clear paths for optimization as the platform grows, particularly for the computationally intensive search functionality that could benefit from specialized index structures.



```
// Data Structures
class User {
    id: Integer
    username: String
    email: String
    address: String
    credit: Decimal
}

class Restaurant {
    id: Integer
    name: String
    email: String
    address: String
    cuisineType: String
    description: String
    openingTimes: String
    rating: Decimal
    ratingCount: Integer
}

class MenuCategory {
    id: Integer
    restaurantId: Integer
    name: String
    displayOrder: Integer
    items: List<MenuItem>
}

class MenuItem {
    id: Integer
    name: String
    description: String
    price: Decimal
    image: String
    isAvailable: Boolean
}
```



```
class Order {
    id: Integer
    userId: Integer
    restaurantId: Integer
    items: List<OrderItem>
    totalAmount: Decimal
    status: String // "Pending", "Preparing", "Ready", "Delivered"
    orderDate: DateTime
}

// Core Functions

// Authentication
function loginUser(email, password):
    user = database.findUserByEmail(email)
    if validCredentials(user, password) then
        return generateAuthToken(user)
    else
        return Error("Invalid credentials")

// Restaurant Management
function createMenuItem(categoryId, name, price, description):
    item = new MenuItem(categoryId, name, price, description)
    database.save(item)
    return item

// Search & Browse
function searchRestaurants(query, cuisineType, location):
    restaurants = database.restaurants

    // Filter by search terms, cuisine, and location
    if query provided, filter by name or description
    if cuisine provided, filter by cuisine type
    if location provided, filter by address

    return restaurants
```



```
function getRestaurantMenu(restaurantId):
    categories = database.findCategoriesByRestaurantId(restaurantId)
        .orderBy(displayOrder)

    for each category:
        load its menu items

    return categories with items

// Order Processing
function createOrder(userId, restaurantId, items):
    // Calculate total from items
    total = sum of (item.price * item.quantity) for all items

    // Create new order
    order = new Order(userId, restaurantId, items, total, "Pending")
    database.save(order)
    return order

function updateOrderStatus(orderId, newStatus):
    order = database.findOrderById(orderId)
    order.status = newStatus
    database.save(order)

// User Features
function addToFavorites(userId, restaurantId):
    create favorite relationship between user and restaurant

function rateRestaurant(userId, restaurantId, rating):
    restaurant = findRestaurantById(restaurantId)

    // Update average rating
    newTotal = (restaurant.rating * restaurant.ratingCount) + rating
    restaurant.ratingCount++
    restaurant.rating = newTotal / restaurant.ratingCount

    save restaurant

// Dashboard
function getRestaurantDashboard(restaurantId):
    today = getCurrentDate()

    // Get today's statistics
    todaysOrders = orders for restaurant on current date
    revenue = sum of completed order totals
    recentOrders = 5 most recent orders

    return dashboard data with statistics
```

Testing

This section outlines the structured testing process undertaken throughout the development of *EatMeOut*, as well as the planned unit test implementation. Testing was carried out across all key phases of the project to ensure functional accuracy, data integrity, user experience consistency, and system stability. In addition to functional and regression testing, core logic components will also be validated using automated unit tests as per project requirements.

Testing Approach

We adopted a black-box testing approach with a focus on verifying:

-Functional correctness across both user and restaurant roles
-Security, including authentication, session handling, and protected routes
-Data integrity across all CRUD operations
-UI and form responsiveness, including input validation and error feedback
-Business logic accuracy, such as pricing tier classification and dynamic ratings

Testing was performed using manual browser-based interactions, database validation, and API inspection.
Later-stage testing focused on integrated feature flow and regression.

To meet the coursework requirement for automated testing, we will additionally implement unit tests using MSTest, a standard .NET testing framework. These tests will target logic-driven components such as:

-Dynamic Rating System – ensuring constant-time running average calculations
-Smart Pricing Tier Algorithm – verifying correct classification based on item prices
-Search Filtering Logic – testing query matching against restaurant names, cuisines, and ingredients
-Credit Deduction and Balance Management – confirming accurate arithmetic and edge-case handling

Testing Stages and Progress

Testing was structured into five clear phases, aligned with feature rollout:

1.Authentication & Registration Phase

-User and restaurant registration/login

-JWT token validation and password hashing using BCrypt

-Route protection and session control

2.Menu & Dashboard Management

-Category and item creation, update, deletion

-Restaurant metrics validation (revenue, ratings, order counts)

-Pricing tier assignment and real-time menu updates

3.User Feature Rollout

-Profile editing, credit top-up, and favorites system

-Multi-criteria search filtering based on intelligent data structures

-Security validation on user-specific endpoints

4.Order Processing

-End-to-end order flow: cart → placement → status tracking → history

-Credit deduction, order summary generation, and feedback prompts

-Restaurant dashboard and withdrawal system

5.Final Regression & Polishing

-Re-testing resolved issues (e.g., empty cart bug, login validation)

-Ul responsiveness, visual feedback on buttons, layout alignment

-Graceful error handling (e.g., database failure, missing input)

-User/restaurant role separation enforcement

Test Case Evidence

A detailed test case table summarising final system testing is included in this report (see below). It contains 15 well-defined test cases covering the full functionality of the application.

In addition, five in-depth test case tables were created for each major development phase. These are included in the submission folder. Each test case includes feature, objective, test steps, expected result, and outcome, demonstrating full traceability of QA across the development lifecycle.

