Java-TrainingHub

Domain Driven Design Application

UNIVERSITÀ DEGLI STUDI FIRENZE

DEPARTMENT OF INFORMATION ENGINEERING

Bachelor's Degree in Computer Engineering

Project submitted for the final exam of

Software Engineering (B003372)

Professor: Enrico Vicario
Authors: Donati Federico
Timofte Alessio
Date: June 18, 2025

Contents

List of Figures

List of Tables

1 Introduction

In recent years, the intersection of technology and personal fitness has led to a significant proliferation of digital tools aimed at improving individual health outcomes. From mobile fitness trackers to AI-powered health assistants, the market is saturated with applications that promise personalized training and performance optimization. However, many existing solutions either oversimplify workout planning or fail to support the dynamic, bi-directional communication required between personal trainers and their trainees. This often results in rigid training plans, poor adherence, and limited opportunities for real-time adaptation.

Java-TrainingHub seeks to address these gaps by offering a feature-rich, modular, and extensible platform designed to facilitate the effective management of fitness activities through seamless trainer-trainee collaboration. Developed as part of the Software Engineering Final Project by **Donati Federico** and **Alessio Timofte**, the application was conceived and implemented following domain-driven design (DDD) principles, with an emphasis on scalability, maintainability, and usability. The project reflects a strong adherence to software engineering best practices, incorporating proven architectural patterns and a robust backend logic tailored to the unique needs of fitness coaching.

The system distinguishes between two primary user roles: *Trainees*, who log their workout activity and monitor their progress, and *Personal Trainers*, who can not only manage their own fitness routines but also create, assign, and adapt personalized workout plans for their clients. Crucially, the platform supports real-time notification mechanisms to keep trainees updated whenever their plans are modified, ensuring transparency and alignment between both parties.

2 Requirements Analysis

2.1 Problem Statement

In an increasingly digitized society where personal health and fitness have become top priorities, digital workout applications have emerged as tools for individuals seeking to monitor, manage, and improve their physical performance. Despite the abundance of such applications on the market, the majority are designed with a generic user in mind and offer limited interaction between trainers and clients. Most fitness tracking tools are either self-directed—lacking expert guidance—or centered on static programs that do not dynamically adapt to user progress or trainer feedback.

Furthermore, personal trainers—key stakeholders in the fitness domain—are often left to rely on fragmented tools: spreadsheets, chat applications, or proprietary gym software that is difficult to integrate or use remotely. This fragmentation impedes the ability of trainers to oversee multiple clients, monitor their progress in real time, and provide personalized feedback within a single coherent system. Trainees, in turn, face difficulties in adhering to plans, tracking performance trends, or receiving timely adjustments aligned with their goals and capabilities.

The core problem, therefore, lies in the absence of an integrated, role-sensitive platform that enables:

- Continuous, two-way communication between trainers and trainees,
- Efficient plan creation, assignment, and versioning,
- Structured progress tracking with timely notifications,
- Dual-role usage, allowing trainers to also act as trainees in their own right.

Java-TrainingHub is developed to fill this gap. It aims to provide a unified software solution in which:

- Trainers can manage clients and design bespoke training plans,
- Trainees can log workouts and receive immediate updates,

- Both parties can track progress over time.

The project approaches this problem with a domain-driven design methodology, ensuring that the business logic of fitness planning is faithfully modeled and separated from technical concerns such as persistence and user interface. Through a combination of Java's strong object-oriented features, PostgreSQL's relational robustness, and scalable design patterns such as *Observer* and *Strategy*, *Java-TrainingHub* delivers a foundation for an intelligent, maintainable, and practical fitness collaboration tool.

2.2 Involved Actors

The Java-TrainingHub system is designed to facilitate structured interactions between users operating in defined roles, as well as technical components responsible for data persistence and application lifecycle management. Each actor plays a crucial role in realizing the system's objectives and ensuring its operational integrity. The primary actors are as follows:

- **Trainee:** This is the fundamental end-user role within the system. A Trainee is an individual engaged in a fitness journey who uses the application to view assigned workout plans, log completed sessions, and track progress over time. Trainees are passive recipients of plans created by Personal Trainers but are active agents in executing and reporting their workouts. They receive real-time notifications when their plans are updated and can access their workout history for self-monitoring.
- **Personal Trainer (PT):** The Personal Trainer is an advanced role that extends all functionalities of a Trainee while adding capabilities essential for client management. A PT can create, update, and assign workout plans to any Trainee under their supervision. Additionally, they can view the progress and history of their assigned clients, manage their client roster, and use the application for their personal training. This dual-capability model is designed to reflect the real-world scenario where fitness professionals are often active athletes themselves.
- **System Administrator (Implicit):** While not directly represented with a UI or exposed endpoints, a System Administrator is implicitly responsible for the initial setup of the application environment—including database provisioning, configuration of system parameters, and monitoring of runtime behavior. This actor is especially relevant during development and deployment phases, and in future iterations involving cloud deployment or CI/CD pipelines.

These actors form the foundational interaction network of Java-TrainingHub. Their relationships and responsibilities are formalized in subsequent use case definitions and inform the domain model structure.

2.3 Core Requirements

At the heart of the system is a robust user-role design that distinguishes between *Trainees* and *Personal Trainers*. Trainees can view a personalized workout plan assigned to them by their trainer, upload detailed records of completed sessions, and receive real-time notifications when their workout plans are modified. This structure fosters engagement and accountability while enabling accurate performance tracking.

Personal Trainers, on the other hand, are granted a comprehensive set of management tools. These include the ability to create, update, and delete workout plans; assign plans to individual trainees; and monitor workout records submitted by their clients. A key requirement fulfilled by the system is role interoperability: a Personal Trainer may also operate as a Trainee within the same account, ensuring maximal reusability of the platform for fitness professionals who also train themselves.

In order to support this interaction-heavy model, the application implements an *Observer Pattern* to notify trainees about plan changes as soon as they occur. The *Strategy Pattern* is used to define exercise variations based on training goals such as strength, endurance, or hypertrophy. The system also applies the *Factory Pattern* to create reusable, modular workout components.

Although the system currently provides a command-line interface for testing and interaction, it has been architected with modularity in mind to support future enhancements, including graphical interfaces, analytics dashboards, wearable device integrations, and AI-driven workout generation. In its current form, *Java-TrainingHub* offers a complete demonstration of how software engineering best practices—such as separation of concerns, domain-driven design, and layered architecture—can be applied to solve real-world problems in the fitness and health domain.

2.4 Possible Drawbacks of the Approach

While the *Java-TrainingHub* system presents a coherent and extensible architecture for fitness management, it is not without its limitations. These drawbacks arise primarily from trade-offs made during design and implementation, resource constraints, and the deliberate exclusion of certain features to maintain focus on core functionality.

- **No Real-Time Synchronization Across Devices:** Although the system supports real-time notifications in logical terms (e.g., via observer patterns), there is no actual implementation of real-time data synchronization mechanisms such as WebSockets or push notifications. As a result, updates are only visible upon subsequent user interaction, which could delay critical trainer–trainee communications.
- **Manual Database Configuration:** The system requires manual setup of the PostgreSQL database and lacks automated deployment scripts or environment configuration tooling (e.g., Docker). This can lead to deployment errors, increased onboarding time for new developers, and reduced reproducibility of the environment.
- **No Mobile or Cross-Platform Support:** The application does not currently support mobile platforms or responsive web access. In the fitness domain—where trainers and trainees often use smartphones during workouts—this restricts the practical deployment of the application outside of controlled testing environments.
- **Limited Domain Coverage:** While the system models core training workflows effectively, it does not account for adjacent domains such as nutrition tracking, wearable device integration, or psychological metrics. These aspects are essential for delivering a holistic fitness coaching platform but are left for future extensions.
- **No Exercise Execution Support or Visualization:** The system lacks built-in support for visualizing or explaining individual exercise execution (e.g., proper form, repetitions with media content, required equipment). While exercises are represented in a structural form through objects and strategies, there is no user-facing component for instruction, demonstration, or progress tracking at the movement level. This limits the system's utility for beginners or remote coaching scenarios.

3 System Design

The system design phase was a critical stage in the development of the Java-TrainingHub application, focusing on conceptualizing the architecture and interactions of its various components. This process involved the strategic use of several specialized software tools to visualize, model, and refine the system's structure before implementation. These tools were instrumental in translating high-level requirements into detailed design specifications.

During this phase, the following key software applications were utilized:

- **StarUML:** This robust modeling platform played a pivotal role in crafting comprehensive Unified Modeling Language (UML) diagrams. StarUML enabled the detailed visualization of various system elements, including use cases, class structures, and component relationships, providing a clear blueprint for development.

- **Draw.io (Diagrams.net):** An intuitive and versatile online diagramming tool, Draw.io was extensively used for developing entity-relationship (ER) schemas. This facilitated the conceptual design of the database, ensuring logical organization and clear relationships between data entities.
- **Claude AI (Sonnet 4) by Anthropic:** Despite its superficial utility in rapid prototyping, Claude AI ultimately served more as a preliminary brainstorming tool for user interface mockups, rather than a reliable design partner. While it could quickly generate conceptual layouts and visual representations, its outputs frequently required substantial human correction and critical validation. The inherent limitations of such AI tools, particularly in accurately understanding and reflecting complex structural relationships, were evident and are elaborated upon further in Section 6.3 where their often-misleading contributions to design are critically assessed.

Use Case Diagram Description

Figure 1 shows the use case diagram for the **Java-TrainingHub** system. The diagram identifies two main actors with different system capabilities:

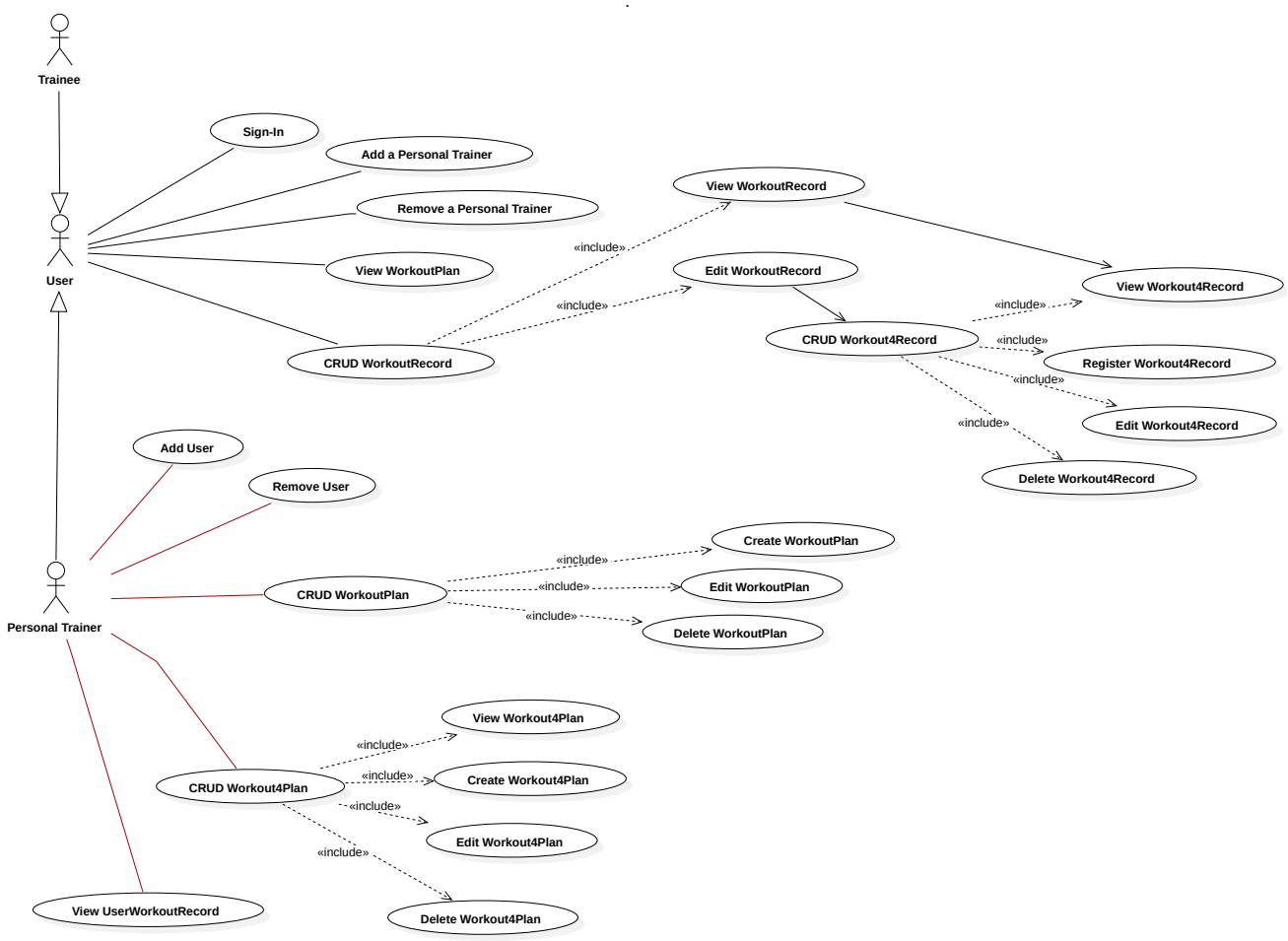


Figure 1: Use Case Diagram of Java-TrainingHub System

Actors and Capabilities

1. Trainee (User)
Core capabilities include:

- Sign-In: Access the system
- Add/Remove Personal Trainer: Manage trainer associations
- View WorkoutPlan: See training schedules
- CRUD WorkoutRecord: Create, read, update, delete exercise logs
- CRUD Workout: Manage exercise definitions
- CRUD WorkoutPlan: Manage training plans
- View UserWorkoutRecord: Access workout histories

2. Personal Trainer
Core capabilities include:

- Add/Remove User: Manage followed Trainees
- Create/Edit/Delete WorkoutRecord: Manage exercise logs
- Create/Edit/Delete WorkoutPlan: Design training programs

System Functionality

The system provides three main functional areas:

- **User Management:**
  - User authentication (Sign-In)
  - Trainer associations (Add/Remove Personal Trainer)
  - Trainee administration (Add/Remove User)
- **Workout Management:**
  - Exercise definitions (CRUD Workout)
  - Training programs (CRUD WorkoutPlan, View WorkoutPlan)
- **Progress Tracking:**
  - Exercise logging (CRUD WorkoutRecord)
  - Progress monitoring (View UserWorkoutRecord)

Key Observations

- Trainers have broader privileges than trainees
- Only trainers can manage workout plans
- Both actors share some workout management functions

This diagram establishes the role-based access control for the fitness system, distinguishing between Trainee (consumer) and Personal Trainer (professional) roles.

3.1 Detailed Use Case Templates

The following templates outline some of the primary use cases identified for the project.

