WellNex Documentation

# CONTENTS

- Problem Statement

- Target User

- Objectives

- Functional & Non-Functional Requirements

- User Stories

- System Constraints

- Expected Design Patterns

- Use Case Scenario

- Repository

- Class Diagram Description

# Problem Statement

Maintaining a healthy lifestyle is challenging for many individuals due to a lack of personalized guidance on nutrition, exercise, and wellness habits. Generic health recommendations often fail to consider individual differences such as age, height, weight, and personal goals.  
  
Many people struggle with identifying the right foods to eat, effective workout routines, and sustainable habits that align with their specific health objectives.  
  
This gap leads to confusion, inconsistent progress, and, in some cases, negative health outcomes due to misinformation or lack of direction.  
  
There is a need for a smart, data-driven solution that provides users with a personalized roadmap for their health journey—helping them make informed decisions about diet, exercise, and lifestyle changes tailored to their unique needs.

# Target User

The application is designed for individuals who seek personalized health guidance, including:  
- Health Enthusiasts  
- Beginners in Fitness & Wellness  
- Weight Management Seekers  
- Busy Professionals  
- Elderly Users  
- Patients with Lifestyle-Related Concerns

# Objectives

- Personalized Health Guidance  
- Comprehensive Roadmap  
- Nutrition and Diet Suggestions  
- Exercise and Fitness Plans  
- Progress Tracking  
- User-Friendly Experience  
- Educational Insights

# Functional Requirements

- User Profile Management  
- Health Assessment/Input  
- Personalized Health Recommendations  
- Recommendation Display  
- Communication and Support

# Non-Functional Requirements

- Performance  
- Scalability  
- Reliability  
- Security  
- Compatibility  
- Privacy  
- Accessibility  
- Usability

# User Stories

- Health Enthusiast: Receive structured plans.  
- Beginner: Clear steps to develop habits.  
- Weight Management Seeker: Personalized diet/workouts.  
- Busy Professional: Quick effective advice.  
- User: Track progress.

# System Constraints

- Legal & Compliance Constraints  
- User Experience & Accessibility  
- Limited Computational Resources  
- User Load Handling  
- Error Handling & Recovery

# Expected Design Patterns

MVVM (Model-View-ViewModel)  
- Separation of concerns  
- Easier testing  
- Maintainability

# Use Case Scenario

Sarah's Personalized Weight Loss Journey  
- Profile Creation  
- Health Assessment  
- Personalized Recommendations

# Repository

https://github.com/abdellaziizz/WellNex

Class Diagram Description

# 1. Overview

**The system consists of:**

* **User Hierarchy**: Abstract User class with concrete subclasses for different user types.
* **Recommendation Hierarchy**: Abstract Recommendation class with concrete subments (DietPlan, ExercisePlan, Meal).
* **Factories**: Interfaces (IFactory, Factory) and concrete implementations (ConcreteFactory) to instantiate objects.

# 2. Class Responsibilities

## 2.1 User Hierarchy (Abstract & Subclasses)

User**(Abstract Class)**

* **Responsibility**: Base class storing common user attributes.
* **Attributes**:
  + Name, Age, Gender, Height, Weight
* **Methods**:
  + CalculateBMI(): float – Computes BMI (Weight / (Height^2)).

## 2.2 Recommendation Hierarchy (Abstract & Subclasses)

Recommendation**(Abstract Class)**

* **Responsibility**: Base class for diet/workout plans.
* **Attributes**:
  + PlanName: String
* **Methods**:
  + GeneratePlan(): void – Abstract method to create a plan.

## 2.3 Factory Components (Factory Method Pattern)

**Interfaces**

1. IFactory
   * **Method**: CreateProfile(): User  
     Instantiates a User subclass based on input

(e.g., "Beginner"→ Beginner object).

1. Factory (Interface for Recommendations)
   * **Method**: createRecommendation(user: User): Recommendation  
     Creates a DietPlan or ExercisePlan tailored to the user.

