Low-Level Design Report

1. Introduction

Maintaining a healthy diet has become increasingly challenging in today's fast-paced world due to busy schedules and limited time for meal planning. Research indicates that approximately 69% of users abandon health and nutrition apps within the first 90 days, primarily due to time-consuming manual data entry, limited personalization, and lack of sustained motivation [12]. Most nutrition apps rely on manual food logging and lack emotional support features critical for long-term engagement.

NutriGame is a mobile application designed to address these challenges by making healthy eating more accessible and engaging. The application helps users monitor their daily nutrition and calculate calorie intake based on individual goals. NutriGame leverages AI-powered food recognition to automatically estimate calorie values and nutrition information from photos, significantly reducing meal logging time [6]. An AI-based chatbot provides psychological encouragement and motivational support throughout the user's wellness journey. In addition to AI features, NutriGame encourages sustained engagement through social features and gamification [11]. Users can connect with friends, start or join daily challenges, share healthy recipes, and earn badges for consistent logging through the streak system.

This report describes the low-level architecture and design of the NutriGame system. Report comprises Object design trade-offs, Engineering standards, Packages and Class interfaces sections. Finally, the report is concluded with the class diagrams and detailed explanations of software components.

1.1 Object design trade-offs

1.1.1 Efficiency vs. Precision

In NutriGame, we aim to balance the speed of data entry with the granularity of nutritional tracking. While manual logging remains a core feature for users requiring exact measurements and maximum precision, relying on it exclusively can create friction that hinders daily engagement. Therefore, we prioritized efficiency by implementing AI-powered food recognition as the primary logging method [6]. We accept a trade-off where the AI provides rapid, estimated nutritional data to maximize user retention , while the system retains the capability for manual entry when users prefer granular control over their diet logs.

1.1.2 User Experience vs. Implementation Simplicity

In NutriGame, we prioritized a fluid User Experience (UX) over the simplicity of the code structure. Since our application relies on external AI services (Google Gemini) which may take several seconds to process an image, a simple, linear code structure would cause the screen to freeze while waiting for a response. To prevent this, we implemented asynchronous background processing [5]. While this requires slightly more complex logic to manage "loading" states and background tasks, it ensures the app remains responsive and usable even while heavy AI analysis is happening in the background.

1.1.3 Security vs. Contextual Functionality

While maximizing the context provided to an AI model typically improves the quality of its responses, we prioritized security and user privacy over unrestricted functionality. We implemented a strict privacy strategy where sensitive data, such as EXIF metadata from images and personal identifiers in chat logs, is stripped or filtered before being sent to external AI services [18]. Although this limits the raw data available to the AI potentially constraining its ability to "know" the user fully—it is a necessary trade-off to ensure compliance with privacy standards and protect user anonymity.

1.1.4 Portability vs. Native Performance

NutriGame aims to be accessible to the widest possible audience on both iOS and Android platforms. To achieve this, we prioritized portability and development efficiency over the raw performance optimization of native code. By utilizing the React Native framework [5], we maintain a single codebase that runs on both operating systems. While native applications (Swift/Kotlin) might offer marginal performance gains, the trade-off allows us to deliver consistent features to all users simultaneously and maintain a modular, unified codebase.

1.2 Interface documentation guidelines

In this report; all classes, attributes and methods are named in the camel case format. Class names start with a capital letter, while others start with lowercase. Class interface descriptions are given in the following format:

class ClassName
Explanation of the class
Attributes
typeOfAttribute nameOfAttribute
Methods

returnType methodName(parameters) Method explanation if necessary

1.3 Engineering standards (e.g., UML and IEEE)

In this report, we have adhered to UML guidelines for defining class interfaces, diagrams, scenarios, and subsystem compositions [3]. We selected UML due to its widespread adoption in the software industry, its ease of application, and its alignment with the methodologies taught in the BIL481 Software Engineering course at TOBB ETU [1]. Furthermore, all citations within this document strictly follow IEEE standards [4], in accordance with the course requirements.

1.4 Error Handling and Edge Cases

NutriGame considers basic error handling and edge cases to keep the application stable and provide a better user experience.

1.4.1. AI Service and Image Issues

- A user might upload an image that does not contain food. The system prompt instructs the AI to recognize this and return a specific response. The app handles this by asking the user to take a new photo or switch to manual log entry.
- The external Gemini API might be slow or unavailable [6]. To prevent the app from freezing, the backend limits the waiting time. If the service fails, it returns a standard error message, and the client informs the user to try manual food logging instead.

1.4.2. Time and Date Tracking Control

- A user might log yesterday's meal today. To prevent this from incorrectly updating today's challenges, the MealLog and WaterLog use the user's selectedDate instead of the current server time.

1.4.3. Network Connection Issues

- If the network drops while a user is writing a recipe (CreatePost) or waiting for an AI response the user is shown an error message.

1.4.4. Chatbot Scope

- A user might ask the chatbot about unrelated topics like coding or politics. The app uses a system prompt to limit the AI's context to diet and motivation. For unrelated questions, the chatbot will provide a standard response guiding the user back to health topics.

1.4.5. File Upload Limits and Formats

- A user might try to upload a very high-resolution photo (e.g., a 20MB raw file) or an unsupported file type (like a PDF or video) for food recognition or a social post. To save server storage and

processing power, the backend checks the file at the entry point. It strictly limits file sizes to 5MB and only accepts JPEG and PNG formats. If a file exceeds these limits or has the wrong format, the server rejects it immediately, and the mobile client shows a warning message asking the user to choose a smaller or correct image.