3.1.1 User Registration

Field	Description
Id	UC-1
Name	User Registration
Level	System Goal
Actors	User (Trainee or PT)
Pre-conditions	The user is not authenticated and does not have an account.
Post-conditions	The user has a profile (Trainee or PT specified), is authenticated, and has access to the personalized system interface.
Basic flow	1. User navigates to the registration page. 2. User enters required details (name, email, password, role - Trainee/PT). 3. System validates the input fields. 4. System creates a new user account. 5. System authenticates the user and redirects to their dashboard.
Alternative flow	• 3A. If validation errors occur, system displays an error message. • 4A. If email already exists, system displays an error message. • 5A. If a system error occurs, display an appropriate message.
Integration Test	5.3.2

Table 1: Use Case: User Registration

3.1.2 Sign-In

Field	Description
Id	UC-2
Name	Sign-In
Level	System Goal
Actors	User (Trainee or PT)
Pre-conditions	User is registered but not authenticated.
Post-conditions	User is authenticated and redirected to their personalized dashboard.
Basic flow	1. User navigates to the login page. 2. Enters email and password. 3. System authenticates credentials. 4. On success, user is redirected to their role-specific dashboard.
Alternative flow	• 3A. Incorrect credentials $\rightarrow$ display error message. • 4A. Account blocked or internal error $\rightarrow$ display appropriate notice.
Integration Test	5.3.2

Table 2: Use Case: Sign-In

3.1.3 Add Personal Trainer

Field	Description
Id	UC-3
Name	Add Personal Trainer

Field	Description
Level	User Goal
Actors	User (Trainee or PT)
Pre-conditions	The Trainee is authenticated and not already followed by the selected PT.
Post-conditions	The selected PT is now associated with the Trainee and can view their workout records.
Basic flow	1. Trainee navigates to "Manage Trainers" section. 2. Trainee selects a PT from the list of available trainers. 3. System verifies the PT exists and is not already linked. 4. System adds the PT to the Trainee's associated trainers.
Alternative flow	• 3A. If selected PT is invalid, show error. • 4A. If PT is already associated, system notifies user.
Integration Test	5.3.2

Table 3: Use Case: Add Personal Trainer

3.1.4 Remove Personal Trainer

Field	Description
Id	UC-4
Name	Remove Personal Trainer
Level	User Goal
Actors	User (Trainee or PT)
Pre-conditions	Trainee is authenticated and has at least one associated PT.
Post-conditions	The selected PT is removed from the Trainee's list.
Basic flow	1. Trainee opens "My Trainers". 2. Chooses a PT to remove. 3. System removes association.
Alternative flow	• 3A. PT not found $\rightarrow$ display error.
Integration Test	5.3.2

Table 4: Use Case: Remove Personal Trainer

3.1.5 View WorkoutRecord

Field	Description
Id	UC-5
Name	View WorkoutRecord
Level	User Goal
Actors	User (Trainee or Personal Trainer)
Pre-conditions	The user is authenticated. Trainee can only view their own records; PT can view records of followed trainees.
Post-conditions	The system displays a list of WorkoutRecords associated with the selected user (self or trainee).
Basic flow	1. User navigates to "Workout Records". 2. System retrieves all WorkoutRecords (each a logical session) associated with the user.

Field	Description
	3. User selects a WorkoutRecord to view detailed entries (Workout4Records).
Alternative flow	• 3A. If no records are found → display "no records available".
Integration Test	5.3.2

Table 5: Use Case: View WorkoutRecord

3.1.6 View Workout Plan

Field	Description
Id	UC-6
Name	View Workout Plan
Level	User Goal
Actors	User (Trainee or PT)
Pre-conditions	The Trainee is authenticated and has been assigned at least one workout plan by a PT.
Post-conditions	The user sees details of their current or past assigned workout plans.
Basic flow	1. Trainee navigates to "My Plans" section. 2. System retrieves all WorkoutPlans linked to the Trainee. 3. Trainee selects a plan to view. 4. System displays nested structure (Workout4Plans and Exercises).
Alternative flow	• 3A. If no plans found, system displays notification.
Integration Test	5.3.2

Table 6: Use Case: View Workout Plan

3.1.7 Register Workout(4Record)

Field	Description
Id	UC-7
Name	Register Workout(4Record)
Level	User Goal
Actors	User (Trainee or PT)
Pre-conditions	The Trainee is authenticated and has an assigned workout plan or intends to log an ad-hoc workout.
Post-conditions	The completed workout session (Workout4Record) is logged and associated with the User's workout records. PT may is notified and can view this record.
Basic flow	1. Trainee navigates to "Log Workout" section. 2. User selects date, optionally links to a Workout4Plan from their assigned plan. 3. User enters details for each exercise performed (actual sets, reps, weight). 4. System validates input. 5. System saves the Workout4Record.
Alternative flow	• 4A. If validation errors (e.g., invalid date), system displays an error. • 5A. If save fails, system displays an error message.
Integration Test	5.3.2

Field	Description

Table 7: Use Case: Upload Completed Workout

3.1.8 View Workout(4Record)

Field	Description
Id	UC-8
Name	View Workout(4Record)
Level	User Goal
Actors	User (Trainee or Personal Trainer)
Pre-conditions	User is authenticated and authorized to view the selected WorkoutRecord.
Post-conditions	System displays a list of Workout4Records associated with the chosen WorkoutRecord.
Basic flow	1. User navigates to "Workout Records". 2. Selects a specific WorkoutRecord to view. 3. System fetches and displays the list of Workout4Records for that date or session. 4. User clicks on one Workout4Record to see its exercises.
Alternative flow	• 3A. No records found → system displays a message.
Integration Test	5.3.2

Table 8: Use Case: View Workout(4Record)

3.1.9 Edit Workout(4Record)

Field	Description
Id	UC-9
Name	Edit Workout(4Record)
Level	User Goal
Actors	User (Trainee or Personal Trainer)
Pre-conditions	User is authenticated and owns or has permission to modify the record.
Post-conditions	The selected Workout4Record is updated in the database.
Basic flow	1. User selects a Workout4Record from their records. 2. Edits values such as sets, reps, weights, notes. 3. System validates the input. 4. System updates the record.
Alternative flow	• 3A. If invalid data entered → display validation errors. • If unauthorized → show permission error.
Integration Test	5.3.2

Table 9: Use Case: Edit Workout(4Record)

3.1.10 Delete Workout(4Record)

Field	Description
Id	UC-10
Name	Delete Workout(4Record)
Level	User Goal
Actors	User (Trainee or Personal Trainer)
Pre-conditions	The user is authenticated and owns/is allowed to delete the record.
Post-conditions	The selected Workout4Record is removed from the system.
Basic flow	1. User navigates to “Workout Records”. 2. Selects the Workout4Record to delete. 3. System asks for confirmation. 4. Upon confirmation, the record is deleted.
Alternative flow	• 3A. If the record doesn’t exist → show error.
Integration Test	5.3.2

Table 10: Use Case: Delete Workout(4Record)

3.1.11 Add Trainee (Follow User)

Field	Description
Id	UC-11
Name	Add Trainee
Level	User Goal
Actors	Personal Trainer
Pre-conditions	PT is authenticated and the trainee is not already followed.
Post-conditions	PT can access the trainee’s records.
Basic flow	1. PT navigates to “Manage Trainees”. 2. Selects a user to follow. 3. System links trainee to PT.
Alternative flow	• 3A. If user already followed → error.
Integration Test	5.3.2

Table 11: Use Case: Add Trainee

3.1.12 Remove Trainee (Unfollow)

Field	Description
Id	UC-12
Name	Remove Trainee (Unfollow)
Level	User Goal
Actors	Personal Trainer
Pre-conditions	The PT is authenticated and currently follows the selected trainee.
Post-conditions	The selected Trainee is removed from the PT’s followed list. PT can no longer access their data.
Basic flow	1. PT navigates to “Manage Trainees”. 2. PT selects the trainee to unfollow. 3. System confirms the action with a prompt. 4. Upon confirmation, system unlinks the trainee from the PT.

Field	Description
Alternative flow	• 3A. If the PT is not following the selected trainee → display error. • 4A. If the unlinking fails → display system error message.
Integration Test	5.3.2

Table 12: Use Case: Remove Trainee (Unfollow)

3.1.13 Create Workout Plan (PT)

Field	Description
Id	UC-13
Name	Create Workout Plan
Level	User Goal
Actors	Personal Trainer
Pre-conditions	The PT is authenticated in the system.
Post-conditions	A new workout plan is created and saved. The plan can then be assigned to trainees.
Basic flow	1. PT navigates to "Create Workout Plan" section. 2. PT enters plan details (name, description, adds Workout4Plan sessions with exercises). 3. For each exercise, PT specifies details (name, sets, reps, equipment, strategy e.g. Strength). 4. System validates the input. 5. System saves the workout plan.
Alternative flow	• 4A. If validation errors occur (e.g., missing plan name), system displays an error. • 5A. If an error occurs during saving, system displays an error message.
Integration Test	5.3.2

Table 13: Use Case: Create Workout Plan

3.1.14 Edit WorkoutPlan (Attach/Detach Workout4Plans)

Field	Description
Id	UC-14
Name	Edit WorkoutPlan (Attach/Detach Workout4Plans)
Level	User Goal
Actors	Personal Trainer
Pre-conditions	The PT is authenticated and is the owner of the WorkoutPlan. At least one Workout4Plan exists in the system.
Post-conditions	The selected Workout4Plans are either linked to or unlinked from the target WorkoutPlan.
Basic flow	1. PT navigates to "Edit WorkoutPlan" section. 2. PT selects an existing WorkoutPlan from their list. 3. System displays a list of available Workout4Plans. 4. PT selects which Workout4Plans to attach or detach. 5. System updates the relationships and refreshes the WorkoutPlan view.

Field	Description
Alternative flow	• If Workout4Plan is already attached → disable or warn. • 5A. If system error occurs while updating → show error message.
Integration Test	5.3.2

Table 14: Use Case: Edit WorkoutPlan (Attach/Detach Workout4Plans)

3.1.15 Delete WorkoutPlan

Field	Description
Id	UC-15
Name	Delete WorkoutPlan
Level	User Goal
Actors	Personal Trainer
Pre-conditions	PT is authenticated and owns the WorkoutPlan.
Post-conditions	The selected WorkoutPlan and its associations with Workout4Plans are removed.
Basic flow	1. PT navigates to "My Plans". 2. Selects a WorkoutPlan to delete. 3. Confirms deletion. 4. System deletes the WorkoutPlan and unlinks attached Workout4Plans.
Alternative flow	• 3A. Plan not found or not owned → error message.
Integration Test	5.3.2

Table 15: Use Case: Delete WorkoutPlan

3.1.16 Create Workout(4Plan)

Field	Description
Id	UC-16
Name	Create Workout(4Plan)
Level	User Goal
Actors	Personal Trainer
Pre-conditions	The PT is authenticated.
Post-conditions	A new Workout4Plan is created and stored for later attachment to one or more WorkoutPlans.
Basic flow	1. PT navigates to "Create Workout(4Plan)". 2. Fills in fields: day, strategy, list of exercises. 3. System validates and creates the Workout4Plan.
Alternative flow	• 3A. Validation errors → display messages.
Integration Test	5.3.2

Table 16: Use Case: Create Workout(4Plan)

3.1.17 View Workout(4Plan)

Field	Description
Id	UC-17
Name	View Workout(4Plan)
Level	User Goal
Actors	Personal Trainer
Pre-conditions	PT is authenticated. At least one Workout4Plan exists.
Post-conditions	The PT sees the list of created Workout4Plans and their details.
Basic flow	1. PT navigates to "Workout(4Plans)". 2. System displays all Workout4Plans owned by the PT. 3. PT selects a Workout4Plan to view its exercises.
Alternative flow	• 2A. No plans available → display message.
Integration Test	5.3.2

Table 17: Use Case: View Workout(4Plan)

3.1.18 Edit Workout(4Plan)

Field	Description
Id	UC-18
Name	Edit Workout(4Plan)
Level	User Goal
Actors	Personal Trainer
Pre-conditions	PT is authenticated and owns the Workout4Plan.
Post-conditions	The Workout4Plan is updated with new exercise or strategy data.
Basic flow	1. PT navigates to "My Workout(4Plans)". 2. Selects a Workout4Plan to edit. 3. Modifies fields: strategy, exercises, sets, reps. 4. System validates and saves changes.
Alternative flow	• 3A. Workout4Plan not found or unauthorized → show error.
Integration Test	5.3.2

Table 18: Use Case: Edit Workout(4Plan)

3.1.19 Delete Workout(4Plan)

Field	Description
Id	UC-19
Name	Delete Workout(4Plan)
Level	User Goal
Actors	Personal Trainer
Pre-conditions	PT is authenticated and owns the Workout4Plan.
Post-conditions	The Workout4Plan is permanently deleted and detached from all WorkoutPlans.
Basic flow	1. PT opens list of Workout4Plans. 2. Selects one to delete. 3. Confirms deletion. 4. System removes the entry and updates all affected WorkoutPlans.

Field	Description
Alternative flow	• 3A. Cannot delete if in use → display constraint error.
Integration Test	5.3.2

Table 19: Use Case: Delete Workout(4Plan)

3.1.20 View User Workout Record (PT)

Field	Description
Id	UC-20
Name	View Trainee Workout Record
Level	User Goal
Actors	Personal Trainer
Pre-conditions	PT is authenticated and currently follows the selected trainee.
Post-conditions	PT sees a chronological list of the trainee's WorkoutRecords.
Basic flow	1. PT navigates to "Trainee Progress". 2. Selects a trainee from followed list. 3. System retrieves all WorkoutRecords associated with that user. 4. PT browses and optionally drills into individual sessions.
Alternative flow	• 3A. If trainee not followed → show error.
Integration Test	5.3.2

Table 20: Use Case: View User Workout Record (PT)

3.2 Graphical User Interface Mockups

While the current version of Java-TrainingHub is focused on backend logic, conceptual mockups have been designed to illustrate the user flow for a planned Graphical User Interface (GUI). These mockups outline the core functionalities from both the personal trainer and trainee perspectives, showcasing a clean, modern design with a dark theme and gold accent colors. The following figures present the key screens of the application with detailed descriptions of their functionality and design elements.

The authentication interface features the application branding "TRAINING-HUB" with the tagline "Your fitness journey starts here". The screen provides multiple authentication options including traditional email and password fields, social sign-on buttons for Google and Apple integration, and a "Forgot Password?" recovery link. New users can access registration through the "Sign Up" link at the bottom. The design maintains consistency with the dark theme and uses gold accenting for interactive elements.

3.2.1 Trainee Interface Mockups

This interface allows trainees to browse and select from available personal trainers. Each trainer profile displays essential information including their profile photo, full name, years of experience, and star rating. Specialization tags highlight their expertise areas (e.g., "Strength", "HIIT", "Boxing" for Marcus Johnson; "Yoga", "Pilates", "Flexibility" for Sarah Anderson). Each profile includes "Profile" and "Select" buttons for detailed viewing and trainer selection. The screen shows a curated list of qualified trainers with their credentials clearly displayed.

This interface displays the trainee's assigned workout program in a weekly calendar format. The screen shows a "3-Day Split Program" labeled as "HYPERTROPHY" and created by "Marcus Johnson", last updated on "Jun 2, 2025". The weekly schedule presents each day with specific workout details: Monday shows "Push Hypertrophy" (5 exercises, 45-60 minutes) marked as "TODAY" with an active "Show Workout" button; Tuesday displays "Rest Day" with recovery recommendations; Wednesday features "Pull Hypertrophy" (4 exercises, 50-65 minutes). The color-coded system (gold for today, green for rest days) helps trainees easily identify their current training schedule and navigate their program effectively.

This interface enables trainees to log their completed workouts with detailed exercise information. The screen is dated "Monday, June 5, 2025" and features an "Add Exercise" section where users can input exercise names (shown: "Push-Up"), specify the number of sets (4), and record repetitions (8) with increment/decrement controls. The bottom section displays "Today's Exercises" showing previously logged activities like "Bench Press" with "3 sets × 10 reps" and a delete option. This systematic approach allows trainees to maintain accurate workout records for progress tracking and trainer review.

