UNIVERSITY OF CALGARY

Forward

An AI-Powered Fitness Management Application

Project Report

Prepared for:
SENG 401 - Software Architecture

Prepared by:
Benjamin Leggett
Himel Paul
Munem Morshed
Sadman Shahriar
Wahid Chowdhury

March 22, 2025

Contents

This document presents a comprehensive report for the Fitness Planner application, an innovative solution designed to help users create personalized workout plans using AI technology, track their fitness progress, and achieve their health goals.

Chapter 1

Requirements Document

1.1 Introduction

1.1.1 Project Overview

Forward is an innovative fitness planning application that leverages artificial intelligence to provide personalized workout and nutrition plans. The application aims to simplify fitness tracking, goal setting, and progress monitoring while providing intelligent recommendations tailored to individual needs and preferences. By integrating with a Large Language Model (LLM), Forward delivers customized workout plans, nutritional advice, and motivational support to help users achieve their health and fitness goals.

1.1.2 Target Audience

The primary target audience for Forward includes:

- Fitness enthusiasts seeking structured workout plans
- Individuals beginning their fitness journey who need guidance
- Busy professionals who need efficient workout planning
- Users looking to track their fitness progress over time
- People with specific fitness goals (weight loss, muscle gain, endurance improvement)
- Individuals seeking nutritional guidance alongside exercise planning

1.1.3 Key Features

Forward offers a comprehensive suite of features designed to enhance the fitness experience:

- AI-powered workout plan generation based on user goals and preferences
- Personalized nutrition tracking and recommendations

- Progress tracking with visual representations
- User authentication and profile management
- Exercise library with detailed instructions and demonstrations
- Goal setting and milestone tracking
- Cross-platform accessibility (desktop and mobile)

1.2 Functional Requirements

The following functional requirements have been implemented in the Forward application:

1. **User Registration and Authentication**
   Users can create accounts, log in securely, and manage their profiles with personal information and preferences.
2. **AI-Generated Workout Plans**
   The system connects to an LLM to generate personalized workout plans based on user goals, experience level, available equipment, and time constraints.
3. **Customizable User Profiles**
   Users can create and edit profiles with information such as age, weight, height, fitness goals, and exercise preferences.
4. **Exercise Library**
   The application provides a comprehensive database of exercises with descriptions, muscle groups targeted, and difficulty levels.
5. **Workout Scheduling**
   Users can schedule workouts on a calendar, receive reminders, and manage their weekly exercise routines.
6. **Progress Tracking**
   The system tracks and displays user progress through metrics like workout completion, weight changes, and performance improvements.
7. **Nutrition Logging**
   Users can log their daily food intake, track calories and macronutrients, and receive nutritional recommendations.
8. **Goal Setting**
   Users can set specific, measurable fitness goals and track their progress toward achieving them.
9. **Dashboard Visualization**
   The application provides visual representations of progress, workout history, and nutritional data through charts and graphs.

10. **Workout History**
    Users can access their complete workout history, including exercises performed, sets, reps, and weights used.

11. **AI-Powered Exercise Recommendations**
    The system suggests alternative exercises based on user preferences, available equipment, and fitness goals.

12. **Responsive Design**
    The application functions seamlessly across desktop and mobile platforms with an optimized user interface for each.

1.3 Non-Functional Requirements

The following non-functional requirements have been implemented in the Forward application:

1. **Performance**
   The system responds to user interactions within 2 seconds and generates AI workout plans within 10 seconds, even during peak usage periods.

2. **Security**
   User data is protected through encryption, secure authentication protocols, and adherence to data protection regulations. Passwords are hashed and sensitive information is secured with industry-standard protocols.

3. **Usability**
   The user interface follows modern design principles, providing intuitive navigation and clear visual hierarchy. The application maintains consistency across all screens and provides helpful tooltips and guidance.

4. **Reliability**
   The system maintains 99.9% uptime and includes error handling mechanisms to prevent data loss during connection failures. Regular backups ensure data persistence.

5. **Scalability**
   The architecture supports increasing user loads without performance degradation and can scale horizontally to accommodate growing user bases.

6. **Compatibility**
   The application is compatible with major browsers (Chrome, Firefox, Safari, Edge) and operates on both iOS and Android mobile platforms as well as desktop environments.