**Concrete Implementations**

* ConcreteFactory  
  Implements both interfaces to:
  + Create User profiles (e.g., BusyUser, FitnessUser).
  + Generate Recommendation objects (e.g., DietPlan for WeightMgmtUser).

# 3. Relationships Between Classes

1. **Inheritance (Generalization)**:
   * User ← Beginner, WeightMgmtUser, etc. (subclasses extend User).
   * Recommendation ← DietPlan, ExercisePlan, Meal.
2. **Factory Method Pattern**:
   * IFactory/Factory interfaces define creation methods.
   * ConcreteFactory implements them to decouple object creation from usage.
3. **Dependency**:
   * DietPlan depends on User to calculate needs (CalculateDailyNeeds(user)).
   * ConcreteFactory depends on User and Recommendation hierarchies.
4. **Association**:
   * Meal is associated with DietPlan (e.g., a DietPlan contains multiple Meal objects).

# 4. Key Dependencies

The Recommendation class and its subclasses rely on the User**object** to:

* Access user-specific attributes (e.g., Age, Weight, FitnessLevel).
* Call methods (e.g., CalculateBMI()) for calculations.
* Determine which type of plan to generate (e.g., DietPlan for WeightMgmtUser).

# 5. How User Attributes Drive Recommendations

**5.1 ExercisePlan Class**

**Method: SuggestExercises() → List<String>  
Logic:**

* **Checks user subclass to customize workouts:**
  + **Beginner: Low-intensity exercises (user.FitnessLevel = "low").**
  + **BusyUser: Short workouts (user.AvailableTimePerDay < 30 mins).**
  + **Elder: Balance-focused routines (user.MobilityLevel = "moderate").**

# 6. Factory Method’s Role in User-Specific Plans

The Factory**interface** ensures the correct Recommendation subclass is created for each User:

1. ConcreteFactory.createRecommendation(user) inspects the User object:
   * If user is WeightMgmtUser → Returns DietPlan.
   * If user is FitnessUser → Returns ExercisePlan with TrainingTypes.
2. The generated plan **automatically adapts** to the user’s attributes.

# 7. Example Scenarios

|  |  |  |
| --- | --- | --- |
| User Type | Recommendation Generated | Key Logic |
| WeightMgmtUser (TargetWeight=70kg, DietType="keto") | DietPlan with TargetCalories=1800, DietType="keto" | Uses user.Weight and DietType. |
| Beginner (FitnessLevel="low") | ExercisePlan with IntensityLevel="low", DurationPerSession=20 mins | Adjusts for fitness newbies. |
| Elder (MobilityLevel="low") | ExercisePlan with chair yoga, hydration reminders | Prioritizes safety. |

WEEK5

# WellNex Prototype Demo & Code Documentation:

**This delivery covers the following:**

* Initial Implementation of Core Functionalities
* Key Design Patterns Implemented
* Code Snippets & Explanation
* Each Applied Design Pattern with Explanation
* How and Where It Is Used
* Benefits & Trade-offs of Each Pattern
* Code Samples with Justifications

## 1.Initial Implementation of Core Functionalities

1. **Splash Screen**:

- Animated logo, Firebase initialization, heart–beat effect.

1. **Login Flow:**

- Email/password authentication via Firebase Auth.

- Interactive UI with fade, scale, and slide transitions.

1. **Profile Setup:**

- User registration form (username, email, password, age, weight, height, gender, goal).

- Validation (required fields, logical ranges).

- Onboarding flow with staggered field animations.

1. **Home** **Dashboard**:

- Overview of daily goals and navigation to features.

1. **Learn Module:**

- Article list with category filters and search.

- Featured daily tip with randomization and fade animation.

1. **Exercise Plan:**

- Plan list, completion tracking, rest timer, expandable details.

1. **Progress Tracker:**

- Chart-based health data display, animated indicators.