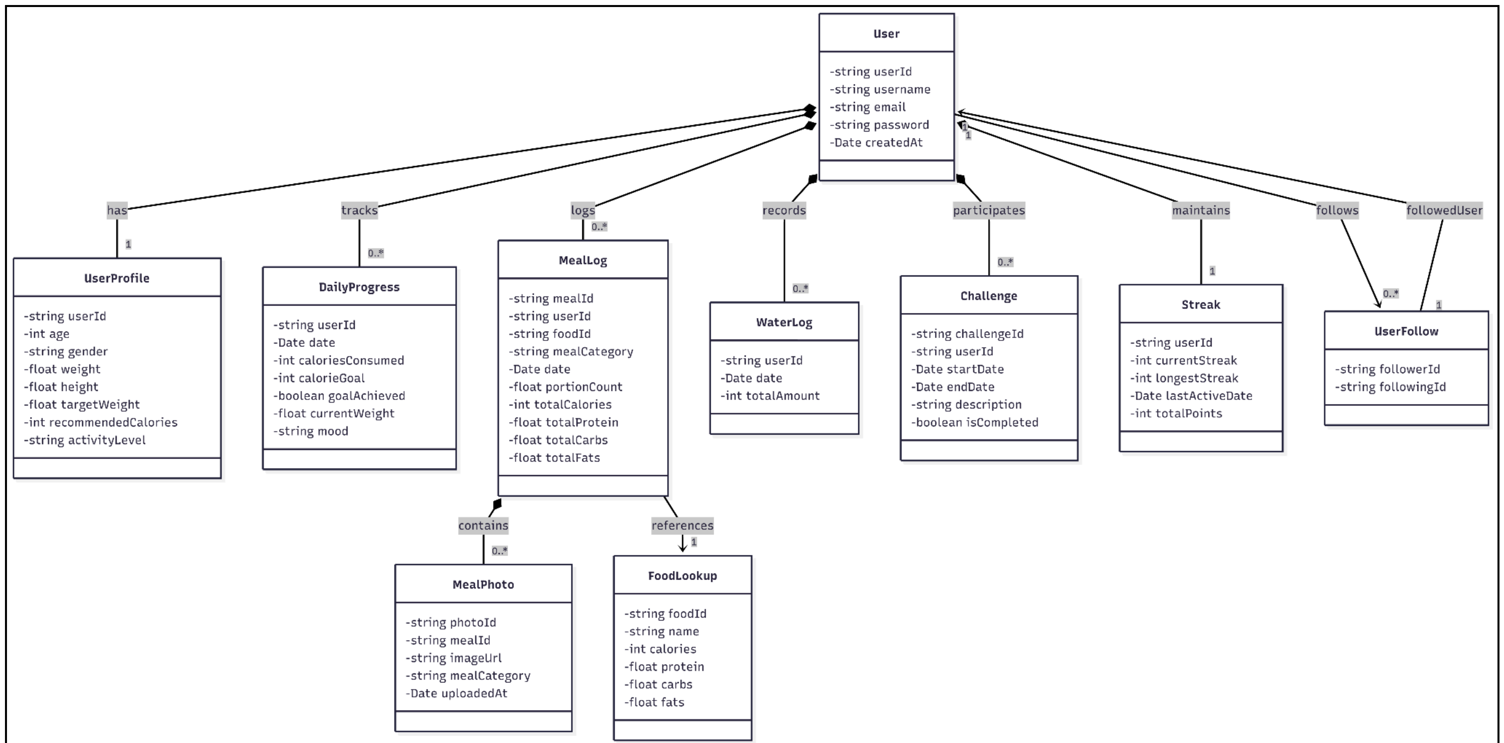
1.5 Definitions, acronyms, and abbreviations

AI**:** Artificial Intelligence
API**:** Application Programming Interface
BMI**:** Body Mass Index
JWT: JSON Web Token[8]
EXIF: Exchangeable Image File Format

2. Packages

2.1 Server

2.1.1 Model



Model Package

User: This is the central authenticated entity. It stores core credentials like username, email, and password, and serves as the parent for all other user-related health and social records.

UserProfile: This class stores personal physiological metrics, such as age, gender, weight, height, and activity level. It is linked to the User via a one-to-one relationship to calculate personalized recommended calories and target weight.

DailyProgress: Used for historical progress analysis, this class tracks health metrics for a specific date. It records calories consumed, the calorie goal, whether the goal was achieved, and the user's mood or weight on that day.

MealLog: The primary ledger for all dietary intake. It stores the meal category (e.g., Breakfast), date, and portion count, while calculating total macronutrients (protein, carbs, fats) by referencing the FoodLookup table.

MealPhoto: Linked directly to a meal log, this class stores the imageUrl and upload date. It supports the AI-powered food recognition workflow by maintaining a visual history of the user's meals.

FoodLookup: Acts as a reference database for food items. It contains specific nutritional data like calories and macronutrients per standard portion to allow accurate logging in the MealLog.

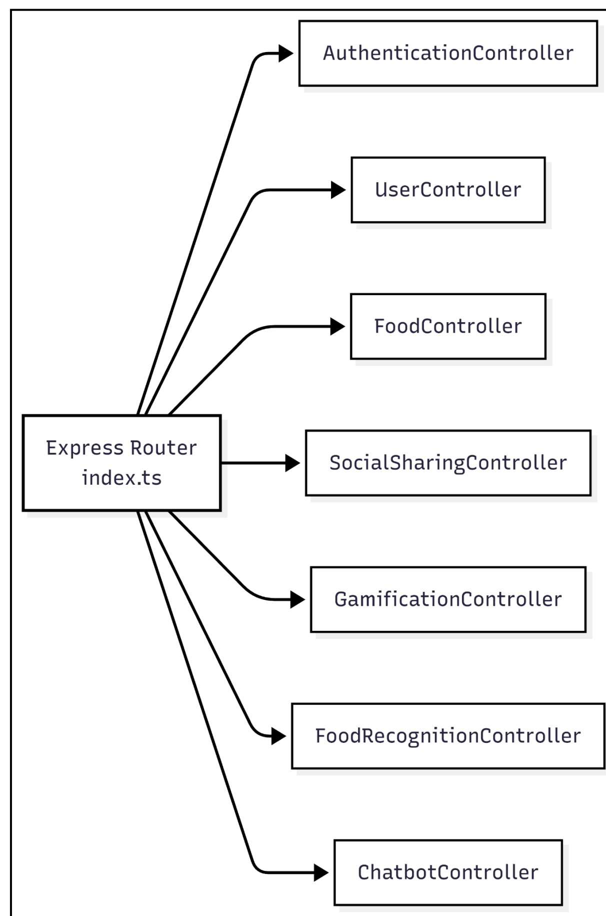
WaterLog: Designed separately from food logs to simplify hydration tracking. It records the total amount of water consumed by a user on a specific date.