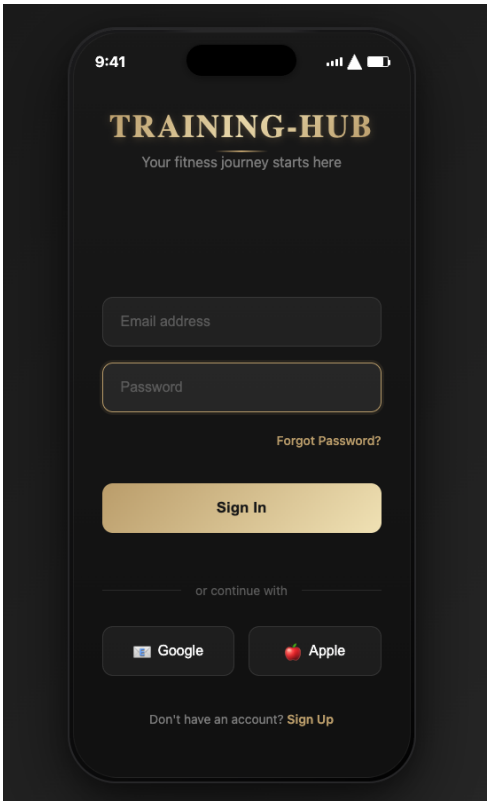


Figure 2: Login Screen

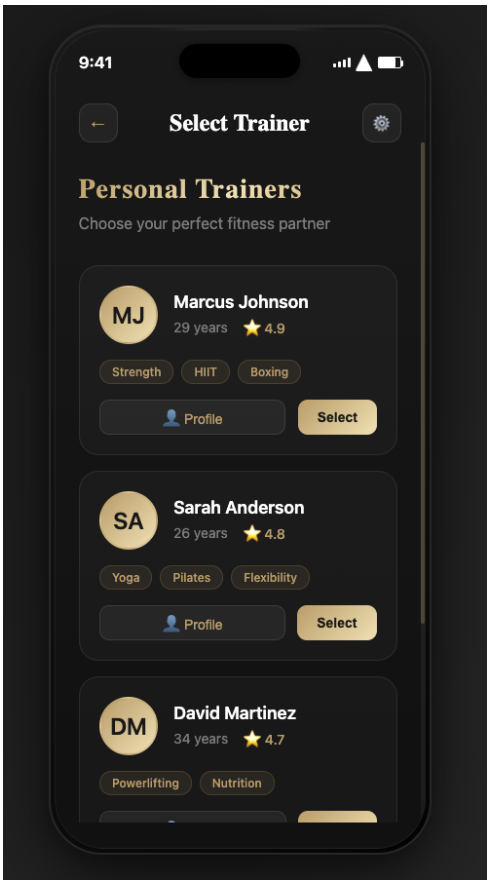


Figure 3: Select Trainer Screen

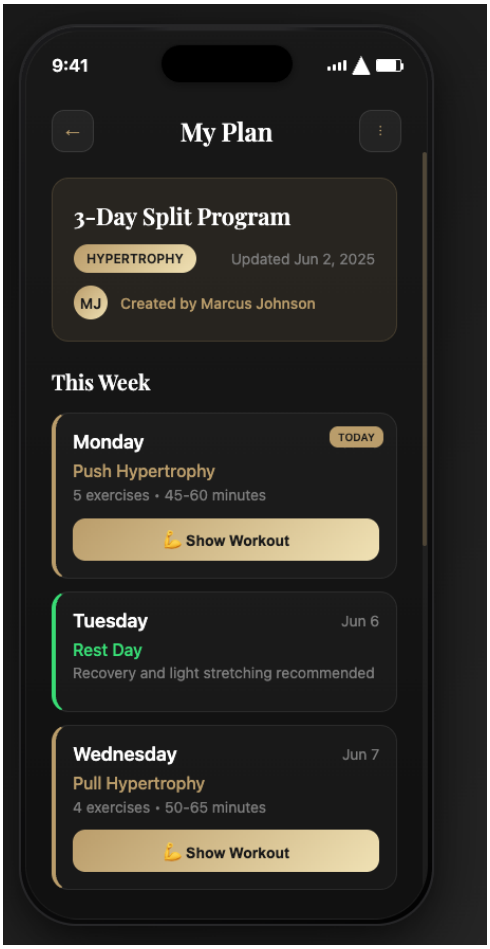


Figure 4: My Plan Screen

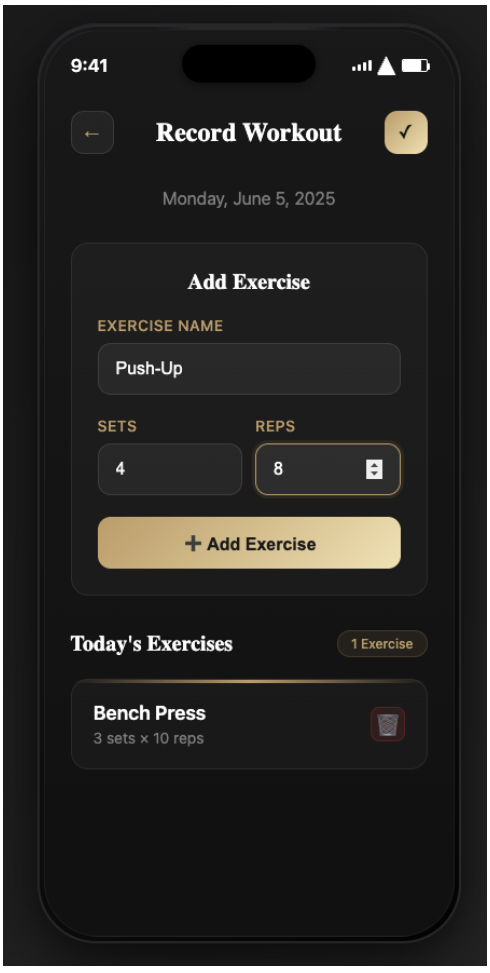


Figure 5: Record Workout Screen

3.2.2 Personal Trainer Interface Mockups

This interface enables personal trainers to add new trainees to their client roster. The screen features a search functionality at the top for finding "registered users" and displays an "Available Users" section showing 4 potential trainees. Each user entry includes their profile initials, full name, age, and action buttons for "View Profile" and "Add". The interface allows trainers to review potential clients before establishing the trainer-trainee relationship, with Emma Davis currently highlighted as selected for demonstration purposes.

This comprehensive interface allows trainers to design detailed workout plans for their trainees. The screen shows it's "Creating plan for Sarah Johnson" and includes fields for plan naming ("3-Day Split Program"), intensity level selection, and weekly schedule configuration. The Monday section is expanded, showing "Upper Body Focus" with 5 exercises listed (Bench Press, Pull-ups, Shoulder Press...) and a duration of 45-60 minutes. The interface includes toggle switches for each day of the week and "View Workout" buttons for detailed exercise configuration. This demonstrates the plan's structured approach to weekly training organization.

This monitoring dashboard allows trainers to track their trainees' progress and activity. The interface features a search bar for finding specific trainees and displays detailed statistics for each client. For Sarah Johnson, it shows her active status (last workout 2 days ago), key metrics (24 total workouts, 8 this month, 3.2 average per week), and recent workout history including Push Strength, Pull Strength, and Legs Strength sessions with dates. Emma Davis is shown as inactive (last workout 1 week ago) with lower activity metrics. The "View Full Records" button provides access to comprehensive workout data, enabling trainers to monitor client engagement and progress effectively.

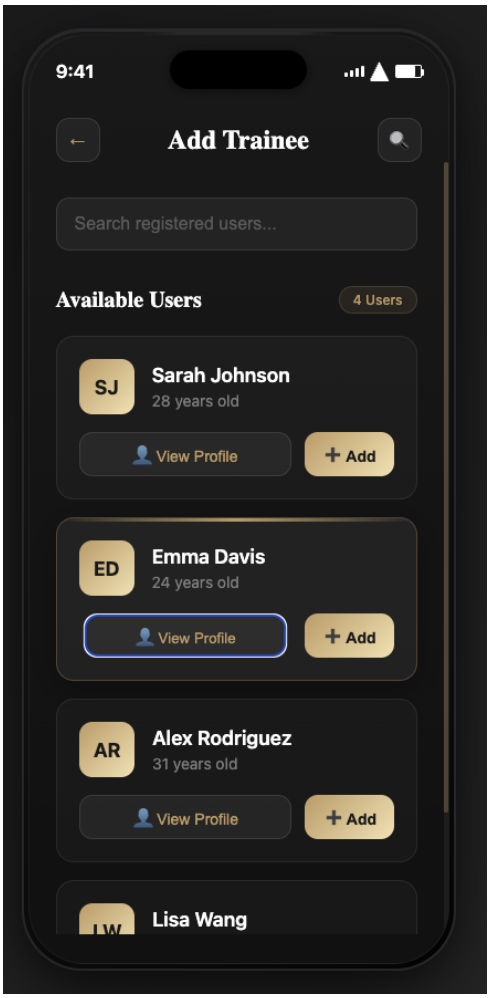


Figure 6: Add Trainee Screen

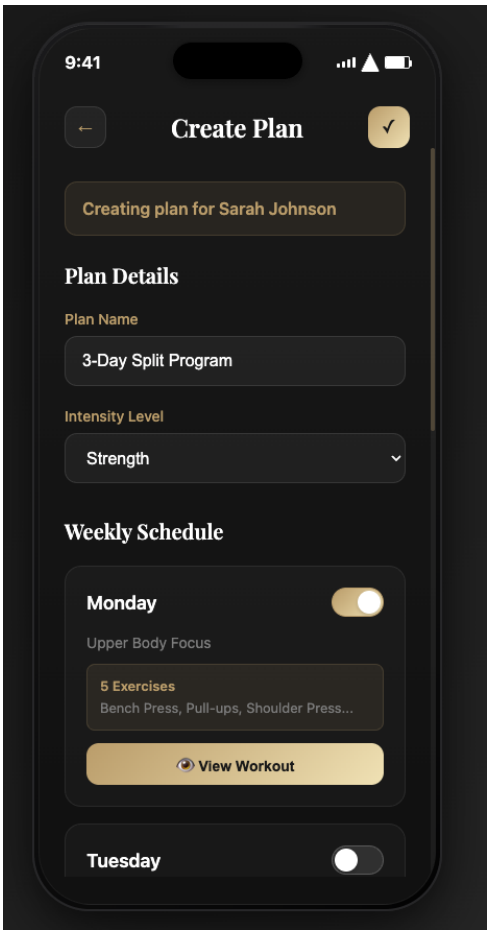


Figure 7: Create Plan Screen

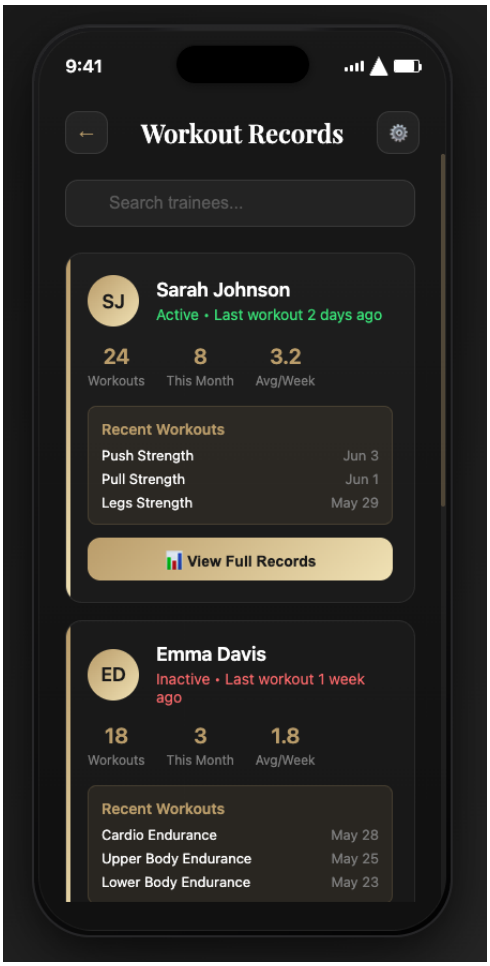


Figure 8: Workout Records Screen

3.2.3 Design Consistency and User Experience

All mockups maintain a consistent dark theme with gold accent colors, ensuring visual coherence across the application. The interface design prioritizes readability with clear typography, appropriate spacing, and intuitive navigation patterns. Interactive elements such as buttons, toggles, and input fields are consistently styled with the signature gold coloring to provide clear visual feedback. The layout structure remains consistent across screens, with standard navigation elements (back buttons, settings icons) positioned uniformly to enhance user familiarity and ease of use.

The mockups demonstrate a user-centered design approach, with clear information hierarchy, logical grouping of related functions, and prominent call-to-action buttons. The interface successfully balances functionality with aesthetics, providing comprehensive features while maintaining an uncluttered, professional appearance suitable for both personal trainers and their clients. The workflow progression from trainer selection through plan creation to workout tracking creates an intuitive user experience that supports the application's core mission of facilitating effective fitness training relationships.

3.3 Database Conceptual Data Modeling

The conceptual design of the Java-TrainingHub database is represented through an Entity-Relationship (ER) model that captures the core functionality of a comprehensive fitness management system. This model defines the entities, their attributes, and the relationships that enable users to create workout plans, log workout sessions, and facilitate personal trainer-client interactions.

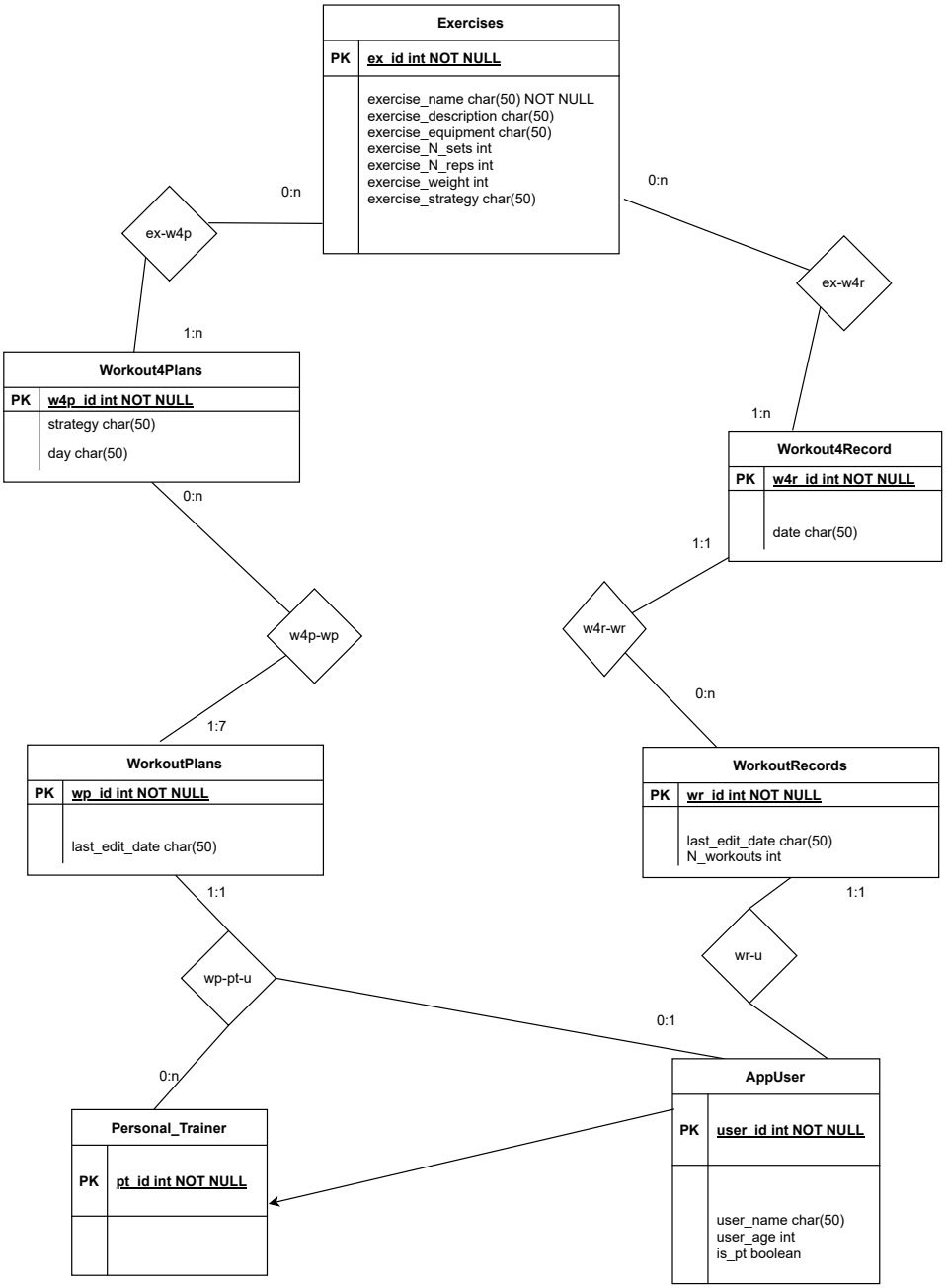


Figure 9: Entity-Relationship Model for Java-TrainingHub

3.3.1 Entity Descriptions

The ER model (Figure 9) encompasses six primary entities that collectively support the system's functionality:

AppUser Entity The **AppUser** entity serves as the central user management component, storing essential information for all system users regardless of their role. This entity includes:

- `user_id` (PK, int, NOT NULL): Unique identifier for each user
- `user_name` (char(50)): Display name for the user
- `user_age` (int): User's age for fitness planning considerations
- `is_pt` (boolean): Role distinguisher flag indicating whether the user is a Personal Trainer (true) or regular user/trainee (false)

Personal_Trainer Entity The **Personal_Trainer** entity extends user functionality specifically for certified trainers within the system:

- `pt_id` (PK, int, NOT NULL): Unique identifier for personal trainer records

This entity maintains a 1:1 relationship with AppUser, ensuring that personal trainers have both general user attributes and specialized trainer-specific data.