7. **Data Privacy**
   User fitness and nutrition data is handled according to privacy laws, with transparent data usage policies and user control over data sharing preferences.

1.4 User Stories and Use Cases

1.4.1 User Stories

- **As a new user**, I want to create an account so that I can start using the fitness planning features.
- **As a fitness beginner**, I want the AI to generate a beginner-friendly workout plan so that I can start my fitness journey safely and effectively.
- **As a busy professional**, I want to schedule workouts that fit my limited time availability so that I can maintain fitness despite my hectic schedule.
- **As a goal-oriented user**, I want to set specific fitness goals and track my progress so that I can stay motivated and see my improvements.
- **As a nutrition-conscious user**, I want to log my daily meals and receive nutritional insights so that I can align my diet with my fitness goals.
- **As an experienced athlete**, I want to customize my AI-generated workout plans so that they match my advanced fitness level and specific training needs.
- **As a mobile user**, I want to access the application on my smartphone so that I can track workouts and nutrition while on the go.

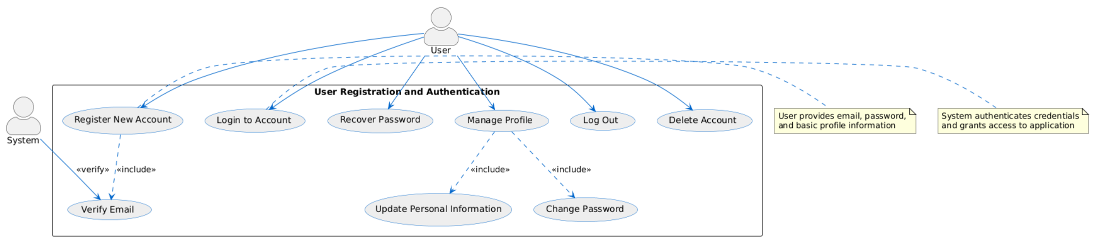
User
System
User Registration and Authentication
Register New Account
Login to Account
Recover Password
Manage Profile
Log Out
Delete Account
«verify»
«include»
Verify Email
«include»
«include»
Update Personal Information
Change Password
User provides email, password, and basic profile information
System authenticates credentials and grants access to application

Figure 1.1: User Registration and Authentication Use Case Diagram

1.4.2 Key Use Cases

Generate AI Workout Plan

Primary Actor: Registered User
Preconditions: User is logged in and has completed profile information
Main Success Scenario:

1. User navigates to workout plan generation page
2. User selects fitness goals (e.g., weight loss, muscle gain)
3. User specifies available equipment
4. User indicates time constraints and workout preferences

5. User requests AI-generated plan
6. System communicates with LLM to process requirements
7. LLM generates personalized workout plan
8. System displays the workout plan to the user
9. User saves the workout plan to their profile

Alternative Flows:

- If the LLM connection fails, the system offers pre-designed workout templates as alternatives
- If user profile is incomplete, system prompts user to complete required information

Postconditions: User has a personalized workout plan saved to their profile

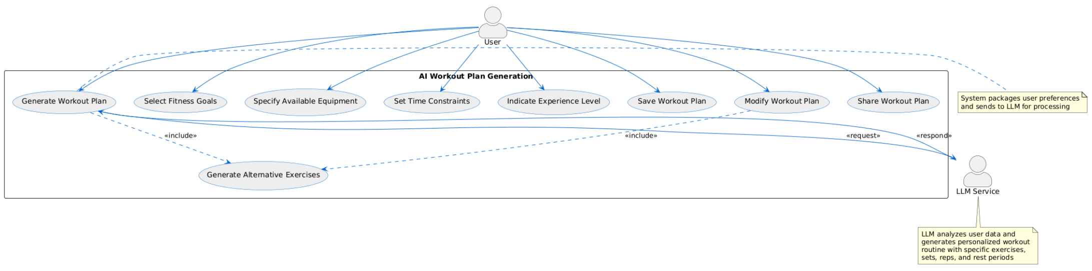
User
AI Workout Plan Generation
Generate Workout Plan
Select Fitness Goals
Specify Available Equipment
Set Time Constraints
Indicate Experience Level
Save Workout Plan
Modify Workout Plan
Share Workout Plan
System packages user preferences and sends to LLM for processing
«include»
«include»
«request»
«respond»
Generate Alternative Exercises
LLM Service
LLM analyzes user data and generates personalized workout routine with specific exercises, sets, reps, and rest periods