Challenge: Represents time-bounded events a user participates in. It stores the start/end dates, a description of the challenge, and a boolean isCompleted status to track gamification progress[11].

Streak: Tracks long-term user engagement[9]. It maintains metrics such as the current streak, the longest streak ever achieved, last active date, and total points earned through consistent logging.

UserFollow: This class represents the social graph of the application. It facilitates many-to-many relationships by mapping follower IDs to following IDs, allowing users can see each other's updates in the social feed.

2.1.2 Controller



Controller Package

AuthenticationController: This controller class is responsible for managing user authentication processes. It handles the registration of new users and the login of existing users.

UserController: This controller class manages user-specific operations. It handles requests related to the authenticated user's profile, including retrieving profile details, updating personal information, and executing account deletion.

FoodController: This controller class handles operations related to food and nutrition tracking. It manages searching for food items, logging meals and water intake, and retrieving daily consumption summaries for the user.

Gamification Controller: This controller is responsible for tracking challenge progress, streaks, and completion status based on user activity logs. It evaluates whether challenge goals are met, updates progress values, assigns rewards upon completion, and triggers notifications for relevant participants.

SocialSharingController: This controller handles all social interaction features exposed to the client, including following users, sharing recipes and posts, creating and managing challenges, and retrieving social and challenge feeds.

FoodRecognitionController: This class handles all food image analysis requests. It validates uploaded images, preprocesses them (EXIF removal, resize), and delegates to FoodRecognitionService. Implements circuit breaker pattern for AI service failures and rate limiting to prevent abuse [20].

ChatbotController: This class manages chatbot interactions with users. It handles both regular and streaming responses from Gemini API. Validates message length, enforces rate limits (10 messages/minute), and caches conversation history for context-aware responses.

2.2 Client

Login: This class manages the user authentication process. It implements a smart input logic that automatically distinguishes between an email and a username based on the input format.

SignUp: This class initiates the multi-step registration process. It serves as the primary data collection interface for user credentials (username, email, password).

SignUpEnterData: This class manages the second phase of registration.

CreateAvatar: This class concludes the user registration workflow. It receives the cumulative finalData object from the previous steps, which contains the user's credentials and physiological profile. It presents a grid of predefined avatar images for selection.

MainPage: This class represents the central dashboard of the NutriGame application. It aggregates and visualizes the user's daily nutritional progress using animated charts (CalorieCircle). It implements a date-based navigation system, allowing users to switch between past and present logs. Furthermore, it serves as

the primary navigation router, providing entry points to meal logging (Breakfast, Lunch, etc.), water tracking, and AI-powered features (ScanFood, Chatbot). It ensures data freshness by re-fetching stats every time the screen gains focus.

AddMeal: This class facilitates the manual entry of food items into the user's daily log.

AddWater: This class manages the user's hydration tracking.

ScanFood: This class serves as the hardware interface for the "AI-Assisted Logging Flow".

Chatbot: This class implements the conversational interface for the "NutriCoach" AI assistant.

Menu: This class serves as the central hub for user profile management and application settings.

ProfileSettingMenu: This class serves as the configuration panel for critical user account settings.

EditProfile: This class provides a comprehensive form interface for users to update their personal information and physical metrics.

3. Class Interfaces

3.1 Server

3.1.1 Model

class User
This class represents the central authenticated user entity of the system. It stores login credentials and basic account information, and it acts as the parent entity for all user-related health, social, and gamification records.
Attributes
private string userId private string username private string email private string password private Date createdAt
Methods

Getter and setter methods.

class UserProfile
This class stores the user's personal and physiological metrics used for personalized calorie recommendations. It is linked to User via userId in a one-to-one relationship.
Attributes
private string userId private int age private string gender private float weight private float height private float targetWeight private int recommendedCalories private string activityLevel
Methods
Getter and setter methods.

class DailyProgress
This class stores daily health tracking metrics and summarizes whether the user achieved their goals for a specific date. It enables historical progress analysis through a one-to-many relationship with User.
Attributes
private string userId private Date date private int caloriesConsumed private int calorieGoal private boolean goalAchieved private float currentWeight private string mood
Methods
Getter and setter methods.

class MealLog
This class is the main ledger for dietary intake records. It stores the meal category, date, portion count, and macronutrient totals, and it references FoodLookup via foodId. It also maintains relationships with the user and meal photos.
Attributes
private string mealId private string userId private string foodId private string mealCategory private Date date private float portionCount private int totalCalories private float totalProtein private float totalCarbs private float totalFats
Methods
Getter and setter methods.

class MealPhoto
This class stores photo records linked to a meal log entry. It is used both for maintaining visual meal history and for supporting AI-powered food recognition workflows.
Attributes
private string photoId private string mealId private string imageUrl private string mealCategory private Date uploadedAt
Methods
Getter and setter methods.

class WaterLog
This class stores the user’s daily hydration information. It is designed separately from meal logs to simplify hydration tracking and daily pattern analysis.

class Challenge
This class represents a time-bounded challenge record for user participation. It stores the challenge duration, description, and completion state, and supports multiple active challenges per user.
Attributes
private string challengeId private string userId private Date startDate private Date endDate private string description private boolean isCompleted
Methods
Getter and setter methods.

class Streak
This class stores streak-related engagement metrics and total points. It uses the user’s activity to maintain consecutive-day counts and historical best streak values.
Attributes
private string userId private int currentStreak private int longestStreak private Date lastActiveDate private int totalPoints
Methods

Getter and setter methods.

class UserFollow
This class represents the social graph using a self-referencing follower/following relationship between users. It creates a many-to-many relationship for social connections.
Attributes
private string followerId private string followingId
Methods
Getter and setter methods.