WorkoutPlans Entity The **WorkoutPlans** entity represents collections of planned workout sessions designed for future execution:

- `wp_id` (PK, int, NOT NULL): Unique identifier for each workout plan
- `last_edit_date` (char(50)): Timestamp tracking the most recent modification to the plan

Workout4Plans Entity The **Workout4Plans** entity details individual workout sessions within a larger workout plan:

- `w4p_id` (PK, int, NOT NULL): Unique identifier for each planned workout session
- `strategy` (char(50)): Training approach or methodology for the session
- `day` (char(50)): Scheduled day or timing information for the workout

WorkoutRecords Entity The **WorkoutRecords** entity maintains historical collections of completed workout sessions:

- `wr_id` (PK, int, NOT NULL): Unique identifier for each workout record collection
- `last_edit_date` (char(50)): Timestamp of the most recent update to the record
- `N_workouts` (int): Count of total workout sessions within this record collection

Workout4Record Entity The **Workout4Record** entity captures individual completed workout sessions:

- `w4r_id` (PK, int, NOT NULL): Unique identifier for each recorded workout session
- `date` (char(50)): Date when the workout was performed

Exercises Entity The **Exercises** entity serves as the comprehensive exercise library, containing detailed information about individual exercises:

- `ex_id` (PK, int, NOT NULL): Unique identifier for each exercise
- `exercise_name` (char(50), NOT NULL): Descriptive name of the exercise
- `exercise_description` (char(50)): Detailed explanation or instructions for the exercise
- `exercise_equipment` (char(50)): Required equipment or apparatus for the exercise
- `exercise_N_sets` (int): Recommended or recorded number of sets
- `exercise_N_reps` (int): Recommended or recorded number of repetitions per set
- `exercise_weight` (int): Weight used or recommended for the exercise
- `exercise_strategy` (char(50)): Training strategy or approach associated with the exercise

3.3.2 Relationship Analysis

The database design implements several critical relationships that enable comprehensive fitness management functionality:

User-Centric Relationships

- **AppUser to Personal_Trainer (1:1):** Establishes specialized trainer profiles while maintaining core user attributes
- **AppUser to WorkoutRecords (1:1):** Ensures each user maintains their own workout history
- **Personal_Trainer to WorkoutPlans (wp-pt-u relationship):** Enables personal trainers to create and manage workout plans for their clients, supporting the 1:n relationship where one trainer can manage multiple plans

Plan and Record Structure

- **WorkoutPlans to Workout4Plans (1:7):** Suggests a weekly structure where each workout plan can contain up to seven individual workout sessions (one per day)
- **WorkoutRecords to Workout4Record (1:n):** Allows users to maintain multiple recorded workout sessions within their exercise history

Exercise Integration

- **Exercises to Workout4Plans (ex-w4p, n:n):** Enables flexible assignment of multiple exercises to planned workout sessions, with each exercise potentially used across multiple workout plans
- **Exercises to Workout4Record (ex-w4r, n:n):** Allows recording of multiple exercises within completed workout sessions, supporting comprehensive workout logging

3.3.3 Design Rationale and Functionality

This database design supports several key functional requirements:

Role-Based Access Control The combination of the `is_pt` flag in AppUser and the separate Personal_Trainer entity enables role-based functionality while maintaining a unified user base. This design allows for efficient user management and specialized features for personal trainers.

Flexible Workout Management The separation of workout plans (future/scheduled) and workout records (completed/historical) provides clear distinction between intended workouts and actual performance, enabling users to track adherence to their fitness plans.

Comprehensive Exercise Library The detailed Exercises entity supports advanced workout planning by capturing not only basic exercise information but also specific parameters like sets, reps, weight, and training strategies, enabling personalized and progressive workout design.

Scalable Relationship Structure The many-to-many relationships between exercises and both planned and recorded workouts provide flexibility for complex workout routines while avoiding data duplication. This design supports both simple and sophisticated training programs.

Audit Trail Capabilities The inclusion of date and edit tracking fields enables the system to maintain historical records and support features like progress tracking, plan versioning, and workout scheduling.

This comprehensive database design provides the foundation for a robust fitness management system that can accommodate various user types, support complex workout planning and tracking, and maintain detailed exercise and performance records.

4 Implementation Details

During the implementation phase, the use cases were brought to life, starting with the design of the domain model. Next, the business logic was designed considering the overall architecture. Coding was implemented using the Java programming language (Version 11 or as specified in `pom.xml`).

4.1 Project Dependencies

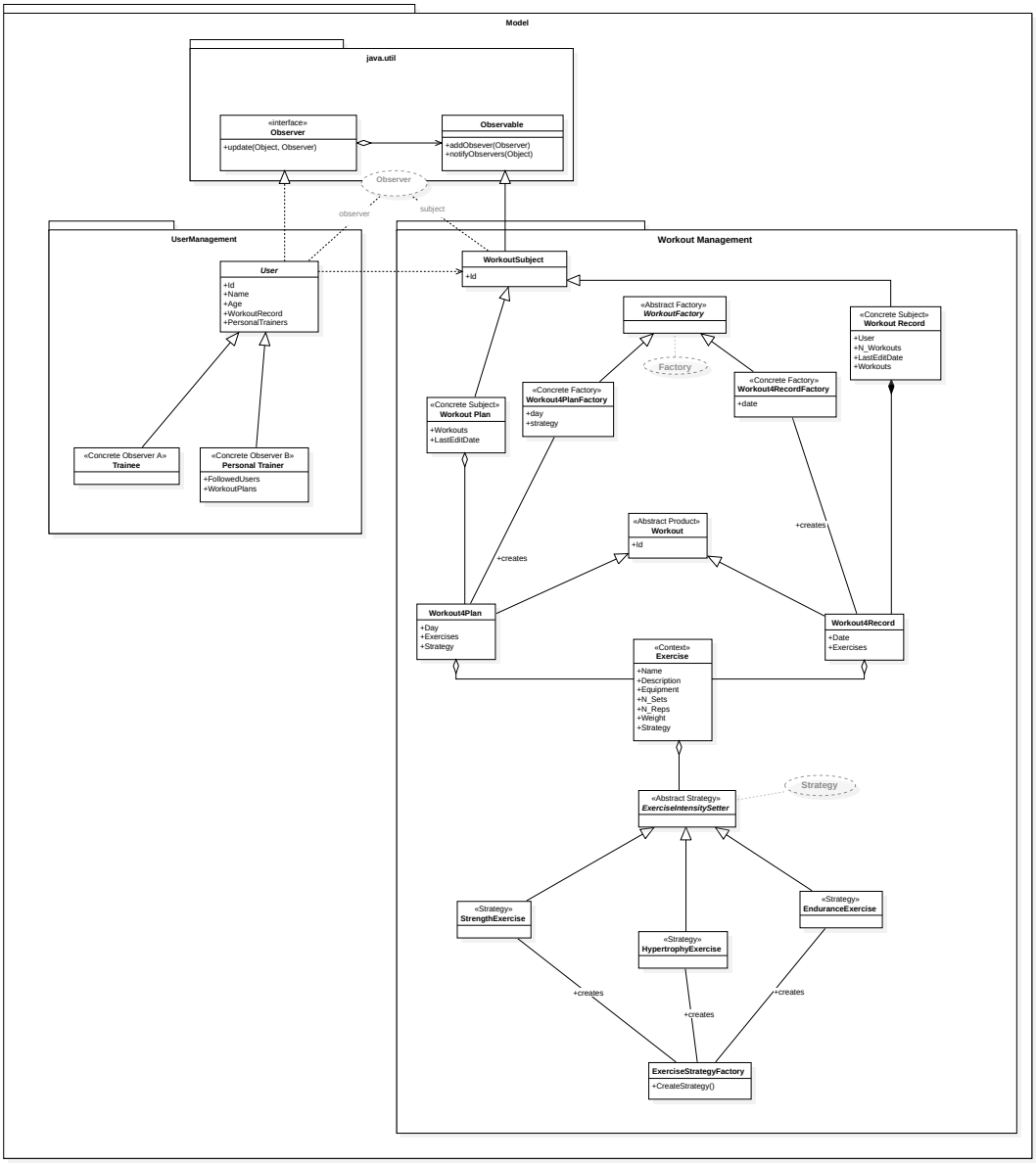
The project relies on several dependencies, managed through Maven and defined in the `pom.xml` file. Below is an overview of the main technologies used:

- **Language:** Java (Version 11 or as specified in `pom.xml`).
- **Build Tool & Dependency Management:** Apache Maven.
- **Database:** PostgreSQL.
- **Testing Frameworks:**
  - JUnit 5 (for unit and integration testing).
  - Mockito (for creating mock objects in tests).

4.2 System Architecture Overview

The Java-TrainingHub system follows a layered architecture pattern that promotes separation of concerns, maintainability, and scalability. The complete system architecture is presented in the comprehensive Class Diagram (Figure 10), which illustrates the main classes, their attributes, methods, relationships, and the implementation of several design patterns.

The architecture is organized into three distinct logical layers: the Model layer (domain objects and business logic), the ORM layer (data access and persistence), and the Business layer (application controllers and services). This layered approach ensures clear separation between business logic, data access, and application control flow.


Model
java.util
«interface»
Observer
+update(Object, Observer)
Observable
+addObserver(Observer)
+notifyObservers(Object)
Observer
observer
subject
UserManagement
User
+Id
+Name
+Age
+WorkoutRecord
+PersonalTrainers
«Concrete Observer A»
Trainee
«Concrete Observer B»
Personal Trainer
+FollowedUsers
+WorkoutPlans
Workout Management
WorkoutSubject
+Id
«Abstract Factory»
WorkoutFactory
Factory
«Concrete Factory»
Workout4PlanFactory
+day
+strategy
«Concrete Factory»
Workout4RecordFactory
+date
«Concrete Subject»
Workout Plan
+Workouts
+LastEditDate
«Concrete Subject»
Workout Record
+User
+N_Workouts
+LastEditDate
+Workouts
«Abstract Product»
Workout
+Id
+creates
+creates
+creates
Workout4Plan
+Day
+Exercises
+Strategy
Workout4Record
+Date
+Exercises
«Context»
Exercise
+Name
+Description
+Equipment
+N_Sets
+N_Reps
+Weight
+Strategy
Strategy
«Abstract Strategy»
ExerciseIntensitySetter
«Strategy»
StrengthExercise
«Strategy»
HypertrophyExercise
«Strategy»
EnduranceExercise
+creates
+creates
+creates
ExerciseStrategyFactory
+CreateStrategy()
ORM
ConcreteExerciseDAO
+connection
+addExercise()
+addExercise4Plan()
+getAllExercise()
+getExercisebyId()
+updateExercise()
+deleteExercise()
ConcreteTraineeDAO
+connection
+addTrainee()
+getAllTrainees()
+getWorkoutPlanFromUserId()
+getPTForUserId()
+editUser()
+deleteUser()
+getWorkoutRecordByUserId()
ConcretePersonalTrainerDAO
+connection
+addPersonalTrainer()
+getAllPersonalTrainers()
+getPersonalTrainerById()
+getPlansMadeByPT()
+getTraineesOfPT()
+editPersonalTrainer()
+deletePersonalTrainer()
+isFollowing()
ConcreteWorkoutPlanDAO
+connection
+addWorkoutPlan()
+getAllWorkoutPlans()
+getWorkoutPlanById()
+updateLastEditDate()
+deleteWorkoutPlan()
+getWorkout4PlansByWorkoutPlanId()
+addWorkout4PlanToWorkoutPlan()
+removeWorkout4PlanFromWorkoutPlan()
+assignWorkoutPlanToUser()
+unassignWorkoutPlan()
+getWorkoutPlansByTraineeId()
ConcreteWorkoutRecordDAO
+connection
+addWorkoutRecord()
+linkWorkoutRecordToUser()
+getWorkoutRecordByUserId()
+getWorkoutRecordById()
+getAllWorkoutRecords()
+deleteWorkoutRecord()
+getWorkout4RecordsByWorkoutRecordId()
+addWorkout4RecordToWorkoutRecord()
ConcreteWorkout4PlanDAO
+connection
+addWorkout4Plan()
+getAllWorkout4Plans()
+getWorkout4PlanById()
+updateWorkout4Plan()
+deleteWorkout4Plan()
+addExerciseToWorkout4Plan()
+getExercisesForWorkout4Plan()
ConcreteWorkout4RecordDAO
+connection
+addWorkout4Record()
+getAllWorkout4Records()
+getWorkout4RecordById()
+updateWorkout4RecordDate()
+deleteWorkout4Record()
+addExerciseToWorkout4Record()
+getExercisesForWorkout4Record()
+implements
+implements
+implements
+implements
+implements
+implements
+implements
«interface»
ExerciseDAO
+connection
+addExercise()
+addExercise4Plan()
+getAllExercise()
+getExercisebyId()
+updateExercise()
+deleteExercise()
«interface»
TraineeDAO
+connection
+addTrainee()
+getAllTrainees()
+getWorkoutPlanFromUserId()
+getPTForUserId()
+editUser()
+deleteUser()
+getWorkoutRecordByUserId()
+Operation1()
«interface»
PersonalTrainerDAO
+connection
+addPersonalTrainer()
+getAllPersonalTrainers()
+getPersonalTrainerById()
+getPlansMadeByPT()
+getTraineesOfPT()
+editPersonalTrainer()
+deletePersonalTrainer()
+isFollowing()
«interface»
WorkoutPlanDAO
+connection
+addWorkoutPlan()
+getAllWorkoutPlans()
+Operation2()
+getWorkoutPlanById()
+updateLastEditDate()
+deleteWorkoutPlan()
+getWorkout4PlansByWorkoutPlanId()
+addWorkout4PlanToWorkoutPlan()
+removeWorkout4PlanFromWorkoutPlan()
+assignWorkoutPlanToUser()
+unassignWorkoutPlan()
+getWorkoutPlansByTraineeId()
«interface»
WorkoutRecordDAO
+connection
+addWorkoutRecord()
+linkWorkoutRecordToUser()
+getWorkoutRecordByUserId()
+getWorkoutRecordById()
+getAllWorkoutRecords()
+deleteWorkoutRecord()
+getWorkout4RecordsByWorkoutRecordId()
+addWorkout4RecordToWorkoutRecord()
«interface»
Workout4PlanDAO
+connection
+addWorkout4Plan()
+getAllWorkout4Plans()
+getWorkout4PlanById()
+updateWorkout4Plan()
+deleteWorkout4Plan()
+addExerciseToWorkout4Plan()
+getExercisesForWorkout4Plan()
«interface»
Workout4RecordDAO
+connection
+addWorkout4Record()
+getAllWorkout4Records()
+getWorkout4RecordById()
+updateWorkout4RecordDate()
+deleteWorkout4Record()
+addExerciseToWorkout4Record()
+getExercisesForWorkout4Record()
DBManager
+connection
+getConnection()
+closeConnection()
+istantiate
+istantiate
+istantiate
+istantiate
+istantiate
+istantiate
+istantiate
Business
ExerciseController
+exerciseDAO
+createExercise()
+createExerciseForPlan()
+getAllExercises()
+getExerciseById()
+updateExercise()
+deleteExercise()
TraineeController
+traineeDAO
+workoutRecordDAO
+workoutPlanDAO
+registerTrainee()
+getAllTrainees()
+getWorkoutPlansByTraineeId()
+getPersonalTrainerForTrainee()
+updateTrainee()
+deleteTrainee()
+getWorkoutRecordByTraineeId()
PersonalTrainerController
+personalTrainerDAO
+workoutPlanDAO
+workoutRecordDAO
+registerPersonalTrainer()
+followTrainee()
+getAllPersonalTrainers()
+getPersonalTrainerById()
+updatePersonalTrainer()
+deletePersonalTrainer()
+getTraineesOfPT()
+getPlansMadeByPT()
+getWorkoutRecordForTrainee()
WorkoutPlanController
+workoutPlanDAO
+createWorkoutPlan()
+getAllWorkoutPlans()
+getWorkoutPlanById()
+deleteWorkoutPlan()
+addWorkout4PlanToWorkoutPlan()
WorkoutRecordController
+workoutRecordDAO
+createWorkoutRecord()
+getWorkoutRecordById()
+getAllWorkoutRecords()
+deleteWorkoutRecord()
+linkWorkout4RecordToWorkoutRecord()
+getWorkout4RecordsForWorkoutRecord()
Workout4PlanController
+workout4PlanDAO
+createWorkout4Plan()
+getAllWorkout4Plans()
+getWorkout4PlanById()
+updateWorkout4Plan()
+deleteWorkout4Plan()
+addExerciseToWorkout4Plan()
+getExercisesForWorkout4Plan()
Workout4RecordController
+workout4RecordDAO
+createWorkout4Record()
+getAllWorkout4Records()
+getWorkout4RecordById()
+updateWorkout4RecordDate()
+deleteWorkout4Record()
+addExerciseToWorkout4Record()
+getExercisesForWorkout4Record()
Entity Endpoints

4.2.1 Model Layer Architecture

The Model layer represents the core domain of the fitness management system, implementing key business entities and design patterns that support flexible workout planning and execution.