Figure 1.2: Detailed AI Workout Plan Generation Use Case Diagram

Track Nutrition and Receive Recommendations

Primary Actor: Registered User
Preconditions: User is logged in
Main Success Scenario:

1. User navigates to nutrition tracking section
2. User logs food consumed for a specific meal
3. System calculates and displays nutritional information
4. System analyzes nutritional data against user's goals
5. System requests LLM to generate nutritional recommendations
6. LLM provides personalized nutrition advice
7. System displays recommendations to the user

Alternative Flows:

- If food item is not in database, user can add custom food entries with nutritional information

- If daily nutrition goals are exceeded, system provides alerts and adjustment suggestions

Postconditions: User's nutrition is logged and recommendations are provided

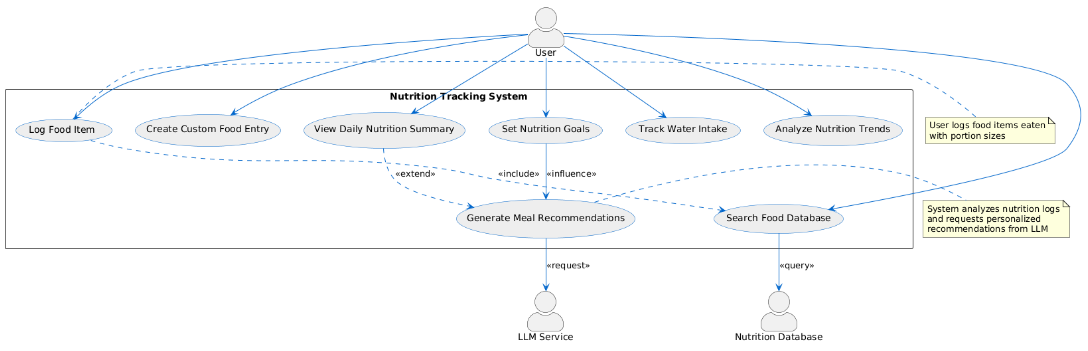
User
Nutrition Tracking System
Log Food Item
Create Custom Food Entry
View Daily Nutrition Summary
Set Nutrition Goals
Track Water Intake
Analyze Nutrition Trends
User logs food items eaten with portion sizes
«extend»
«include»
«influence»
Generate Meal Recommendations
Search Food Database
System analyzes nutrition logs and requests personalized recommendations from LLM
«request»
«query»
LLM Service
Nutrition Database

Figure 1.3: Nutrition Tracking Use Case Diagram

Set and Track Fitness Goals

Primary Actor: Registered User
Preconditions: User is logged in
Main Success Scenario:

1. User navigates to goals section
2. User creates a new fitness goal with specific metrics
3. User sets target date for goal achievement
4. System stores goal information
5. System tracks relevant metrics from user's activities
6. System displays progress visualization toward goal
7. System provides motivational feedback based on progress

Alternative Flows:

- If goal appears unrealistic, system suggests modifications based on user's history
- If goal deadline is approaching with insufficient progress, system offers adjusted plans

Postconditions: User's goal is established and tracking begins

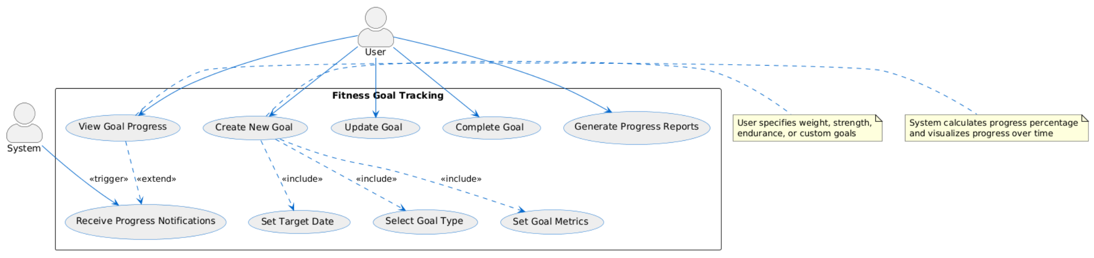
User
Fitness Goal Tracking
System
View Goal Progress
Create New Goal
Update Goal
Complete Goal
Generate Progress Reports
«trigger»
«extend»
«include»
«include»
«include»
Receive Progress Notifications
Set Target Date
Select Goal Type
Set Goal Metrics
User specifies weight, strength, endurance, or custom goals
System calculates progress percentage and visualizes progress over time