2. Key Design Patterns Implemented

|  |  |
| --- | --- |
| Pattern | Purpose |
| MVVM | Separate UI from state/logic (Provider) |
| Repository | Abstract data source implementations |
| Dependency Injection | Centralize object creation (Provider/Factory) |
| Observer | Reactive updates (ChangeNotifier) |
| Singleton | Global shared instances (services) |

### 3. Code Snippets & Explanation

#### 3.1 Dependency Injection (Factory)

A black text on a white background

AI-generated content may be incorrect.

**Explanation:** `GetIt` registers shared services and viewmodels. Allows constructor injection rather than `new` inside classes.

#### 3.2 MVVM Binding

A computer code with black text

AI-generated content may be incorrect.

**Explanation:** UI calls `signIn()` on VM; VM updates `isLoading` and notifies UI via `notifyListeners()`.

### 4. Each Applied Design Pattern with Explanation

#### 4.1 MVVM

**What:** Model–View–ViewModel separates UI (View), data (Model), and logic/state (ViewModel).

**Where:** `viewmodels/` + `screens/`; UI uses `Consumer` or `Provider.of` to listen to VM.

**Why:** Improves testability, maintainability; UIs are stateless, VMs hold state.

#### 4.2 Repository

**What:** Encapsulates data source logic behind abstract methods.

**Where:** `repositories/` folder; used by VMs via constructor.

**Why:** Decouples network/database code from VMs and UI; easier to mock in tests.

#### 4.3 Dependency Injection

**What**: Central object creation rather than `new` throughout code.

**Where:** `factory.dart` or `main.dart` registering providers.

**Why:** Swapping implementations and testing become simpler.

#### 4.4 Observer (ChangeNotifier)

**What:** Provides reactive updates via `notifyListeners()`.

**Where:** VMs extend `ChangeNotifier`, UI uses `ChangeNotifierProvider`.

**Why:** Efficiently updates only widgets that depend on changed data.

#### 4.5 Singleton

**What:** Ensures a single shared instance of a class (e.g., `AuthService`).

**Where:** Registered in DI container as lazy singleton.

**Why:** Common instance for operations such as authentication state.

## 5. How and Where It Is Used

**MVVM:** All feature screens wrap with `ChangeNotifierProvider(create: ...)`; listen to VMs.

**Repo:** `UserViewModel` calls `UserRepository.login(...)`; repository accesses Firebase.

**DI:** `main.dart` calls `setup()`; screens & VMs retrieve with `Provider.of` or `sl<MyService>()`.

**Observer:** VM state changes drive UI rebuilds in `Consumer` widgets.

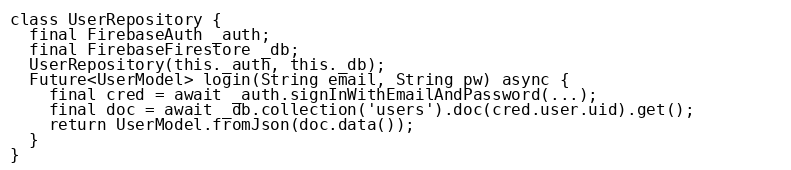
**Singleton:** Services like `AuthService` used across app; holds auth token.

## 6. Benefits & Trade-offs of Each Pattern

|  |  |  |
| --- | --- | --- |
| Pattern | Benefits | Trade-offs |
| MVVM | Clear separation, testable logic | Boilerplate, learning curve |
| Repository | Decouples data sources, mocks easily | Extra layer, slight indirection |
| DI | Flexible, swap implementations easily | Setup complexity, global state |
| Observer | Reactive UI, efficient updates | Harder to trace update sources |
| Singleton | Single shared instance, easy access | Hidden dependencies, global state |

## 7. Code Samples with Justifications

#### 7.1 Repository Pattern



**> \*Justification:\*** Repository exposes high‐level `login` method, hiding Firebase SDK details from VMs.

#### 7.2 DI with Provider

A black text on a white background

AI-generated content may be incorrect.

**> \*Justification:\*** Provider package enables lazy creation and easy access via `context.read()` or `context.watch()`.

<small>Generated: 2025-05-15</small>