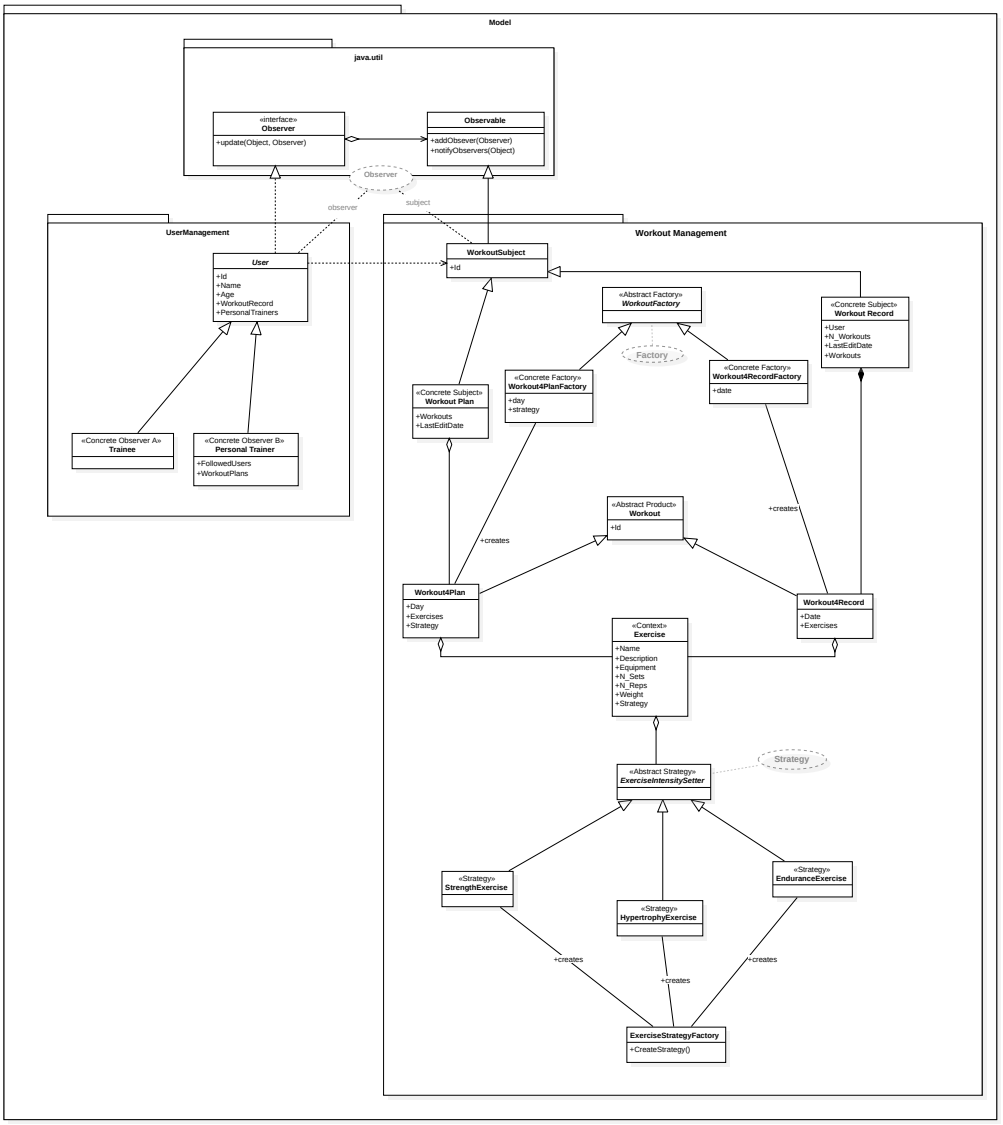


Figure 11: Model Layer - Domain Objects and Design Patterns

User Management Subsystem The user management subsystem implements a hierarchical structure with polymorphic behavior:

- **User (Base Class):** Contains core attributes including `Id`, `Name`, `Age`, `WorkoutRecord`, and `PersonalTrainers`. This serves as the foundation for all user types in the system.
- **Trainee (Concrete Observer A):** Extends the User class and implements the Observer pattern interface. Trainees receive notifications when their workout plans are modified by personal trainers.
- **Personal Trainer (Concrete Observer B):** Also extends User and implements Observer pattern. Contains additional attributes `FollowedUsers` and `WorkoutPlans` to manage client relationships and created workout plans.

Workout Management Subsystem The workout management subsystem employs multiple design patterns to provide flexible workout creation and tracking:

- **WorkoutSubject (Subject Interface):** Defines the contract for observable objects, containing `Observers` collection and `Id` attribute.
- **Workout Record (Concrete Subject):** Implements the Subject interface for tracking completed workouts. Attributes include `User`, `N_Workouts`, `LastEditDate`, and `Workouts`.
- **Workout Plan (Concrete Subject):** Another Subject implementation for planned workouts, containing `Workouts` and `LastEditDate`.
- **Workout4Record:** Represents individual completed workout sessions with `Date` and `Exercises` attributes.
- **Workout4Plan:** Represents individual planned workout sessions with `Day`, `Exercises`, and `Strategy` attributes.

Design Pattern Implementation

Observer Pattern Implementation: The system implements the Observer pattern to enable automatic notifications when workout plans are updated:

- **Observer (Abstract Observer):** Defines the interface for objects that should be notified of changes
- **WorkoutSubject (Subject Interface):** Maintains a list of observers and provides methods for notification
- **Trainee and Personal Trainer:** Act as concrete observers that receive notifications about workout plan changes

Factory Pattern Implementation: Multiple factory patterns are employed for flexible object creation:

- **WorkoutFactory (Abstract Factory):** Provides the interface for creating workout-related objects
- **Workout4PlanFactory (Concrete Factory):** Specializes in creating planned workout sessions with `day` and `strategy` parameters
- **Workout4RecordFactory (Concrete Factory):** Specializes in creating recorded workout sessions with `date` parameter
- **Workout (Abstract Product):** Serves as the base for all workout objects with an `Id` attribute

Strategy Pattern Implementation: The Strategy pattern enables flexible exercise intensity management:

- **ExerciseIntensitySetter (Abstract Strategy):** Defines the interface for different exercise intensity approaches
- **StrengthExercise (Concrete Strategy):** Implements strength-focused exercise parameters
- **HypertrophyExercise (Concrete Strategy):** Implements muscle-building focused exercise parameters
- **EnduranceExercise (Concrete Strategy):** Implements endurance-focused exercise parameters
- **ExerciseStrategyFactory:** Provides `CreateStrategy()` method for instantiating appropriate strategies

Exercise Management The **Exercise (Context)** class serves as the context for the Strategy pattern and contains comprehensive exercise information:

- Basic attributes: `Name`, `Description`, `Equipment`
- Performance parameters: `N_Sets`, `N_Reps`, `Weight`
- Strategy reference: `Strategy` for dynamic intensity management

4.2.2 Business Layer Architecture

The Business layer implements the application's control logic through a series of specialized controllers that orchestrate operations between the Model and ORM layers.

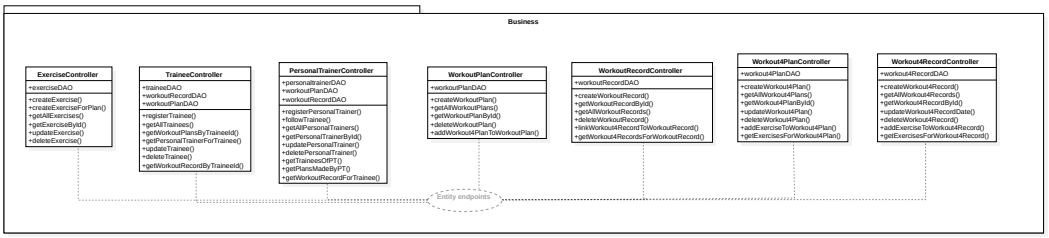


Figure 12: Business Layer - Application Controllers

Controller Architecture Each controller follows the same architectural pattern, maintaining references to necessary DAO objects and providing business logic methods:

ExerciseController:

- Dependencies: `exerciseDAO`
- Core operations: `createExercise()`, `createExerciseForPlan()`, `getAllExercises()`
- Management operations: `getExerciseById()`, `updateExercise()`, `deleteExercise()`

TraineeController:

- Dependencies: `traineeDAO`, `workoutRecordDAO`, `workoutPlanDAO`
- User management: `registerTrainee()`, `getAllTrainees()`, `updateTrainee()`, `deleteTrainee`
- Workout operations: `getWorkoutPlansByTraineeId()`, `getWorkoutRecordByTraineeId()`
- Relationship management: `getPersonalTrainerForTrainee()`

PersonalTrainerController:

- Dependencies: `personaltrainerDAO`, `workoutPlanDAO`, `workoutRecordDAO`
- Trainer management: `registerPersonalTrainer()`, `getAllPersonalTrainers()`, `getPersonalTrainerById()`
- Client relationship: `followTrainee()`, `getTraineesOfPT()`
- Workout management: `getPlansMadeByPT()`, `getWorkoutRecordForTrainee()`
- Administrative: `updatePersonalTrainer()`, `deletePersonalTrainer()`

WorkoutPlanController:

- Dependencies: `workoutPlanDAO`
- Plan management: `createWorkoutPlan()`, `getAllWorkoutPlans()`, `getWorkoutPlanById()`, `deleteWorkoutPlan()`
- Composition management: `addWorkout4PlanToWorkoutPlan()`

WorkoutRecordController:

- Dependencies: `workoutRecordDAO`
- Record management: `createWorkoutRecord()`, `getWorkoutRecordById()`, `getAllWorkoutReco` `deleteWorkoutRecord()`
- Session management: `linkWorkout4RecordToWorkoutRecord()`, `getWorkout4RecordsForWorko`

Workout4RecordController:

- Dependencies: `workout4RecordDAO`
- Session management: `createWorkout4Record()`, `getAllWorkout4Records()`, `getWorkout4Reco`
- Modification operations: `updateWorkout4RecordDate()`, `deleteWorkout4Record()`
- Exercise integration: `addExerciseToWorkout4Record()`, `getExercisesForWorkout4Record()`

Workout4PlanController:

- Dependencies: `workout4PlanDAO`
- Planning operations: `createWorkout4Plan()`, `getAllWorkout4Plans()`, `getWorkout4PlanById`
- Plan management: `updateWorkout4Plan()`, `deleteWorkout4Plan()`
- Exercise integration: `addExerciseToWorkout4Plan()`, `getExercisesForWorkout4Plan()`

4.2.3 ORM Layer Architecture

The ORM (Object-Relational Mapping) layer provides a clean abstraction between the business logic and the database, implementing the Data Access Object (DAO) pattern with interface-based design.

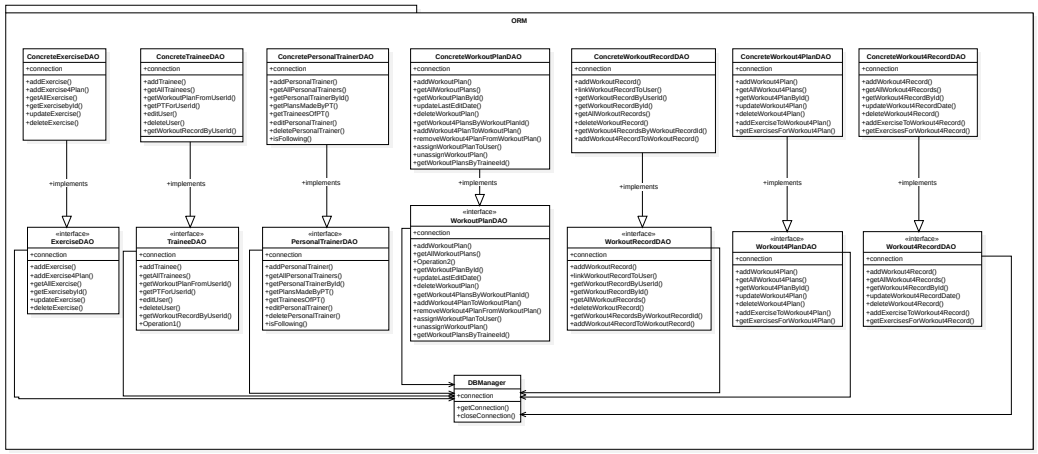


Figure 13: ORM Layer - Data Access Objects and Database Management

DAO Pattern Implementation The ORM layer follows a consistent pattern where each entity has both an interface and a concrete implementation, promoting loose coupling and testability:

Database Management:

- **DBManager:** Centralized database connection management with `connection` attribute and methods `getConnection()` and `closeConnection()`

User Data Access:

- **TraineeDAO Interface & ConcreteTraineeDAO:** Manages trainee-specific operations including registration, profile management, and workout plan/record retrieval
- **PersonalTrainerDAO Interface & ConcretePersonalTrainerDAO:** Handles personal trainer operations, client relationships, and plan management

Workout Data Access:

- **WorkoutPlanDAO Interface & ConcreteWorkoutPlanDAO:** Manages workout plan lifecycle, plan-session relationships, and user assignments
- **WorkoutRecordDAO Interface & ConcreteWorkoutRecordDAO:** Handles workout record persistence and user linkage
- **Workout4PlanDAO Interface & ConcreteWorkout4PlanDAO:** Manages individual planned workout sessions and exercise associations
- **Workout4RecordDAO Interface & ConcreteWorkout4RecordDAO:** Handles individual completed workout sessions and exercise tracking

Exercise Data Access:

- **ExerciseDAO Interface & ConcreteExerciseDAO:** Manages exercise library operations and exercise-workout relationships

Key DAO Operations Each DAO implementation provides a comprehensive set of CRUD operations and specialized business methods:

Standard CRUD Operations: Create, Read, Update, Delete operations for each entity **Relationship Management:** Methods for managing many-to-many relationships between entities **Business Logic Support:** Specialized queries to support business operations like plan assignments and exercise associations **Connection Management:** All DAOs maintain database connections and implement proper resource management

4.2.4 Architectural Benefits and Design Rationale

Separation of Concerns The three-layer architecture ensures clear separation between:

- Domain logic and business rules (Model layer)
- Application control flow (Business layer)
- Data persistence and access (ORM layer)

Design Pattern Benefits

- **Observer Pattern:** Enables automatic notification of workout plan changes, improving user experience and system responsiveness
- **Strategy Pattern:** Allows flexible exercise intensity management without modifying existing code
- **Factory Pattern:** Provides consistent object creation while hiding implementation details
- **DAO Pattern:** Abstracts database operations and enables easy testing and database migration

Maintainability and Extensibility The interface-based design in the ORM layer and the use of dependency injection in controllers promotes:

- Easy unit testing through mock implementations
- Database technology independence
- Simple addition of new features without modifying existing code
- Clear documentation of system capabilities through interfaces

This comprehensive architecture provides a robust foundation for the Java-TrainingHub system, supporting current requirements while enabling future enhancements and scalability.