Figure 1.4: Fitness Goal Tracking Use Case Diagram

Workout Scheduling

Primary Actor: Registered User
Preconditions: User is logged in and has at least one workout plan
Main Success Scenario:

1. User navigates to calendar/scheduling section
2. User selects a date and time for the workout
3. User chooses a workout plan to schedule
4. User sets optional reminder settings
5. System saves the scheduled workout
6. System sends reminders according to user preferences
7. User receives notification when workout is due

Alternative Flows:

- If user wants recurring workouts, system creates pattern of scheduled sessions
- If time slot conflicts with existing event, system alerts user and suggests alternatives

Postconditions: Workout is scheduled and reminders are set

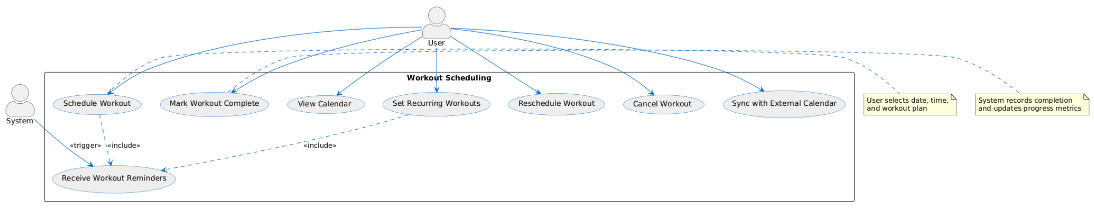

Figure 1.5: Workout Scheduling Use Case Diagram

Progress Tracking and Dashboard

Primary Actor: Registered User
Preconditions: User is logged in and has recorded activity data
Main Success Scenario:

1. User navigates to dashboard section
2. System retrieves user's workout history, nutrition logs, and goal progress
3. System generates visual representations of data (charts, graphs)
4. User views overall progress metrics and trends
5. User can filter data by time periods or activity types
6. System highlights achievements and milestones

Alternative Flows:

- If insufficient data exists, system provides placeholders and suggestions for tracking
- User can export progress reports for external use

Postconditions: User views comprehensive progress visualization

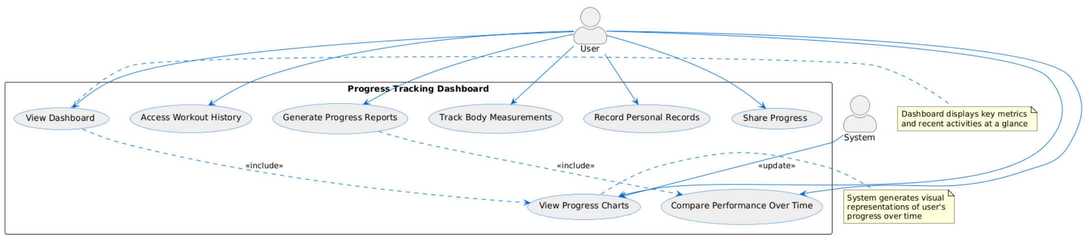
User
Progress Tracking Dashboard
View Dashboard
Access Workout History
Generate Progress Reports
Track Body Measurements
Record Personal Records
Share Progress
System
Dashboard displays key metrics and recent activities at a glance
«include»
«include»
«update»
View Progress Charts
Compare Performance Over Time
System generates visual representations of user's progress over time

Figure 1.6: Progress Tracking and Dashboard Use Case Diagram

Chapter 2

Design Document

2.1 System Architecture Overview

The Forward application follows a modern web application architecture with clearly separated frontend and backend components. This separation allows for independent development, testing, and deployment while maintaining a cohesive user experience.