3.1.2 Controller

Authentication Controller

class AuthenticationController
This controller class is responsible for managing user authentication processes. It handles the registration of new users and the login of existing users by validating input data, interacting with the user model, and generating JWT tokens for session management[8].
Attributes
None: This class is stateless and relies on the userModel for data persistence.
Methods
register(req: Request, res: Response, next: NextFunction): Promise<Response> : Validates user inputs and creates a new user in the database. login(req: Request, res: Response, next: NextFunction): Promise<Response> : Authenticates credentials and issues a JWT token for session management.

User Controller

class UserController
This controller class manages user-specific operations. It handles requests related to the authenticated user's profile, including retrieving profile details, updating personal information, and executing account deletion.
Attributes

None : This class is stateless and relies on the userModel for data persistence.
Methods
updateUserProfile(req: Request, res: Response, next: NextFunction): Promise<Response> : Updates the authenticated user's profile attributes getUserProfile(req: Request, res: Response, next: NextFunction): Promise<Response> : Retrieves the authenticated user's complete profile information deleteAccount(req: Request, res: Response, next: NextFunction): Promise<Response> : Permanently removes the authenticated user's account record from the database.

Food Controller

class FoodController
This controller class handles operations related to food and nutrition tracking. It manages searching for food items, logging meals and water intake, and retrieving daily consumption summaries for the user.
Attributes
None: This class is stateless and relies on the foodModel for data persistence.
Methods
searchFood(req: Request, res: Response, next: NextFunction): Promise<Response> : Searches the database for food items matching a provided name and returns a list of results. addToMeal(req: Request, res: Response, next: NextFunction): Promise<Response> : Adds a selected food item to the user's daily meal log, calculating total nutritional values (calories, protein, fat, carbs) based on the portion count. deleteFromMeal(req: Request, res: Response, next: NextFunction): Promise<Response> : Removes a specific food entry from the user's meal log. getMealLog(req: Request, res: Response, next: NextFunction): Promise<Response> : Retrieves all food entries for a specific date and meal category (e.g., Breakfast, Lunch, Dinner, Snack). getMealTotal(req: Request, res: Response, next: NextFunction): Promise<Response> : Calculates and returns the total nutritional intake (calories, macros) for a specific date and meal category (e.g., Breakfast, Lunch, Dinner, Snack). addToWater(req: Request, res: Response, next: NextFunction): Promise<Response> : Logs a specific amount of water consumption for the user on a given date.

deleteFromWater(req: Request, res: Response, next: NextFunction): Promise<Response> : Removes a specific water entry from the daily log and updates the daily total. getWaterTotal(req: Request, res: Response, next: NextFunction): Promise<Response> : Retrieves the total amount of water consumed by the user for a specific date.

Gamification Controller

class GamificationController
This controller is responsible for tracking challenge progress, streaks, and completion status based on user activity logs[11]. It evaluates whether challenge goals are met, updates progress values, assigns rewards upon completion, and triggers notifications for relevant participants. The controller acts as a coordination layer by delegating core business logic to ChallengeTrackingService, RewardService, and NotificationService.
Attributes
private ChallengeTrackingService challengeTrackingService private RewardService rewardService private NotificationService notificationService private AuthService authService private Logger logger
Methods
public async handleProgressEvent(req: Request, res: Response): Promise<void> public async updateStreak(req: Request, res: Response): Promise<void> public async evaluateChallengeCompletion(req: Request, res: Response): Promise<void> public async completeChallenge(req: Request, res: Response): Promise<void> public async getChallengeProgress(req: Request, res: Response): Promise<void>

Social Media Controller

class SocialSharingController
This controller handles all social interaction features exposed to the client, including following users, sharing recipes and posts, creating and managing challenges, and retrieving social and challenge feeds. It validates incoming requests, enforces privacy and visibility rules, and delegates business logic to SocialService, PostService, and ChallengeService while remaining a thin coordination layer.
Attributes
private SocialService socialService private PostService postService private ChallengeService challengeService private AuthService authService

private RateLimiter rateLimiter private Logger logger
Methods
public async followUser(req: Request, res: Response): Promise<void> public async unfollowUser(req: Request, res: Response): Promise<void> public async createPost(req: Request, res: Response): Promise<void> public async getFeed(req: Request, res: Response): Promise<void> public async createChallenge(req: Request, res: Response): Promise<void> public async respondToChallengeInvite(req: Request, res: Response): Promise<void> public async getChallenges(req: Request, res: Response): Promise<void> public async getUserPosts(req, res): Promise<void> public async deletePost(req, res): Promise<void>

Food Recognition Controller

class FoodRecognitionController
This class handles all food image analysis requests. It validates uploaded images, preprocesses them (EXIF removal, resize), and delegates to FoodRecognitionService. Implements circuit breaker pattern for AI service failures and rate limiting to prevent abuse[10, 20].
Attributes
private FoodRecognitionService foodRecognitionService private ImagePreprocessor imagePreprocessor private CircuitBreaker circuitBreaker private RateLimiter rateLimiter
Methods
public async analyzeFood(req: Request, res: Response): Promise<void> public async analyzeBatch(req: Request, res: Response): Promise<void> private validateImage(file: MulterFile): ValidationResult private handleAnalysisError(error: Error, res: Response): void