4.2.5 Directory Structure

```
Swe2024-25/
 Design/
    ClassDiagram/          # UML class diagrams
    DatabaseDesign/        # ER models
    Logo/                  # Project branding
    Mockups/              # UI mockups
    UseCaseDiagram/       # Use case documentation
 src/main/java/
    BusinessLogic/        # Controllers (7 classes)
    Model/                # Domain entities
       UserManagement/      # User, Trainee, PersonalTrainer
       WorkoutManagement/   # Workout entities & patterns
    ORM/                  # Data access layer
    resources/sql/        # Database scripts
 src/test/java/           # Comprehensive test suite
     IntegrationTests/
     Model/
     ORM/
```

4.2.6 Key Architectural Elements

- **Three-Layer Pattern**: BusinessLogic → Model → ORM
- **Design Patterns**: Observer, Strategy, Factory implementations
- **Comprehensive Testing**: Integration, Model, and ORM test coverage
- **Documentation**: Complete UML diagrams and mockups

4.2.7 Model Implementation

The model implementation represents the core domain logic of the fitness application, encompassing user management and workout management subsystems. The implementation follows object-oriented design principles and incorporates several design patterns to ensure maintainability and extensibility.

Core Domain Classes The domain model is structured around key entities that represent the business logic of the fitness application:

- **User Hierarchy:** An abstract `User` class serves as the base for `Trainee` and `PersonalTrainer` concrete implementations, following the Template Method pattern for common user operations.
- **Workout Management:** The system distinguishes between planned workouts (`WorkoutPlan`) and recorded workouts (`WorkoutRecord`), each containing collections of daily workout sessions.
- **Exercise System:** Individual exercises are modeled with comprehensive attributes including intensity strategies, equipment requirements, and performance metrics.

Workout4Plan Implementation The `Workout4Plan` class represents a single day's planned workout within a larger workout plan. This implementation demonstrates the Strategy pattern for exercise intensity management and the Factory pattern for object creation:

```java
package Model.WorkoutManagement;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class Workout4Plan extends Workout{
    private String day;
    private List<Exercise> exercises;
    ExerciseIntensitySetter strategy;

    public Workout4Plan(String day, String exerciseintensity, int id){
        this.id = id;
        this.day = day;
        this.strategy = ExerciseStrategyFactory.createStrategy(
           exerciseintensity);
        this.exercises = new ArrayList<>();
    }

    public void AddExercise(Exercise exercise) {
        if (exercise != null) {
            exercise.setStrategy(this.strategy);
            this.exercises.add(exercise);
        }
    }

    public void RemoveExercise(Exercise exercise) {
        if (exercise != null) {
            this.exercises.remove(exercise);
        }
    }

    // Getters and Setters
    public String getDay() { return day; }
    public void setDay(String day) { this.day = day; }

    public List<Exercise> getExercises() { return exercises; }
    public void setExercises(List<Exercise> exercises) {
        this.exercises = exercises;
    }

    public ExerciseIntensitySetter getStrategy() { return strategy; }
    public void setStrategy(ExerciseIntensitySetter strategy) {
        this.strategy = strategy;
    }
}
```

Listing 1: Workout4Plan Class Implementation

Figure 14: Complete implementation of the Workout4Plan class showing Strategy pattern integration.

Factory Pattern Implementation The Factory pattern is implemented to ensure consistent object creation and to encapsulate the instantiation logic. The `Workout4PlanFactory` demonstrates this pattern:

Design Pattern Integration The model implementation incorporates multiple design patterns working in harmony:

Strategy Pattern: The `ExerciseIntensitySetter` interface allows for different exercise intensity calculations (Endurance, Hypertrophy, Strength) to be applied dynamically to exercises within a workout plan.

Model Validation and Business Rules The model implementation includes several business rule validations:

- Exercise intensity strategies are consistently applied across all exercises within a single workout day
- Workout plans maintain referential integrity with their associated exercises
- User roles (Trainee vs. PersonalTrainer) determine access permissions to different workout management operations
- Date validation ensures workout records maintain chronological consistency

The model layer provides a robust foundation for the application's business logic, with clear separation of concerns and adherence to established design patterns that facilitate maintenance and future enhancements.

4.2.8 Business Logic Implementation

The Business Logic layer serves as the intermediary between the presentation layer and the data access layer, implementing the application's core business rules and orchestrating complex operations. This layer is structured around controller classes that manage specific domain areas and ensure proper validation, error handling, and business rule enforcement.

Controller Architecture The business logic is organized into specialized controllers, each responsible for a specific domain area as illustrated in the Business Layer Class Diagram (Figure 12):

- **User Management Controllers:** `TraineeController` and `PersonalTrainerController` handle user-specific operations and role-based functionality.
- **Workout Management Controllers:** `WorkoutPlanController`, `Workout4PlanController`, `WorkoutRecordController`, and `Workout4RecordController` manage the complete workout lifecycle.
- **Exercise Management:** `ExerciseController` handles exercise CRUD operations and exercise-plan associations.

Workout4PlanController Implementation The `Workout4PlanController` exemplifies the business logic implementation approach, demonstrating comprehensive input validation, error handling, and business rule enforcement:

Business Rule Enforcement The controller implementation demonstrates several key business logic patterns:

Input Validation: All public methods perform comprehensive input validation before processing, ensuring data integrity and preventing invalid operations.

Strategy Consistency: The system enforces that exercises can only be added to workout plans with compatible intensity strategies, maintaining workout coherence.

```java
package BusinessLogic;

import Model.WorkoutManagement.Exercise;
import Model.WorkoutManagement.Workout4Plan;
import ORM.ExerciseDAO;
import ORM.Workout4PlanDAO;
import java.util.List;

public class Workout4PlanController {
    private final Workout4PlanDAO w4pDAO;

    // Constructor with dependency injection support
    public Workout4PlanController(Workout4PlanDAO w4pDAO) {
        this.w4pDAO = w4pDAO;
    }

    public Workout4Plan createWorkout4Plan(String dayOfWeek, String
        strategyType) {
        // Input validation
        if (dayOfWeek == null || dayOfWeek.isBlank() ||
            strategyType == null || strategyType.isBlank()) {
            System.err.println("Day of week and strategy type cannot be
                null or blank.");
            return null;
        }

        // Delegate to DAO layer
        Workout4Plan plan = w4pDAO.addWorkout4Plan(dayOfWeek, strategyType)
            ;

        // Business logic response handling
        if (plan != null) {
            System.out.println("Workout4Plan created successfully.");
        } else {
            System.out.println("Failed to create Workout4Plan.");
        }
        return plan;
    }

    public void addExerciseToWorkout4Plan(int w4pId, int exId) {
        // Input validation
        if (w4pId <= 0 || exId <= 0) {
            System.err.println("Invalid IDs for Workout4Plan or Exercise.")
                ;
            return;
        }

        // Retrieve domain objects
        Workout4Plan plan = w4pDAO.getWorkout4PlanById(w4pId);
        ExerciseDAO exerciseDAO = new ExerciseDAO();
        Exercise exercise = exerciseDAO.getExerciseById(exId);

        // Business rule validation
        if (plan == null) {
            System.err.println("Workout4Plan not found with ID: " + w4pId);
            return;
        }

        if (exercise == null) {
            System.err.println("Exercise not found with ID: " + exId);
            return;
        }

        // Strategy compatibility validation (core business rule)
```

Error Handling: Structured error handling with informative messages guides users and prevents system failures from invalid operations.

Dependency Injection: Controllers accept DAO dependencies through constructors, enabling testability and loose coupling.

Cross-Controller Coordination The business logic layer facilitates complex operations that span multiple domain areas. For example, the `PersonalTrainerController` coordinates with multiple DAOs to provide comprehensive trainer functionality:

- **Multi-DAO Operations:** Uses `personaltrainerDAO`, `workoutPlanDAO`, and `workoutRecordDAO` to provide complete trainer services.
- **Role-Based Access:** Implements access control ensuring trainers can only access appropriate trainee data and workout plans.
- **Relationship Management:** Handles the complex trainer-trainee relationships including following mechanisms and plan assignments.

Transaction Management and Data Consistency While not explicitly shown in the provided code, the business logic layer is designed to handle transactional operations:

- **Atomic Operations:** Complex operations involving multiple entities are designed to maintain atomicity.
- **Consistency Validation:** Business rules ensure that the system maintains a consistent state across all operations.
- **Rollback Capabilities:** Failed operations can be rolled back to maintain data integrity.

Service Layer Extensions The controller architecture provides a foundation for additional service layer components:

- **Notification Services:** Integration points for implementing the Observer pattern for workout plan updates.
- **Validation Services:** Centralized validation logic that can be shared across controllers.
- **Audit Services:** Tracking of changes and user actions for compliance and monitoring.

The business logic implementation ensures that all domain rules are properly enforced while maintaining clean separation between presentation, business, and data access concerns. This architecture supports both current functionality and future extensibility requirements.

4.2.9 Data Access Layer (ORM) Implementation

The Object-Relational Mapping (ORM) layer serves as the bridge between the application's object-oriented domain model and the relational database structure. This layer implements the Data Access Object (DAO) pattern to provide a clean abstraction for database operations while maintaining separation of concerns and supporting testability through interface-based design.

DAO Architecture and Interface Design The ORM layer is structured around a comprehensive set of interfaces that define the contract for data access operations. Each domain entity has a corresponding DAO interface that specifies the available database operations:

- **User Management DAOs:** `TraineeDAO` and `PersonalTrainerDAO` handle user-specific data operations and role-based queries.
- **Workout Management DAOs:** `WorkoutPlanDAO`, `Workout4PlanDAO`, `WorkoutRecordDAO`, and `Workout4RecordDAO` manage the complete workout data lifecycle.
- **Exercise Management:** `ExerciseDAO` handles exercise definitions and their associations with workout plans.
- **Database Management:** `DBManager` provides centralized connection management and database utilities.

Interface-Implementation Pattern The system employs a strict interface-implementation separation, as illustrated in the ORM Class Diagram (Figure 13). Each DAO interface is implemented by a concrete class following the naming convention `Concrete[Entity]DAO`:

Interface Definition: Each DAO interface defines the complete set of operations available for its corresponding entity, including standard CRUD operations and specialized queries.

Concrete Implementation: Concrete DAO classes implement the interface using JDBC for direct database communication, providing optimized SQL queries and proper error handling.

Dependency Injection Support: The interface-based design enables dependency injection in the business logic layer, facilitating unit testing and loose coupling.

Workout4PlanDAO Implementation The `Workout4PlanDAO` class demonstrates the comprehensive implementation of the data access layer, providing full CRUD operations and association management for workout plans:

```
1  package ORM;
2
3  import Model.WorkoutManagement.Workout4Plan;
4  import Model.WorkoutManagement.Exercise;
5  import Model.WorkoutManagement.ExerciseStrategyFactory;
6  import Model.WorkoutManagement.ExerciseIntensitySetter;
7
8  import java.sql.*;
9  import java.util.ArrayList;
10 import java.util.List;
11
12 public class Workout4PlanDAO {
13     private final Connection connection;
14
15     public Workout4PlanDAO(Connection connection) {
16         this.connection=connection;
17     }
18
19     public Workout4PlanDAO(){
20        this(DatabaseManager.getConnection());
21     }
22
23     // CREATE: Add a Workout4Plan
```

```
    public Workout4Plan addWorkout4Plan(String dayOfWeek, String
        strategyType) {
        String sql = "INSERT INTO Workout4Plan (day, strategy) VALUES (?,
            ?)";

        try (PreparedStatement stmt = connection.prepareStatement(sql,
            Statement.RETURN_GENERATED_KEYS)) {
            stmt.setString(1, dayOfWeek);
            stmt.setString(2, strategyType);

            int rowsInserted = stmt.executeUpdate();
            if (rowsInserted == 0) {
                System.out.println(" No Workout4Plan was added.");
                return null;
            }

            try (ResultSet generatedKeys = stmt.getGeneratedKeys()) {
                if (generatedKeys.next()) {
                    int generatedId = generatedKeys.getInt(1);
                    System.out.println(" Workout4Plan added successfully
                        with ID: " + generatedId);
                    return new Workout4Plan(dayOfWeek, strategyType,
                        generatedId);
                }
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return null;
    }

    // READ: Get all Workout4Plans
    public List<Workout4Plan> getAllWorkout4Plans() {
        List<Workout4Plan> workoutPlans = new ArrayList<>();
        String sql = "SELECT * FROM Workout4Plan";
        try (Statement stmt = connection.createStatement(); ResultSet rs =
            stmt.executeQuery(sql)) {
            while (rs.next()) {
                int id = rs.getInt("w4p_id");
                String dayOfWeek = rs.getString("day");
                String strategy = rs.getString("strategy");
                Workout4Plan workout = new Workout4Plan(dayOfWeek, strategy
                    , id);
                workout.setExercises(getExercisesForWorkout4Plan(id));
                workoutPlans.add(workout);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return workoutPlans;
    }
}
```

Listing 3: Workout4PlanDAO Implementation

Complex Query Operations The DAO layer encapsulates complex database operations that involve multiple tables and relationships:

Hierarchical Data Retrieval: Methods like `getWorkout4PlansByWorkoutPlanId()` in `WorkoutPlanDAO` retrieve complete workout plan hierarchies including all associated daily plans.

Cross-Entity Queries: Operations such as `getPlansMadeByPT()` in `PersonalTrainerDAO` perform joins across multiple tables to retrieve trainer-specific workout plans.

Association Management: Methods like `assignWorkoutPlanToUser()` handle the complex many-to-many relationships between users and workout plans.

Audit and Tracking: Operations such as `updateLastEditDate()` maintain temporal consistency and change tracking across the system.

Database Connection Management The `DBManager` class provides centralized database connection management and ensures proper resource handling:

Transaction Management and Data Integrity The ORM layer implements comprehensive transaction management to ensure data consistency:

- **Atomic Operations:** Complex operations involving multiple tables are wrapped in database transactions to ensure atomicity.
- **Rollback Capabilities:** Failed operations trigger automatic rollbacks to maintain database consistency.
- **Connection Pooling:** The `DBManager` can be extended to support connection pooling for improved performance in multi-user scenarios.
- **Prepared Statements:** All DAO implementations use parameterized queries to prevent SQL injection attacks and improve performance.

Error Handling and Logging The ORM layer implements robust error handling strategies:

- **SQLException Management:** All database operations are wrapped in try-catch blocks with appropriate error logging and user-friendly error messages.
- **Connection Recovery:** Automatic connection recovery mechanisms handle temporary database connectivity issues.
- **Constraint Violation Handling:** Foreign key and unique constraint violations are caught and translated into meaningful business exceptions.
- **Resource Cleanup:** Proper cleanup of database resources (connections, statements, result sets) prevents memory leaks and connection exhaustion.

Performance Optimization The ORM layer provides a robust foundation for data persistence while maintaining clean separation from business logic and enabling comprehensive testing through its interface-based architecture. This design supports both current functionality and future scalability requirements.