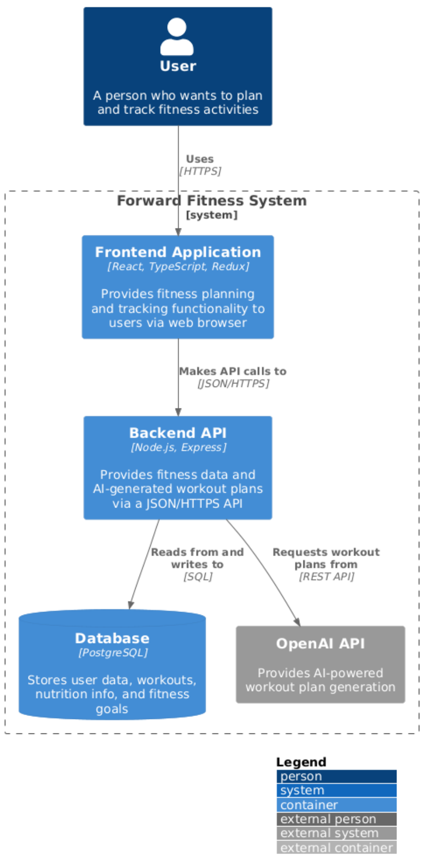
User
A person who wants to plan and track fitness activities
Uses
[HTTPS]
Forward Fitness System
[system]
Frontend Application
[React, TypeScript, Redux]
Provides fitness planning and tracking functionality to users via web browser
Makes API calls to
[JSON/HTTPS]
Backend API
[Node.js, Express]
Provides fitness data and AI-generated workout plans via a JSON/HTTPS API
Reads from and writes to
[SQL]
Requests workout plans from
[REST API]
Database
[PostgreSQL]
Stores user data, workouts, nutrition info, and fitness goals
OpenAI API
Provides AI-powered workout plan generation
Legend
person
system
container
external person
external system
external container

Figure 2.1: Overall System Architecture

2.1.1 Frontend Architecture

The Forward frontend is built using React with TypeScript, providing type safety and improved developer experience. The application employs a feature-based architecture organized around business domains rather than technical concerns:

- **Component-Based Structure:** UI elements are organized as reusable React components
- **Redux State Management:** Application state is managed using Redux Toolkit with slice-based organization
- **Responsive Design:** Material-UI components ensure the application functions correctly on both desktop and mobile devices
- **Route-Based Code Splitting:** Features are loaded on demand for optimized performance

The frontend architecture follows a unidirectional data flow pattern, where user interactions trigger actions that modify the Redux store, which then updates the UI components through React's rendering cycle.

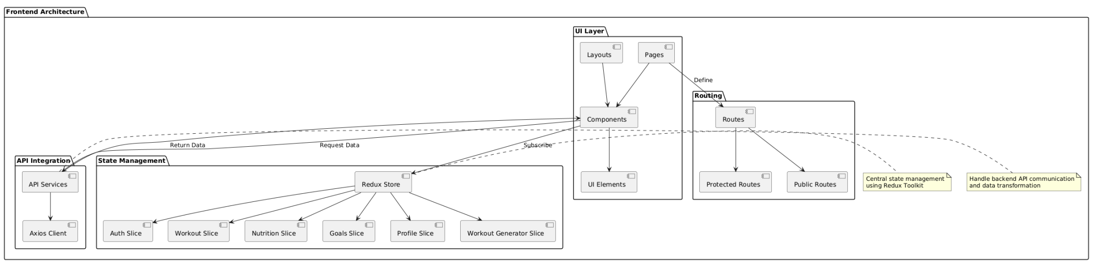

Figure 2.2: Frontend Architecture

2.1.2 Backend Architecture

The backend is implemented as a RESTful API server using Node.js and Express. It follows a layered architecture that separates concerns and promotes maintainability:

- **Routes Layer:** Defines API endpoints and maps HTTP requests to appropriate controllers
- **Controllers Layer:** Handles request validation, orchestrates service calls, and prepares responses
- **Services Layer:** Contains business logic and interacts with data access layer
- **Models Layer:** Represents data entities and interfaces with the database
- **Middleware:** Provides cross-cutting functionality (authentication, error handling, logging)
- **Configuration:** Manages environment-specific settings and external integrations