Chatbot Controller

class ChatbotController
This class manages chatbot interactions with users. It handles both regular and streaming

responses from Gemini API[6]. Validates message length, enforces rate limits (10 messages/minute), and caches conversation history for context-aware responses.
Attributes
private ChatbotService chatbotService private RateLimiter rateLimiter private ConversationCache conversationCache
Methods
public async chat(req: Request, res: Response): Promise<void> public async chatStream(req: Request, res: Response): Promise<void> public async getChatHistory(req: Request, res: Response): Promise<void> private validateMessage(message: string): boolean private filterPersonalInformation(message: string): string private handleChatError(error: Error, res: Response): void

3.2 Mobile Client (Frontend)

3.2.1 Shared Data Types

This section describes the data structures defined within the Mobile Client to strictly type the JSON responses received from the Backend API. These models ensure type safety for the UI components

- **UserProfile Interface:** This interface represents the logged-in user's complete profile within the application. It combines personal identity information with gamification progress. It ensures that the user's progress and health targets are displayed consistently across the main menu and dashboard screens.
- **UpdateMealParams Interface:** This interface defines the specific data format used when navigating between the AddMeal and MainPage screens. It acts as a safety check to ensure that when a user logs a new meal, the correct calorie and food data is passed back to the main dashboard for immediate update.
- **Post Interface:** Defined to structure user-generated content within the SocialFeed. Its primary purpose is to distinguish between standard text/image updates and structured Recipe posts. It allows the UI to conditionally render complex recipe details
- **Challenge Interface:** Defined to manage the gamification state on the client side. It acts as the contract between the GamificationController and the UI, specifically for visualizing progress bars.
- **PublicUser Interface:** Defined to implement a Data Minimization strategy for social interactions. Unlike the full UserProfile model, this interface strictly exposes only non-sensitive public information (username, avatar, follow status).
- **Badge Interface:** Defined to structure the reward system data.

- **Comment Interface:** This interface defines the structure for user comments within the social feed. It groups the message text together with the sender's display information.

3.2.2. Screens

class Login
This class manages the user authentication process. It implements a smart input logic that automatically distinguishes between an email and a username based on the input format (checking for '@'). It handles the asynchronous communication with the backend (/api/auth/login), securely stores the received JWT token using SecureStore[17], and manages the "Remember Me" persistence logic. It also handles error states and navigation to the Sign-Up or Main Page.
Attributes
private String email private String username private String password private boolean remember private boolean loading private NavigationProp navigation
Methods
private async handleLogin(): Promise<void> private handleInputChange(text:string): void private toggleRememberMe(): void

class SignUp
This class initiates the multi-step registration process. It serves as the primary data collection interface for user credentials (username, email, password). Instead of immediately creating an account, it acts as a "Data Carrier," validating the initial inputs and passing them as navigation parameters to the subsequent data collection stage (SignUpEnterData). This ensures a seamless user experience without partial account creation.
Attributes
private String email private String name private String password

private boolean loading private NavigationProp navigation
Methods
private async handleSignUp(): void

class SignUpEnterData
This class manages the second phase of registration. It receives the initialData (credentials) from the previous screen and prompts the user for physiological metrics necessary for the AI's nutritional calculations. It calculates derived metrics like BMI (if applicable) and prepares the complete profile data for the final avatar creation step.
Attributes
private int age private float weight private float height private String gender (‘male’/’female’) private String goal private String target_weight private boolean loading public RouteProp route (contains initialData) private NavigationProp navigation
Methods
private async handleSignUpData(): Promise<void>

class CreateAvatar
This class concludes the user registration workflow. It receives the cumulative finalData object from the previous steps, which contains the user's credentials and physiological profile. It presents a grid of predefined avatar images for selection. Upon confirmation, it acts as the final transaction handler, sending the complete payload (User Data + Avatar URL) to the backend API /api/auth/register endpoint to persistently create the user account.
Attributes

private String name private String selected private Array<Object> avatars private boolean loading public RouteProp route (contains finalData) private NavigationProp navigation
Methods
private async handleContinue(): Promise<void> private handleSelect(avatarPath: Object): void private getUserName(): void

class MainPage
This class represents the central dashboard of the NutriGame application. It aggregates and visualizes the user's daily nutritional progress using animated charts (CalorieCircle). It implements a date-based navigation system, allowing users to switch between past and present logs. Furthermore, it serves as the primary navigation router, providing entry points to meal logging (Breakfast, Lunch, etc.), water tracking, and AI-powered features (ScanFood, Chatbot). It ensures data freshness by re-fetching stats every time the screen gains focus.
Attributes
private float calorie private float protein private float carb private float fat public Date selectedDate Private float dailyGoal private Animated.Value slideAnim Public NavigationProp navigation
Methods
Private async fetchDailyData(): Promise<void> Private changeDate(days: number): void Private async fetchUserGoal(): Promise<void>

class AddMeal
This class facilitates the manual entry of food items into the user's daily log. It interfaces with the Global Food Library (/api/food/search_food) to allow users to search for items. It features a specialized "Portion Control System" (Modal + Stepper) that allows users to adjust the serving size (e.g., 1.5 portions) and automatically recalculates the macronutrients before saving. It also supports real-time synchronization, fetching existing logs for the specific meal category (e.g., Breakfast) upon entry.
Attributes
private String searchText private Array<FoodItem> filteredData private Array<FoodItem> selectedItems private FoodItem selectedMeal public float stepperValue Public NavigationProp navigation
Methods
Private async searchFood(text: String): Promise<void> Private handleAddWithStepper(): void Private async handleAddMeal(items: FoodItem[]): Promise<void> Private async handleRemoveItem(index: int, item: FoodItem): Promise<void> Private sync fetchDailyData(): Promise<void>

class AddWater
This class manages the user's hydration tracking. Unlike standard lists, it utilizes a gamified visual component (WaterWave) to represent the user's daily water intake percentage. It fetches the current water log from the backend to initialize the fluid level. It provides an interface for users to add water in standardized increments (e.g., 200ml, 500ml) and communicates these updates to the backend to ensure the daily hydration goal (default: 3000ml) is met.
Attributes
private int currentLevel private int dailyCapacity private float waveProgress private boolean loading public RouteProp (contains selectedDate) Public NavigationProp navigation