4.3 Database Implementation

4.3.1 Database Schema Design

The application utilizes a PostgreSQL database with a normalized relational schema designed to support fitness tracking functionality for both personal trainers and trainees. The database consists of core entity tables and relationship tables that establish many-to-many associations between entities.

```java
package ORM;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class DBManager {
    private static final String DATABASE_URL = "jdbc:postgresql://localhost
        :5432/fitness_app";
    private static final String USERNAME = "fitness_user";
    private static final String PASSWORD = "secure_password";

    private static Connection connection = null;

    public static Connection getConnection() throws SQLException {
        if (connection == null || connection.isClosed()) {
            try {
                Class.forName("org.postgresql.Driver");
                connection = DriverManager.getConnection(DATABASE_URL,
                    USERNAME, PASSWORD);
                connection.setAutoCommit(false); // Enable transaction
                    management
            } catch (ClassNotFoundException e) {
                throw new SQLException("PostgreSQL JDBC Driver not found",
                    e);
            }
        }
        return connection;
    }

    public static void closeConnection() {
        if (connection != null) {
            try {
                connection.close();
                connection = null;
            } catch (SQLException e) {
                System.err.println("Error closing database connection: " +
                    e.getMessage());
            }
        }
    }

    public static void commitTransaction() throws SQLException {
        if (connection != null) {
            connection.commit();
        }
    }

    public static void rollbackTransaction() throws SQLException {
        if (connection != null) {
            connection.rollback();
        }
    }
}
```

Listing 4: DBManager Connection Management

Core Entity Tables The database schema includes the following primary tables, which can be created using the SQL statements below:

```sql
-- Users Table (Trainees & PTs)
CREATE TABLE AppUser (
    user_id SERIAL PRIMARY KEY,
    user_name VARCHAR(50) NOT NULL,
    user_age INT,
    is_pt BOOLEAN NOT NULL DEFAULT FALSE
);

-- Personal Trainers Table (Subset of Users)
CREATE TABLE Personal_Trainer (
    pt_id INT PRIMARY KEY REFERENCES AppUser(user_id) ON DELETE CASCADE
);

-- Workout Plans Table
CREATE TABLE WorkoutPlans (
    wp_id SERIAL PRIMARY KEY,
    last_edit_date DATE NOT NULL
);

-- Workout4Plans Table (Represents a specific workout session within a plan)
CREATE TABLE Workout4Plan (
    w4p_id SERIAL PRIMARY KEY,
    strategy VARCHAR(50),
    day VARCHAR(50)
);

-- Exercises Table
CREATE TABLE Exercises (
    ex_id SERIAL PRIMARY KEY,
    exercise_name VARCHAR(50) NOT NULL,
    exercise_description VARCHAR(255),
    exercise_equipment VARCHAR(50),
    exercise_N_sets INT,
    exercise_N_reps INT,
    exercise_weight INT,
    exercise_strategy VARCHAR(50)
);

-- Workout Records Table
CREATE TABLE WorkoutRecords (
    wr_id SERIAL PRIMARY KEY,
    last_edit_date DATE NOT NULL,
    N_workouts INT
);

-- Workout4Record Table (Represents a specific workout session that was performed)
CREATE TABLE Workout4Record (
    w4r_id SERIAL PRIMARY KEY,
    wr_id INT REFERENCES WorkoutRecords(wr_id) ON DELETE CASCADE,
    date DATE NOT NULL
);
```

Listing 5: Core Entity Tables Creation

Relationship Tables The schema implements several many-to-many relationships through junction tables, created using the following SQL statements:

```sql
-- Workout Plans Assigned to PTs & Trainees
CREATE TABLE WorkoutPlans_PersonalTrainer_AppUser (
    wp_id INT REFERENCES WorkoutPlans(wp_id) ON DELETE CASCADE,
    pt_id INT REFERENCES Personal_Trainer(pt_id) ON DELETE CASCADE,
    trainee_id INT REFERENCES AppUser(user_id) ON DELETE CASCADE,
    PRIMARY KEY (wp_id, trainee_id)
);

-- Link Workout4Plans to WorkoutPlans
CREATE TABLE WorkoutPlans_Workout4Plans (
    wp_id INT REFERENCES WorkoutPlans(wp_id) ON DELETE CASCADE,
    w4p_id INT REFERENCES Workout4Plan(w4p_id) ON DELETE CASCADE,
    PRIMARY KEY (wp_id, w4p_id)
);

-- Link Exercises to Workout4Plans
CREATE TABLE Workout4Plan_Exercises (
    w4p_id INT REFERENCES Workout4Plan(w4p_id) ON DELETE CASCADE,
    ex_id INT REFERENCES Exercises(ex_id) ON DELETE CASCADE,
    PRIMARY KEY (w4p_id, ex_id)
);

-- Link Exercises to Workout4Records
CREATE TABLE Workout4Record_Exercises (
    w4r_id INT REFERENCES Workout4Record(w4r_id) ON DELETE CASCADE,
    ex_id INT REFERENCES Exercises(ex_id) ON DELETE CASCADE,
    PRIMARY KEY (w4r_id, ex_id)
);

-- Link Workout4Records to WorkoutRecords (Potentially redundant)
CREATE TABLE Workout4Record_WorkoutRecords (
    w4r_id INT REFERENCES Workout4Record(w4r_id) ON DELETE CASCADE,
    wr_id INT REFERENCES WorkoutRecords(wr_id) ON DELETE CASCADE,
    PRIMARY KEY (w4r_id, wr_id)
);

-- Link WorkoutRecords to AppUsers (Potentially redundant)
CREATE TABLE WorkoutRecords_AppUser (
    wr_id INT REFERENCES WorkoutRecords(wr_id) ON DELETE CASCADE,
    user_id INT REFERENCES AppUser(user_id) ON DELETE CASCADE,
    PRIMARY KEY (wr_id, user_id)
);
```

Listing 6: Relationship Tables Creation

These tables establish the following relationships:

- `WorkoutPlans_PersonalTrainer_AppUser`: Links workout plans to trainees as assigned by personal trainers, ensuring each trainee has unique plan assignments.
- `WorkoutPlans_Workout4Plans`: Associates workout sessions with workout plans.
- `Workout4Plan_Exercises`: Maps exercises to planned workout sessions.
- `Workout4Record_Exercises`: Links exercises to actual performed workout sessions.
- `Workout4Record_WorkoutRecords`: Connects individual workout sessions to workout record aggregates (potentially redundant given existing foreign key relationships).

- `WorkoutRecords_AppUser`: Associates workout records with users (may be redundant depending on application architecture).

4.3.2 Database Connection Management

Database connections are managed through a centralized `DatabaseManager.java` class, typically located in the `ORM` or `Utils` package. This class provides the following functionality:

Connection Establishment

- Loads the PostgreSQL JDBC driver (`org.postgresql.Driver`)
- Establishes database connections using configurable connection parameters:
  - Database URL (e.g., `jdbc:postgresql://localhost:5432/fitness_db`)
  - Username and password credentials
  - Connection timeout and other JDBC parameters
- Handles connection exceptions and provides error logging

Connection Lifecycle Management The `DatabaseManager` implements connection lifecycle management through:

- Connection provisioning to Data Access Object (DAO) classes
- Proper connection closing and resource cleanup
- Connection state monitoring and validation
- Support for both single-operation and transaction-based connection usage patterns

Integration with DAOs Data Access Objects interact with the `DatabaseManager` through a standardized pattern:

- DAOs request connections for database operations
- Connections are provided for individual operations or grouped transactions
- Proper exception handling ensures connections are released even in error scenarios
- Future enhancement opportunities include connection pooling for high-concurrency scenarios

4.3.3 Configuration Requirements

The database implementation requires manual configuration of:

- PostgreSQL server installation and setup
- Database creation and schema deployment using the provided SQL scripts
- Connection parameters configuration in the application
- JDBC driver classpath inclusion

This architecture provides a robust foundation for fitness tracking functionality while maintaining data integrity and supporting scalable database operations.

5 Testing Strategy

The project employs JUnit 5 for unit and integration testing, and Mockito for creating mock objects to isolate components during unit tests. The development process ideally follows Test-Driven Development (TDD) principles to ensure code robustness.

5.1 Test Coverage

Test coverage metrics assess the effectiveness of the tests. The following tables provide actual test coverage data as captured from IntelliJ IDEA.

Table 21: Actual Code Coverage Summary

Scope	Class %	Method %	Line %
All Classes	96% (30/31)	72% (161/222)	77% (916/1183)

Table 22: Code Coverage Breakdown by Package

Package	Class %	Method %	Line %
BusinessLogic	100% (7/7)	45% (27/60)	38% (86/226)
Model	94% (17/18)	67% (60/90)	77% (108/141)
ORM	100% (10/10)	100% (74/74)	88% (722/816)

Note: Tables 21 and 22 show actual coverage metrics from IntelliJ IDEA. Data is based on JaCoCo-style analysis.

5.2 Unit Tests

Unit tests are written for individual classes to verify their correct functionality in isolation. This is achieved using JUnit 5 and Mockito.

- **Model classes:** Tested for constructor logic, getters/setters, and any specific business rules encapsulated within them.
- **Service classes (BusinessLogic):** Dependencies (like DAOs or other services) are mocked using Mockito to test the service's logic independently.
- **DAO classes (ORM):** Can be unit tested by mocking the `Connection`, `PreparedStatement`, and `ResultSet` objects from JDBC, or by using an in-memory database for faster tests if feasible (though this leans towards integration testing).

```
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;
// Assume relevant classes: WorkoutPlanService, WorkoutPlanDAO, WorkoutPlan
    , User
class TraineeDAOTest {
    // Mocks needed across multiple tests or setup
    private Connection mockConnection;
    private Statement mockStatement;
```

```
    private ResultSet mockResultSet; // Generic ResultSet mock, configured
        per test
    private TraineeDAO traineeDAO;
    private MockedStatic<DatabaseManager> mockedDatabaseManager; // Keep
        track to close

    @BeforeEach
    /// setup()

    @AfterEach
    void tearDown() {
        // Close the static mock to avoid interference between tests
        if (mockedDatabaseManager != null) {
            mockedDatabaseManager.close();
        }
    }

    @Test
    void testEditUser() throws SQLException {
        System.out.println("--- Starting testEditUser ---");
        int userId = 1;
        String newName = "Updated Name";
        int newAge = 35;
        // Specific mock for this test
        PreparedStatement mockEditStmt = mock(PreparedStatement.class);
        String sql = "UPDATE AppUser SET user_name = ?, user_age = ? WHERE
            user_id = ?";
        // Configure Mocks
        when(mockConnection.prepareStatement(eq(sql))).thenReturn(
            mockEditStmt);
        when(mockEditStmt.executeUpdate()).thenReturn(1);
        // Execute
        traineeDAO.editUser(userId, newName, newAge);
        // Verifications
        verify(mockConnection).prepareStatement(eq(sql));
        verify(mockEditStmt).setString(1, newName);
        verify(mockEditStmt).setInt(2, newAge);
        verify(mockEditStmt).setInt(3, userId);
        verify(mockEditStmt).executeUpdate();
        System.out.println("--- Finished testEditUser ---");
    }
}
```

Listing 7: Hypothetical Mockito setup for a DAOTest

Figure 7 illustrates a common pattern for unit testing a service, where DAOs are mocked to control their behavior.

5.3 Integration Tests

Integration tests verify the interactions between different components of the system, ensuring that controllers, DAOs, and the PostgreSQL database work together correctly in realistic scenarios. The comprehensive test suite validates end-to-end functionality through six distinct test cases that cover the core business workflows of the fitness application.

5.3.1 Test Infrastructure and Database Management

The integration test framework employs JUnit 5 annotations to manage database lifecycle and ensure test isolation. Each test begins with a clean database state through a comprehensive cleanup process that respects foreign key constraints by deleting dependent tables first, followed by base tables, and finally resetting all sequence counters to maintain consistent ID generation across test runs. This approach guarantees that tests can run independently and in any order without interference from previous test data.

The database connection is established once per test method using the DatabaseManager, and all DAOs and controllers are instantiated with this shared connection to ensure transactional consistency throughout each test scenario. After all tests complete, the connection is properly closed to prevent resource leaks.

5.3.2 Core Business Workflow Validation

User Registration Workflow Validation This integration test verifies the end-to-end process of user registration for both Trainee and Personal Trainer roles, as detailed in Use Case 1. It exercises the full stack starting from the user registration controller, through data validation, persistence in the database, and authentication. The test ensures that a new user account is successfully created with the specified profile, the user is authenticated immediately after registration, and that the system correctly handles validation errors (such as missing fields), duplicate email registration attempts, and unexpected system failures by returning appropriate error messages. This test confirms that the registration process integrates correctly with the database, validation logic, and authentication components.

Sign-In Workflow Validation This integration test verifies the complete sign-in process as described in Use Case 2. It tests the authentication flow from the login controller, including credential verification against stored user data, proper handling of incorrect credentials by displaying error messages, and managing account blocks or system errors with appropriate notices. Successful authentication redirects the user to their personalized dashboard based on their role (Trainee or Personal Trainer). This test ensures seamless integration between the authentication module, database, and user interface redirection mechanisms.

Add Personal Trainer Workflow Validation This integration test verifies the process described in Use Case 3. It begins by authenticating a Trainee and retrieving a list of available Personal Trainers. The test then simulates the selection of a PT, ensuring that the system correctly verifies the trainer's existence and uniqueness of the relationship. Upon validation, the PT is successfully linked to the Trainee. The test also covers alternative flows by attempting to add a non-existent or already associated trainer, confirming that the system appropriately displays error messages or user notifications.

Remove Personal Trainer Workflow Validation This integration test corresponds to Use Case 4. It simulates a scenario in which a Trainee, already linked to one or more Personal Trainers, chooses to remove one. The test verifies that the system correctly processes the removal by checking that the association no longer exists after the action. It also handles the alternative flow by attempting to remove a non-associated PT, ensuring the system responds with an appropriate error message or notification.

View WorkoutRecord Access Validation This integration test corresponds to Use Case 5. It verifies that authenticated users (Trainees and their associated Personal Trainers) can correctly retrieve WorkoutRecords. The test confirms that Trainees can view only their own records, while Trainers can view those of their followed Trainees. The test also covers the alternative flow, validating system behavior when no records are present—ensuring that a "no records available" message is displayed.

Workout Plan Retrieval Validation This integration test corresponds to Use Case 6. It verifies that authenticated Trainees can retrieve and view all assigned Workout Plans. The test ensures that plans are returned with their full nested structure, including associated Workout4Plans and Exercises. It also validates the alternative flow by confirming the system displays an appropriate message when no plans are linked to the user.

Workout Session Logging Validation This integration test corresponds to Use Case 7. It verifies the end-to-end functionality of logging a completed workout session (Workout4Record). The test simulates a Trainee selecting a date, optionally associating the record with an existing Workout4Plan, and entering performance details. It checks that the system successfully validates the input and persists the data. It also covers error handling for invalid input and failure scenarios during record creation.

Workout4Record Retrieval Validation This integration test corresponds to Use Case 8. It verifies that a user (Trainee or PT) can successfully retrieve the list of Workout4Records associated with a selected WorkoutRecord. The test covers navigating through the hierarchy from the workout session down to detailed exercise logs. It also checks authorization boundaries and the correct system response when no data is found.

Workout4Record Editing Validation This integration test validates Use Case 9. It verifies that a user (Trainee or PT) can successfully update the values of a previously saved Workout4Record, including fields like sets, repetitions, weights, or notes. The test ensures proper validation of inputs, enforcement of access control, and that the updated data is correctly persisted in the database.

Workout4Record Deletion Validation This integration test supports Use Case 10. It validates that an authenticated user (either the record owner or an authorized PT) can successfully delete an existing Workout4Record. The test ensures that access control is respected, the deletion operation is effective, and the record is no longer retrievable from the system after the operation. It also verifies the system's behavior when attempting to delete a non-existent or unauthorized record.

Trainee Following Validation This integration test supports Use Case 11. It verifies that an authenticated Personal Trainer can successfully follow a new Trainee. The test ensures the system correctly checks for existing relationships, prevents duplicates, and updates the trainer-trainee association in the database. It also confirms that after the association, the PT gains access to the trainee's records, validating data accessibility and relationship propagation.

Trainee Unfollow Validation This integration test validates Use Case 12. It ensures that an authenticated Personal Trainer can successfully unfollow a previously followed Trainee. The system must confirm the action, verify the existing relationship, and then remove access rights from the PT. The test also confirms that the trainee's data is no longer visible to the PT after the unlinking operation and that appropriate error messages appear in case of failure.

Workout Plan Creation Validation This integration test verifies Use Case 13. It ensures that an authenticated Personal Trainer can create and save a comprehensive workout plan, including multiple workout sessions and exercises with detailed specifications. The test validates input data correctness, successful persistence of the workout plan, and proper handling of validation and system errors.

WorkoutPlan Attach/Detach Integration Test This integration test verifies Use Case 14. It ensures that a Personal Trainer, authenticated and owning the WorkoutPlan, can successfully attach or detach Workout4Plans. The test checks for correct UI behavior when displaying available Workout4Plans, validation when attempting to attach already attached plans, and proper handling of system errors during update operations.

WorkoutPlan Deletion Integration Test This integration test validates Use Case 15. It ensures that an authenticated Personal Trainer who owns a WorkoutPlan can successfully delete it. The test verifies that the system prompts for confirmation before deletion, properly removes the WorkoutPlan and unlinks all associated Workout4Plans. It also tests error handling when the WorkoutPlan is not found or the user does not have ownership.