Tools and Testing Environment

-Manual testing performed via browser simulations (Chrome DevTools)

-API and request monitoring through browser network tools

-SQL database inspection for back-end validation

-MSTest framework will be used for unit testing logic components in .NET

-GitHub version control used for issue tracking and commit-based testing cycles

Conclusion

Testing was applied rigorously and continuously across all development stages of *EatMeOut*. Every key feature was validated through a combination of structured functional testing and targeted logic evaluation. The inclusion of automated unit testing using MSTest will further ensure that critical components behave as expected in isolation, satisfying the quality and alignment standards outlined in the coursework brief.

These rest of the unit tests will be included in the Project Management `/testing` folder.

ID	Category	Feature	Objective	Test Steps / Data Used	Expected Results	Status
TC01	Authentication	User Login	Verify valid credentials allow access	Login with: edwart@test.com / Valid123	User is authenticated and redirected to homepage	☑
TC02	Authentication	User Login	Confirm invalid login is blocked and	Login with: edwart@test.com / WrongPwd	Login denied; error message	☑
TC03	Navigation & UI	Menu Loading	Ensure restaurant menus load dynamically and accurately	Select "PizzaHut" from homepage	Menu items displayed with names, prices, images, and descriptions	☑
TC04	Authentication	Order Creation	Validate the full order process from selection to confirmation	Select 3 items → Go to cart → Place order	Order recorded in DB; confirmation shown; restaurant notified	☑
TC05	Input Validation	Checkout Field Validation	Prevent orders with missing address	Leave address field blank during checkou	Submission blocked; form validation warning appears	☑
TC06	Edge Case	Special Character Handling	Ensure input fields handle invalid or special characters	Input: Name: <Alex> Address: @#$%^&	Inputs sanitized or rejected;	☑
TC07	Data Handling	SQL Write & Read	Confirm order data is correctly stored and retrieved from DB	Place test order → Manually inspect DB or use order history feature	Order data correct and accessible; relational links preserved	☑
TC08	Session Management	Logout	Verify session ends and protected pages are no longer accessible	Log in → Logout → Try accessing /order-history page	Access denied; redirect to login; JWT invalidated	☑
TC09	UX/History	Order History	Validate retrieval of previously placed orders	Place multiple test orders → Access Order History	Accurate list shown; most recent first; each includes timestamp, items, and cost	☑
TC10	UI Consistency	Desktop Layout & Scaling	Ensure the interface renders correctly across various desktop screen sizes	Resize browser window between 1024x768 and 1920x1080	Layout adjusts smoothly; all elements remain visible and properly aligned	☑
TC11	Error Handling	Database Connection Loss	Ensure graceful handling of DB failure during order	Search: “Italian”, “Pizza”, or “Burger”	Relevant results returned based on name, cuisine, and ingredients	☑
TC12	Performance	Large Menu Load	Test speed and stability when loading a large menu	Submit ratings: 3, 5, 4 → Refresh listing	Running average updated; no full recalculation needed (O(1))	☑
TC13	Regression	Validation Fix Re-Test	Confirm recent bug fix (missing address allowed orders) is now resolved	Check menus with £4, £9, £15 items	Restaurant assigned to £, ££, or £££ tier correctly based on average menu item price	☑
TC14	System Stability	Multiple Rapid Orders	Submit multiple orders back-to-back to test order queue integrity	Place 5 orders in 30 seconds	All orders processed correctly; no duplication or data	☑
TC15	UI Feedback	Button Response	Ensure interactive elements provide visual feedback	Disable DB then place order	Error message shown; system stable; no crash	☑

Conclusion

EatMeOut has delivered on its promise of a restaurant takeaway system that tackles the issues that a lot of restaurants and customers face; a simple solution that does not compromise on ease of use, efficiency, and customer satisfaction. The project has been built with ASP.NET Core and Entity Framework core, which provides secure user authentication, efficient data management as well as scalable API endpoints. The frontend offers an easy and intuitive user interface meeting every criteria of what a modern restaurant takeaway system should be. Restaurants are able to view their statistics and see what is working for them and what is not. EatMeOut also makes the user side easy and fun to use.

Time management and spreading the development work out properly between the group members has kept this project back from reaching its full potential. Over the course of the project, the team has mismanaged its time to a small extent and therefore we have not been able to implement some features that we would have liked to implement. Additionally, some team members have done more work or less work depending on the task that was handed to them. This has meant that not everyone has been working an equal amount. Another limitation was not knowing the programming language and SQL that well yet. This meant the team had to start from 0 on the language and SQL, and build the project from there. Everybody did work hard to understand it and give their all to make it work, but not being familiar with the language and SQL from the beginning (.NET as well), meant that a lot of time had to be spent learning them instead of focusing on completing the project.

In the future, the team has discussed tighter time constraints and more open communication. This will lead to a better work environment as people are more willing to work on what they need to do. Through tighter time constraints, it will not aim to make anybody work faster or slower, but it will give a gap on how long someone should work on a feature before it is then split up and then given to somebody else to work on. This means that the work will be more evenly split between people and everyone has something to do. Open communication also means that if we are finding something difficult, or can't fully work on a feature and get it up and running, someone else can take over and make sure it does. These two key factors are what makes a good team a great team, and will definitely help with a future approach on the matter.

We have a large list of ideas to incorporate and are excited for what the future holds.

References