Methods
Private async fetchHydrationData(): Promise<void>
Private async handleAddWater(amount: int):Promise<void>
Private calculateProgress(current: int, total: int): float

class ScanFood
This class serves as the hardware interface for the "AI-Assisted Logging Flow". It manages the device's camera hardware using the Expo-camera library[7] to capture high-resolution food images. It implements a robust permission handling logic, prompting the user for access and guiding them to system settings if denied. Once an image is captured, it provides a preview layer for validation before initiating the upload process to the AI analysis engine.
Attributes
private CameraView cameraRef
private String photoUri
private PermissionResponse permission
Public NavigationProp navigation
Methods
Private async takePicture(): Promise<void>
Private async handleScan(): Promise<void>
Private async handlePermissionButton(): Promise<void>

class Chatbot
This class implements the conversational interface for the "NutriCoach" AI assistant. It utilizes react-native-gifted-chat to provide a familiar messaging experience. Unlike standard chat apps, it includes a custom Markdown renderer to properly display structured AI responses (e.g., diet plans, bullet points). It also manages a local session history system, allowing users to create new conversations, switch between past sessions via an animated sidebar menu, or delete old logs.
Attributes
private Array<IMessage> messages
private Array<ChatSession> history
private String chatID

Private boolean showMenu Private Animated.Value slideAnim NavigationProp navigation
Methods
Private onSend(messages: IMessage[]): void Private toggleMenu(): void Private loadChat(session: ChatSession): void Private openNewChat(): void

class Menu
This class serves as the central hub for user profile management and application settings. Unlike a static list, it acts as a secondary dashboard that visualizes the user's gamification status (Current Streak) using Lottie animations [16] and displays key health targets (Target Weight). It fetches the latest user profile data from the backend upon focus to ensure avatar and username consistency. It also provides navigation entry points to detailed settings and handles the secure logout process.
Attributes
private UserProfile profile private boolean loading Public NavigationProp navigation
Methods
Private async fetchProfile(): Promise<void> Private async handleLogout(): Promise<void> Private getAvatarSource(path: string): ImageSource

class ProfileSettingsMenu
This class serves as the configuration panel for critical user account settings. It receives the current UserProfile data via navigation parameters to display a personalized header. Its primary responsibility is to provide secure access to account modification features (Edit Profile) and irreversible actions (Delete Account). It implements a confirmation modal system to prevent accidental data loss and handles the secure cleanup of credentials upon account deletion.
Attributes

private UserProfile userData private RouteProp route (contains userData) Public NavigationProp navigation
Methods
Private async handleDeleteAccount(): Promise<void> Private getAvatarSource(path: string): ImageSource

class EditProfile
This class provides a comprehensive form interface for users to update their personal information and physical metrics. It pre-fills the fields with the current userData passed via navigation parameters. It manages the state for editable attributes (username, email, weight, height) and includes a modal-based avatar selector. Upon submission, it validates and sends the updated payload to the backend via a PUT request, ensuring the user's profile remains current for accurate calorie calculations.
Attributes
private String username private String email Private String password Private String currentWeight Private String height Private String selectedAvatar Private boolean saving Public RouteProp route (contains userData) Public NavigationProp navigation
Methods
Private async handleSave(): Promise<void> Private getAvatarSource(path: string): ImageSource

class SocialFeed

This class serves as the dynamic hub for community interaction. It calls SocialSharingController.getFeed to retrieve a mixed stream of user-generated content, including recipes, workout achievements, and challenge updates. It implements to fetch the latest data and handles the rendering of different post types (Recipe vs. Standard Post).
Attributes
Private Array<Post> feedData Private boolean refresh Private boolean loading Public NavigationProp navigation
Methods
Private async fetchFeed():Promise<void> Private renderFeedItem(item: Post): JSX.Element Private navigateToCreatePost(): void

class CreatePost
This class allows users to share content or recipes. It provides a form interface that collects text, images, and optional recipe details (ingredients, steps). Upon submission, it constructs the payload and calls SocialSharingController.createPost. It handles image compression and validation before the network request.
Attributes
Private String caption Private String imageUri Private boolean isRecipe Private String ingredients Private boolean uploading Public NavigationProp navigation
Methods
Private async handlePostSubmit():Promise<void> Private async pickImage(): Promise<void>

class UserProfileView
This class allows users to view other users' profiles. It calls SocialSharingController.getUserPosts to display their content history. Crucially, it manages the relationship status by calling SocialSharingController.followUser or unfollowUser based on the current state.
Attributes
Private PublicUser targetUser Private Array<Post> userPosts Private boolean isFollowing Private boolean loading Public RouteProp route Public NavigationProp navigation
Methods
Private async fetchProfileData(): Promise<void> Private async handleFollowToggle(): Promise<void>

class ChallengeProgress
This class visualizes the specific details of an active challenge. It calls GamificationController.getChallengeProgress to show a progress bar. It listens for updates from handleProgressEvent via the backend which aggregates data from the user's MealLog and WaterLog. When a challenge reaches 100%, it calls GamificationController.completeChallenge to claim rewards and badges.
Attributes
Private Challenge challengeData Private float currentProgress Private boolean isCompleted Public RouteProp route Public NavigationProp navigation
Methods