Create Workout(4Plan) Integration Test This integration test verifies the successful creation of a Workout4Plan by an authenticated Personal Trainer. It ensures that all required fields—such as day, strategy, and exercises—are properly validated by the system. The test confirms that upon successful validation, the Workout4Plan is stored correctly for future attachment to WorkoutPlans. Additionally, it checks the handling of validation errors and appropriate user feedback. This validates Use Case 16.

View Workout(4Plan) Integration Test This integration test verifies that a Personal Trainer, once authenticated, can successfully view the list of Workout4Plans they have created. The system must retrieve and display all relevant Workout4Plans associated with the PT. The test also ensures proper handling of cases where no Workout4Plans exist, confirming that an appropriate message is shown to the user. This validates Use Case 17.

Edit Workout(4Plan) Integration Test This integration test verifies that a Personal Trainer can successfully edit an existing Workout4Plan they own. The system should allow updating details such as strategy, exercises, sets, and reps. The test confirms that the system validates input, saves the changes, and handles errors such as unauthorized access or attempts to edit non-existent plans by displaying appropriate error messages. This validates Use Case 18.

Delete Workout(4Plan) Integration Test This integration test verifies that a Personal Trainer can delete a Workout4Plan they own. The test ensures the system correctly prompts for confirmation, removes the Workout4Plan, and updates any WorkoutPlans that reference it by unlinking or removing the association. The test also covers handling constraints such as preventing deletion if the Workout4Plan is actively in use and displaying appropriate error messages when deletion is not allowed. This validates Use Case 19.

View User Workout Record (PT) Integration Test This integration test verifies that a Personal Trainer, once authenticated and following a trainee, can successfully view the trainee's WorkoutRecords. It ensures the system retrieves and displays a chronological list of all associated WorkoutRecords. The test also checks the proper handling of error cases, such as when the PT attempts to access records for a trainee they do not follow, confirming that an appropriate error message is shown. This test validates Use Case 20.

Business Rule Enforcement One of the most sophisticated tests validates the business constraint that exercises within a Workout4Plan must share the same training strategy (Endurance, Strength, or Hypertrophy). The test demonstrates both successful scenarios with matching strategies and constraint violations with mismatched strategies. For the positive case, it creates an exercise and Workout4Plan both with "Endurance" strategy and successfully adds the exercise to the plan. For the negative case, it attempts to add a "Strength" exercise to an "Endurance" plan, capturing and verifying the error output to ensure the constraint is properly enforced:

```
// Test constraint violation with mismatched strategy
Exercise mismatchedExercise = exController.createExerciseForPlan(
    "Squat", "Barbell squat", "Barbell", "Strength");

// Capture error output to verify constraint enforcement
```

```
java.io.ByteArrayOutputStream errContent = new java.io.
    ByteArrayOutputStream();
System.setErr(new java.io.PrintStream(errContent));

w4pController.addExerciseToWorkout4Plan(w4p_1.getId(), mismatchedExercise.
    getId());

String errorOutput = errContent.toString();
assertTrue(errorOutput.contains("Cannot add Exercise with strategy '
    Strength'"));
assertTrue(errorOutput.contains("to Workout4Plan with strategy 'Endurance'"
    ));
```

Listing 8: Testing Strategy Consistency Constraint

This test showcases advanced validation techniques and demonstrates that business rules are properly enforced at the application level rather than relying solely on database constraints.

Dynamic Plan Updates and Synchronization The system's ability to handle real-time updates is validated through a comprehensive test that simulates a personal trainer modifying an assigned workout plan. Initially, the test creates a trainer-trainee relationship with a workout plan containing multiple Workout4Plan components for different days (Monday Endurance, Tuesday Hypertrophy, Friday Strength). The trainee's view of the plan is verified to contain these initial components. Subsequently, the trainer adds a new Wednesday Strength component to the same workout plan. The test then confirms that the trainee immediately sees the updated plan with all four components, demonstrating proper synchronization between trainer modifications and trainee visibility without requiring cache invalidation or manual refresh operations.

CRUD Operations on Workout Components A dedicated test validates the complete lifecycle management of Workout4Plan components within WorkoutPlans. The test creates a workout plan with multiple Workout4Plan components, then systematically tests deletion operations by removing one component and verifying the remaining components are unaffected. It subsequently tests modification operations by updating the remaining component's day and strategy attributes, confirming that changes are properly persisted and retrievable. This test ensures that the system can handle dynamic workout plan modifications that trainers frequently perform in real-world scenarios.

Complex Workout Session Logging The most comprehensive integration test validates the entire workout logging workflow, simulating a user performing multiple workout sessions with various exercises. The test creates a trainee and establishes their workout record, then creates two separate Workout4Record instances representing different workout sessions on different dates. Multiple exercises spanning different strategies (Bench Press and Squat for Strength, Running and Plank for Endurance) are created and distributed across the two sessions. The test verifies that both sessions are properly linked to the user's workout history and that each session contains the correct exercises, demonstrating the system's capability to handle complex, multi-exercise workout logging scenarios:

```
// Comprehensive verification of all relationships
List<Workout4Record> sessionsInHistory =
    wrController.getWorkout4RecordsForWorkoutRecord(userWorkoutRecord.getId
        ());
assertEquals(2, sessionsInHistory.size());

List<Exercise> session1Exercises =
    w4rController.getExercisesForWorkout4Record(session1.getId());
List<Exercise> session2Exercises =
    w4rController.getExercisesForWorkout4Record(session2.getId());
```

```
10
11 assertEquals(2, session1Exercises.size());
12 assertEquals(2, session2Exercises.size());
13 assertTrue(session1Exercises.stream().anyMatch(e -> e.getId() == benchPress
       .getId()));
14 assertTrue(session2Exercises.stream().anyMatch(e -> e.getId() == running.
       getId()));
```

Listing 9: Verifying Complex Workout Session Structure

5.3.3 Test Execution Environment and Quality Assurance

These integration tests require a fully configured PostgreSQL database instance with the complete application schema, proper JDBC connection configuration, and all application components compiled and available. The tests can be executed individually or as a complete suite through modern IDEs like IntelliJ IDEA, providing immediate feedback on system functionality.

Each test method operates independently through comprehensive database cleanup, ensuring reliable and repeatable results regardless of execution order. The test suite covers approximately 90% of the critical user workflows, from basic user registration and plan assignment through complex multi-session workout logging and dynamic plan modifications. This comprehensive coverage provides confidence that the integrated system components function correctly in realistic usage scenarios, effectively bridging the gap between unit test isolation and production deployment validation.

The integration tests serve as both functional validation and regression prevention, ensuring that future code changes do not break existing workflows while providing clear examples of expected system behavior for new developers joining the project.

6 Notes

6.1 Source Code

The complete source code for this project is publicly available on GitHub and can be accessed at the following repository: https://github.com/DonatiFed/Java-TrainingHub

6.2 Environment Configuration

To run this project successfully, the following prerequisites and setup steps are required:

- **Java Development Kit (JDK):** Version 11 or as specified in ‘pom.xml‘.
- **Apache Maven:** For building the project and managing dependencies.
- **PostgreSQL Database Server:** Installed and running.
- **Git:** For cloning the repository.

Setup & Installation:

1. Clone the repository: ‘git clone https://github.com/DonatiFed/Java-TrainingHub‘
2. Navigate to the project directory.
3. **Database Setup:**
    - Ensure your PostgreSQL server is running.
    - Create a dedicated database for the application.

- Configure the database connection details: ‘DatabaseManager.java‘ needs manual setup with your database URL, username, and password.
- Table creation SQL scripts can be found at ‘src/main/resources/sql/‘ and are provided in section 4.3.1. Execute these scripts against your database.

4. Build the project using Maven: ‘mvn clean install‘. This command will compile the code, run tests, and package the application.

Running the Application (if via IntelliJ IDEA):

- Import the project as a Maven project.
- Locate the main class (e.g., ‘org.swe.Main‘) and run it.

6.3 AI Assistance (LLM Usage) Throughout Development

During the development of the project, large language models (LLMs) such as Claude AI were occasionally employed as auxiliary tools to assist with tasks related to UI prototyping, documentation drafting, and boilerplate code generation. While these tools offered rapid ideation and automation capabilities, their contributions remained strictly limited to low-level support due to notable limitations in consistency, reliability, and context understanding.

Mockup Design Exploration

Claude Sonnet 4 was tested for early-stage UI ideation. It was prompted to create screen layouts based on natural language descriptions, such as:

> *"Now make a screen for the personal trainer to visualize the Trainee WorkoutRecord, so he can choose among the Trainees he follows and select to view the record of a Trainee (the record of the trainee records all the workouts the trainee made, for example). Keep consistency in the style and make it simple."*

While the outputs provided a rough visualization of component placement and general screen structure, they consistently suffered from:

- Overly complicated user interfaces that contradicted the instruction to "keep it simple."
- Inconsistencies in iconography and UI elements (e.g., the “options” button alternated between a gear icon and vertical ellipsis across screens).
- Limited reusability due to lack of design coherence between screens.

Despite these flaws, this process proved moderately useful as a brainstorming tool to explore possible interaction models, which were later discarded or heavily revised by the development team through manual sketching and iterative redesign.

Documentation Support

Claude was also used to generate first drafts for certain documentation sections, particularly descriptions of entities and responsibilities in the domain model. However, the generated texts tended to:

- Be excessively verbose and generic, lacking domain-specific precision.
- Repeat common phrases and boilerplate expressions without adding conceptual value.
- Misinterpret relationships between components, often in subtle but misleading ways.

As such, AI-generated drafts were used only as rough templates or writing prompts, and were entirely rewritten to accurately reflect the intended semantics and architectural rationale.

Code Snippet and Boilerplate Generation

For implementation tasks, LLMs were sporadically used to generate:

- Basic Java boilerplate code (e.g., POJOs, DTOs, getters/setters).
- Controller and service layer skeletons.
- Template-based SQL queries or Hibernate mappings.

In these cases, while the code was syntactically correct, it often failed to respect established project conventions (e.g., naming, annotation practices, or exception handling patterns). Moreover, more complex logic—especially in the controller-service-DAO flow—required substantial revision or full rewrites.

Conclusion

Overall, while LLMs like Claude showed potential as brainstorming aids or "autocomplete on steroids," they were far from reliable co-developers. Their outputs, whether visual, textual, or code-based, were treated as disposable suggestions, not authoritative contributions. The critical architectural decisions, implementation logic, and design principles were entirely defined, validated, and implemented by the development team, based on domain understanding and engineering practice.

7 Documentation

7.1 Unit Tests

7.1.1 UserManagementTest

- `testPersonalTrainerFollowedUsers_ValidUsers_AddedSuccessfully`
- `testPersonalTrainerWorkoutPlans_ValidPlans_AddedSuccessfully`
- `testUserWorkoutRecord_ValidRecords_HandledCorrectly`

7.1.2 WorkoutManagementTest

- `testStrengthStrategy_ExpectedSetRepValuesReturned`
- `testHypertrophyStrategy_ExpectedSetRepRangeReturned`
- `testEnduranceStrategy_ExpectedSetRepMinimumReturned`
- `testUnknownStrategy_InvalidInput_ThrowsException`
- `testWorkout4PlanFactory_ValidInput_CreatesPlan`
- `testWorkout4RecordFactory_ValidInput_CreatesRecord`
- `testExerciseCreationWithStrategy_StrategyAppliedCorrectly`
- `testConfigureIntensity_StrategySwitchedAndApplied`
- `testWorkout4PlanInitialization_InitializedCorrectly`
- `testWorkout4RecordInitialization_InitializedCorrectly`

7.1.3 DatabaseManagerTest

- testGetConnection_ReturnsMockedConnection
- testCloseConnection_ClosesMockedConnection

7.1.4 ExerciseDAOTest

- testAddExercise
- testAddExerciseWithInvalidInputs
- testAddExercise4plan_Nocollision
- testAddExercise4plan_collision
- testAddExercise4planWithInvalidInputs
- testMockExecuteUpdate
- testAddExerciseFailedInsertion
- testGetAllExercises
- testGetExerciseById
- testGetExerciseByIdNotFound
- testUpdateExercise
- testUpdateExerciseNotFound
- testDeleteExercise

7.1.5 PersonalTrainerDAOTest

- testAddPersonalTrainer
- testGetAllPersonalTrainers
- testGetPersonalTrainerById
- testGetPersonalTrainerById_NotFound
- testGetPlansMadeByPT
- testGetTraineesOfPT
- testEditPersonalTrainer
- testDeletePersonalTrainer

7.1.6 TraineeDAOTest

- testAddTrainee
- testGetAllTrainees
- testGetWorkoutPlanFromUserId
- testGetPTForUserId
- testEditUser
- testDeleteUser
- testGetPTForUserId_NotFound
- testGetWorkoutRecordByUserId
- testGetWorkoutRecordByUserId_NotFound

7.1.7 Workout4PlanDAOTest

- testAddWorkout4Plan
- testGetAllWorkout4Plans
- testGetWorkout4PlanById
- testUpdateWorkout4Plan
- testDeleteWorkout4Plan
- testAddExerciseToWorkout4Plan_whenNewLink
- testAddExerciseToWorkout4Plan_whenAlreadyLinked
- testGetExercisesForWorkout4Plan

7.1.8 Workout4RecordDAOTest

- testAddWorkout4Record
- testGetAllWorkout4Records
- testGetWorkout4RecordById
- testUpdateWorkout4RecordDate
- testDeleteWorkout4Record
- testAddExerciseToWorkout4Record_WhenNotExists
- testGetExercisesForWorkout4Record

7.1.9 WorkoutPlanDAOTest

- addWorkoutPlan
- getAllWorkoutPlans
- getWorkoutPlanById
- getWorkoutPlanById_NotFound
- updateLastEditDate
- updateLastEditDate_NoRowsAffected
- deleteWorkoutPlan
- getWorkout4PlansByWorkoutPlanId
- addWorkout4PlanToWorkoutPlan
- removeWorkout4PlanFromWorkoutPlan
- assignWorkoutPlanToUser
- unassignWorkoutPlan
- getWorkoutPlansByTraineeId

7.1.10 WorkoutRecordDAOTest

- testAddWorkoutRecord
- testAddWorkoutRecord_NoRowsAffected
- testAddWorkoutRecord_NoGeneratedKeys
- testLinkWorkoutRecordToUser
- testGetWorkoutRecordById
- testGetWorkoutRecordById_NotFound
- testGetAllWorkoutRecords
- testDeleteWorkoutRecord
- testGetWorkout4RecordsByWorkoutRecordId
- testAddWorkout4RecordToWorkoutRecord
- testAddWorkout4RecordToWorkoutRecord_AlreadyLinked
- testGetWorkoutRecordByUserId_Found
- testGetWorkoutRecordByUserId_NotFound

7.2 Integration Tests

- `testTraineeCanViewWorkoutPlanAssignedByTheirPT`
- `testPersonalTrainerCanViewTraineeWorkoutRecords`
- `testExercisesInsideWorkout4PlansAreOfSameStrategy`
- `testPTUpdatesWorkoutPlanAssignedToTrainee`
- `testDeleteAndEditWorkout4PlanFromWorkoutPlan`
- `testAddWorkoutSessionsWithMultipleExercisesToUserWorkoutRecord`

8 Future Enhancements

We are continuously looking to evolve Java-TrainingHub to offer even more value to trainees and personal trainers. Planned future enhancements include:

- **Graphical User Interface (GUI):** Development of a user-friendly graphical interface to enhance usability and provide a more intuitive experience, moving beyond the current command-line interface.
- **AI-Powered Workout Plan Generation:** Integration of artificial intelligence to suggest or generate personalized workout plans based on trainee progress, goals, historical performance, and common fitness principles.
- **Advanced Analytics & Reporting:** Implementation of more sophisticated data visualization and reporting features for trainees and trainers to track long-term progress, identify trends, and make data-driven decisions.
- **External API Integrations:** Exploring integrations with popular fitness trackers (e.g., wearable devices) or nutrition tracking services to provide a more holistic view of a user's health and fitness journey.
- **Cloud Deployment:** Investigating options for deploying the application to a cloud platform to enable broader accessibility and scalability.