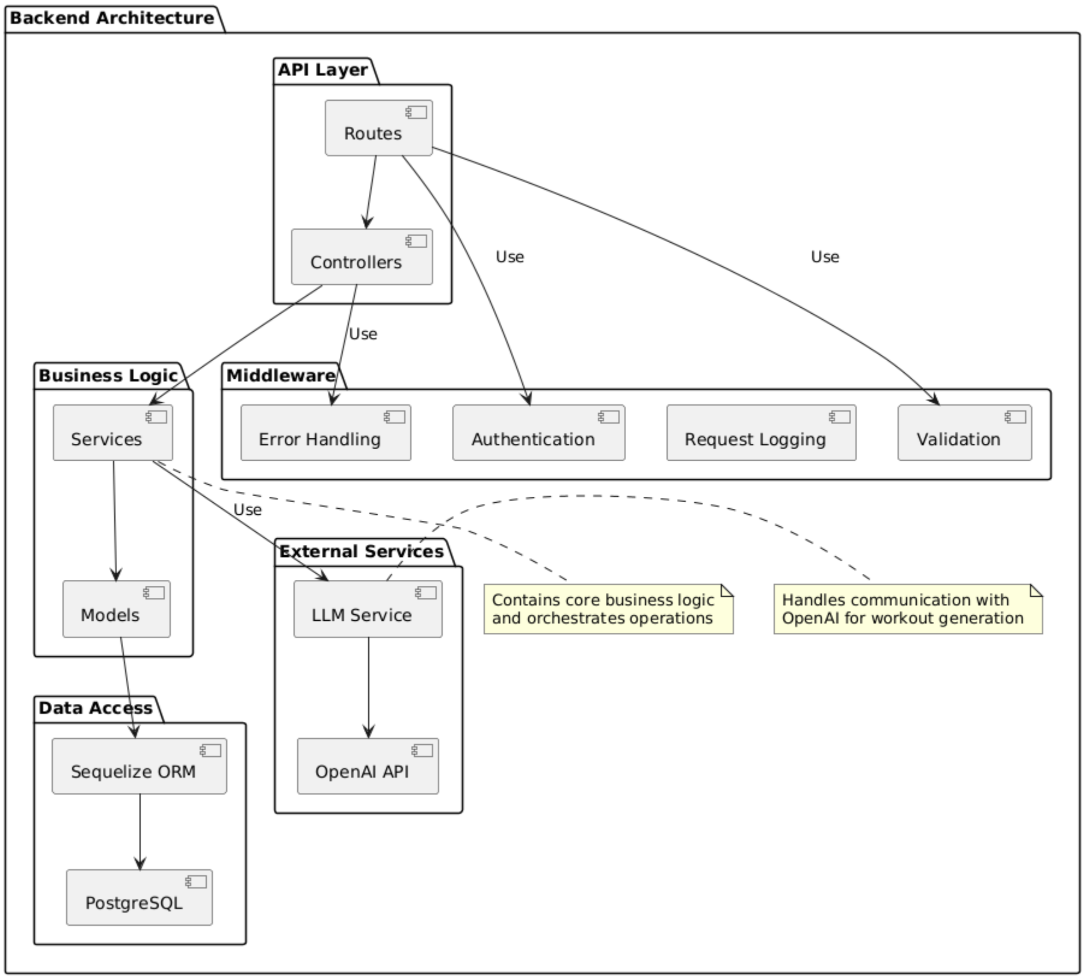

Figure 2.3: Backend Architecture

2.1.3 Database Design

Forward uses a PostgreSQL relational database with Sequelize ORM for data persistence. The database schema is designed to efficiently store user data, workouts, nutrition information, and fitness goals:

- **Users:** Stores user account information, credentials, and profile data
- **Workouts:** Contains workout plans and sessions
- **Exercises:** Stores exercise details linked to workouts
- **Nutrition:** Records meal information and nutritional data
- **FitnessGoals:** Maintains user-defined fitness goals and progress