Private async fetchProgress(): Promise<void> Private async checkCompletion(): Promise<void> Private async claimReward(): Promise<void>

class CreateChallenge
This class enables users to create new challenges. It allows selecting a challenge type (e.g., Calorie Limit, Water Streak) and assigning it to themselves or a friend (from their following list). Upon confirmation, it calls SocialSharingController.createChallenge, which handles the logic of notifying the target user.
Attributes
Private String title Private string type Private string targetUserID Private Date endDate Public NavigationProp navigation
Methods
Private async handleCreate(): Promise<void> Private selectTargetUser(): void

3.2.3. Shared UI Components

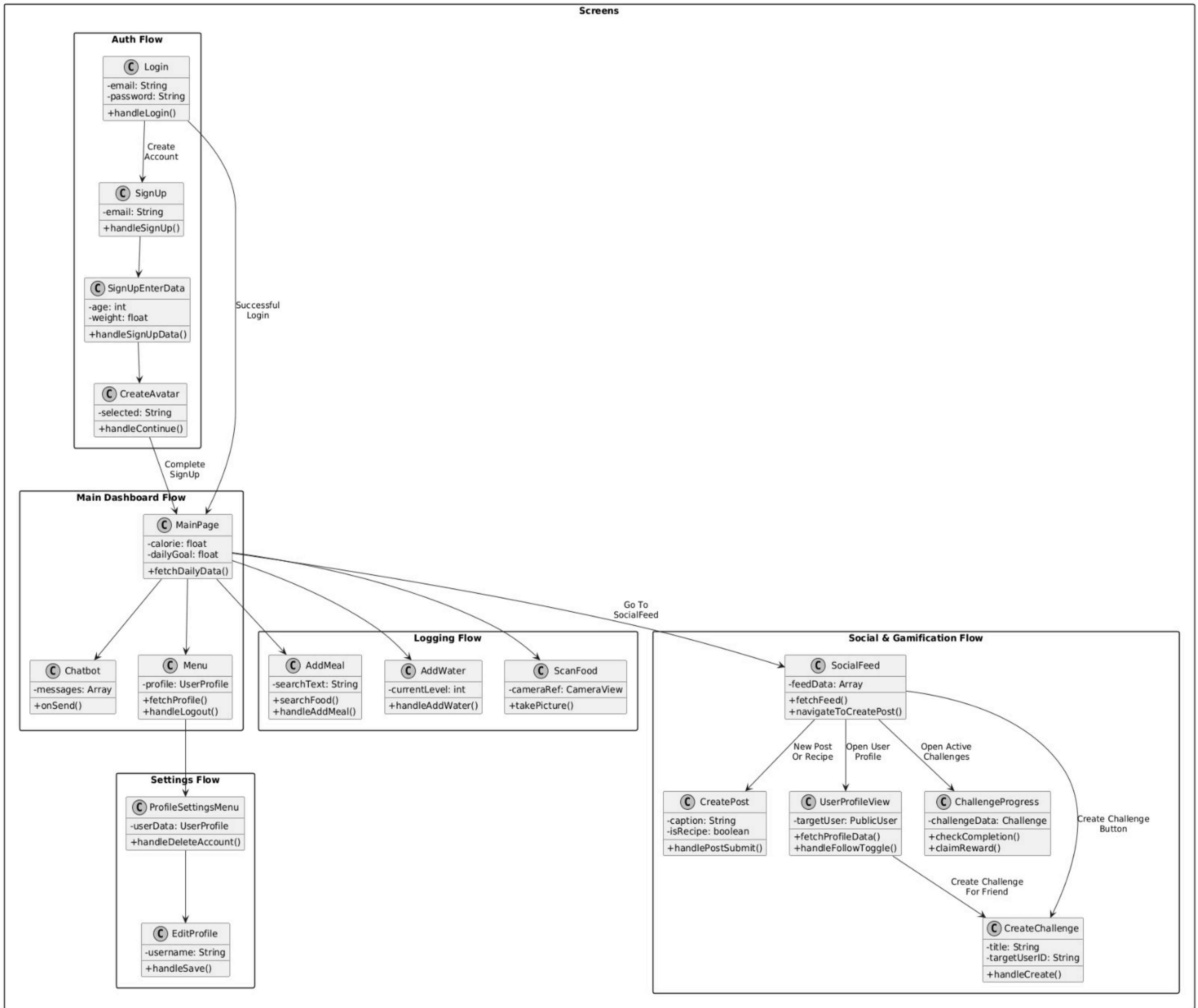
class CalorieCircle
This component acts as the primary visual indicator for the user's daily nutritional progress. It utilizes react-native-circular-progress to render an animated ring representing the calorie budget. Unlike simple charts, it encapsulates specific diet logic: it automatically calculates daily targets for Protein (30%), Carbohydrates (50%), and Fats (20%) based on the user's calorie goal and displays them as vertical "tube" progress bars alongside the main circle.
Attributes
Public float calories Public float goal

Public float protein Public float carb Public float fat
Methods
renderMacroTube(value,max, color, label) macroGoalCalculation

class GenderSelection
This functional component implements a custom radio-group interface for biological sex selection. It operates as a controlled component, synchronizing its internal state with the parent via the value prop and useEffect hook [5]. It provides immediate visual feedback through style toggling (styles.selected) when a user interacts with the touchable options, ensuring the UI always reflects the current selection state.
Attributes
Public ‘male’ | ‘female’ | null value
Methods
handleSelect()

Class GoalSelection
This functional component provides a standardized selection interface for the user's primary dietary goal. Unlike standard pickers, it utilizes the react-native-paper Menu component to offer a polished, modal-like experience. It enforces data integrity by restricting input to a specific set of enumerated values (e.g., "Weight Loss", "Muscle Mass"), ensuring compatibility with the backend's calorie calculation engine. It is wrapped in React.memo to optimize rendering performance.
Attributes
Public String value Public Function onChange
Methods

Class WaterContainer
This advanced visual component renders a realistic fluid simulation to represent hydration levels. It leverages @shopify/react-native-skia [14] for high-performance 2D graphics and react-native-reanimated [15] for smooth, 60fps animations. The component generates dynamic sine wave paths (createWavePath worklet) that oscillate based on a shared clock value, creating a continuous "sloshing" effect. It visually clips these waves within a circular container to simulate a water tank gauge.
Attributes
Public float progress Public int maxCapacity
Methods
createWavePath(progress: number, clockValue: number, size: number, phaseOffset: number, amplitude: number, frequency: number)



Client-Side Class Diagram

4. Glossary

AI (Artificial Intelligence): Machine learning systems that analyze data to make intelligent predictions. In NutriGame, AI is utilized for the Food Recognition Service to automatically estimate calorie values from photos and for the Chatbot to provide psychological support.

Asynchronous Processing: A programming pattern where the application does not

wait for a long-running task to finish before allowing other interactions. NutriGame implements this to ensure the UI remains responsive while waiting for heavy AI analysis tasks to complete in the background.

Circuit Breaker: A design pattern used to detect failures and prevent the application from constantly trying to execute a failing operation. The FoodRecognitionController implements this pattern to handle external AI service failures gracefully without crashing the application[20].

EXIF (Exchangeable Image File Format): Standard metadata embedded in image files containing details like location and camera settings. NutriGame explicitly strips this data from user-uploaded photos to protect user privacy before processing them [18].

Gamification: The integration of game mechanics into non-game applications to enhance engagement. NutriGame utilizes gamification through daily streaks, challenges, and animated rewards to motivate users to track their nutrition [11].

Gemini API: An external AI service provided by Google. It is used by the ChatbotController to generate empathetic, context-aware responses and by the FoodRecognitionController for image analysis [6].

JWT (JSON Web Token): A compact, URL-safe means of representing claims to be transferred between two parties. The AuthenticationController issues these tokens to validate user sessions statelessly across the Mobile Client and Backend API [8].

Lottie: A library for rendering Adobe After Effects animations in real-time. The Menu class uses Lottie to render high-quality animations for the user's "Current Streak" display [16].

React Native: A framework for building native applications using React. NutriGame uses this to maintain a single, portable codebase that functions on both iOS and Android devices [5].

SecureStore: A React Native library that provides a way to encrypt and securely store key-value pairs on the device. The Login class uses this to safely store the user's JWT token, preventing unauthorized access [17].

Worklet: A small JavaScript function that runs on a separate, high-performance UI thread. The WaterContainer component uses worklets to calculate fluid wave physics at 60fps without blocking the main JavaScript thread [15].

5. References

[1] B. Bruegge and A. H. Dutoit, Object-Oriented Software Engineering Using UML, Patterns, and Java, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2010.
[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA, USA: Addison-Wesley, 1994.
[3] Object Management Group, "Unified Modeling Language (UML) Specification, Version 2.5.1," OMG, Needham, MA, USA, Standard formal/2017-12-05, Dec. 2017. [Online]. Available: https://www.omg.org/spec/UML/2.5.1
[4] IEEE, "IEEE Standard for Software Design Descriptions," IEEE Std 1016-2009, Jul. 2009. doi: 10.1109/IEEESTD.2009.5167255.
[5] Meta Platforms, Inc., "React Native: Learn once, write anywhere," 2024. [Online]. Available: https://reactnative.dev/docs/getting-started
[6] Google, "Gemini API Documentation," 2024. [Online]. Available: https://ai.google.dev/docs
[7] Expo, "Expo Documentation," 2024. [Online]. Available: https://docs.expo.dev
[8] Auth0, "JSON Web Tokens (JWT) Introduction," 2024. [Online]. Available: https://jwt.io/introduction
[9] M. Nystrom, Game Programming Patterns. Genever Benning, 2014. [Online]. Available: https://gameprogrammingpatterns.com
[10] M. T. Nygard, Release It! Design and Deploy Production-Ready Software, 2nd ed. Raleigh, NC, USA: Pragmatic Bookshelf, 2018.
[11] S. Deterding, D. Dixon, R. Khaled, and L. Nacke, "From game design elements to gamefulness: Defining 'gamification'," in Proc. 15th Int. Academic MindTrek Conf.: Envisioning Future Media Environments, Tampere, Finland, 2011, pp. 9–15. doi: 10.1145/2181037.2181040.
[12] D. Johnson, S. Deterding, K.-A. Kuhn, A. Staneva, S. Stoyanov, and N. Hides, "Gamification for health and wellbeing: A systematic review of the literature," Internet Interventions, vol. 6, pp. 89–106, Nov. 2016. doi: 10.1016/j.invent.2016.10.002.
[13] Express.js, "Express – Node.js web application framework," 2024. [Online]. Available: https://expressjs.com
[14] Shopify, "React Native Skia Documentation," 2024. [Online]. Available: https://shopify.github.io/react-native-skia
[15] Software Mansion, "React Native Reanimated Documentation," 2024. [Online]. Available: https://docs.swmansion.com/react-native-reanimated
[16] Airbnb, "Lottie for React Native," 2024. [Online]. Available: https://airbnb.io/lottie
[17] Expo, "Expo SecureStore," 2024. [Online]. Available: https://docs.expo.dev/versions/latest/sdk/securestore
[18] OWASP Foundation, "OWASP Mobile Application Security Verification Standard (MASVS)," 2023. [Online]. Available: https://mas.owasp.org/MASVS
[19] R. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, Univ. of California, Irvine, CA, USA, 2000.
[20] M. Fowler, "CircuitBreaker," martinfowler.com, 2014. [Online]. Available: https://martinfowler.com/bliki/CircuitBreaker.html