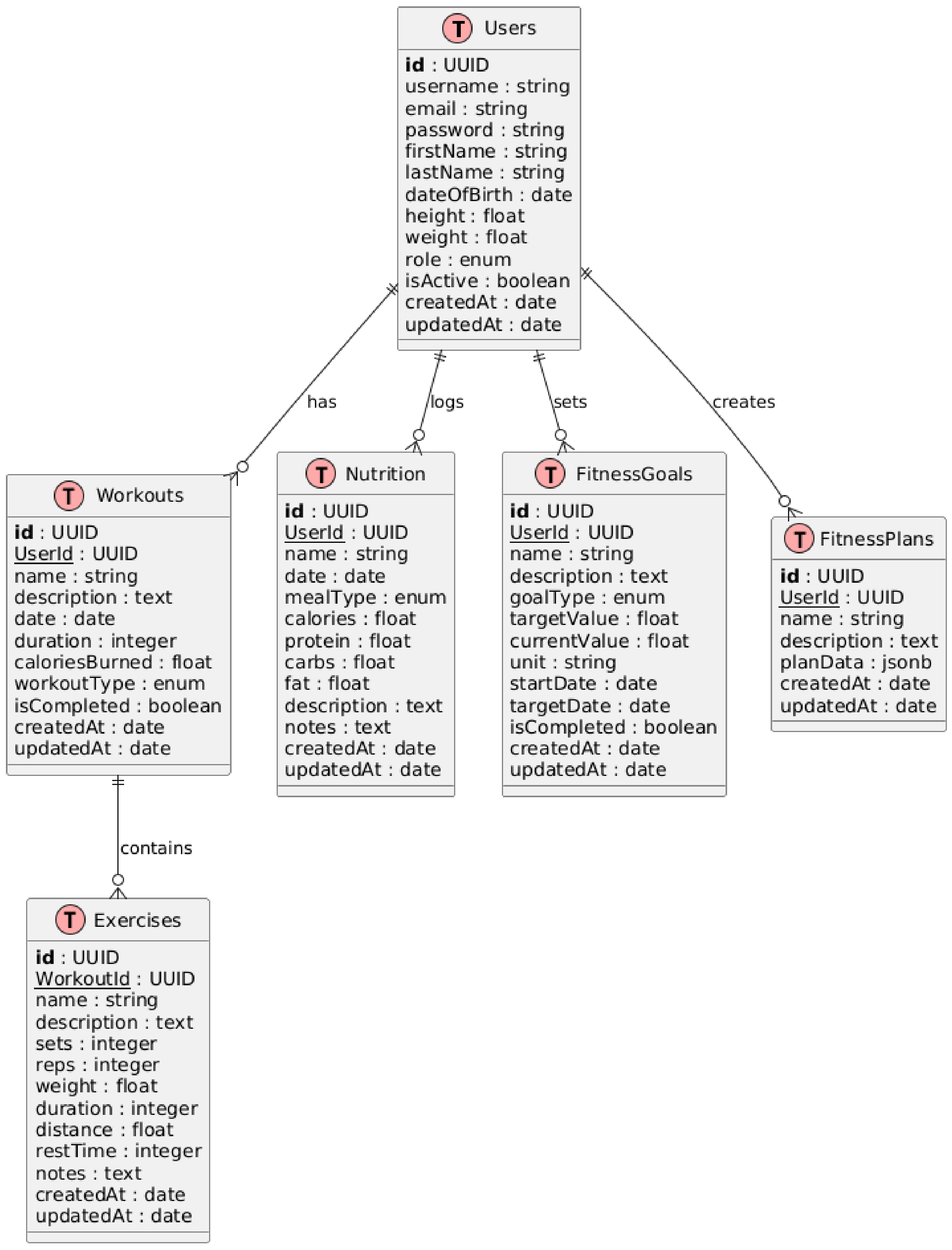

Figure 2.4: Database Schema

2.1.4 LLM Integration

Forward integrates with OpenAI's API to provide AI-powered workout plan generation. The integration architecture is designed to be modular and extensible:

- **LLM Service:** Encapsulates API communication with OpenAI
- **AI Controller:** Processes user preferences and handles LLM-generated responses
- **Request Formatting:** Transforms user preferences into effective prompts
- **Response Processing:** Parses LLM output into structured workout plans

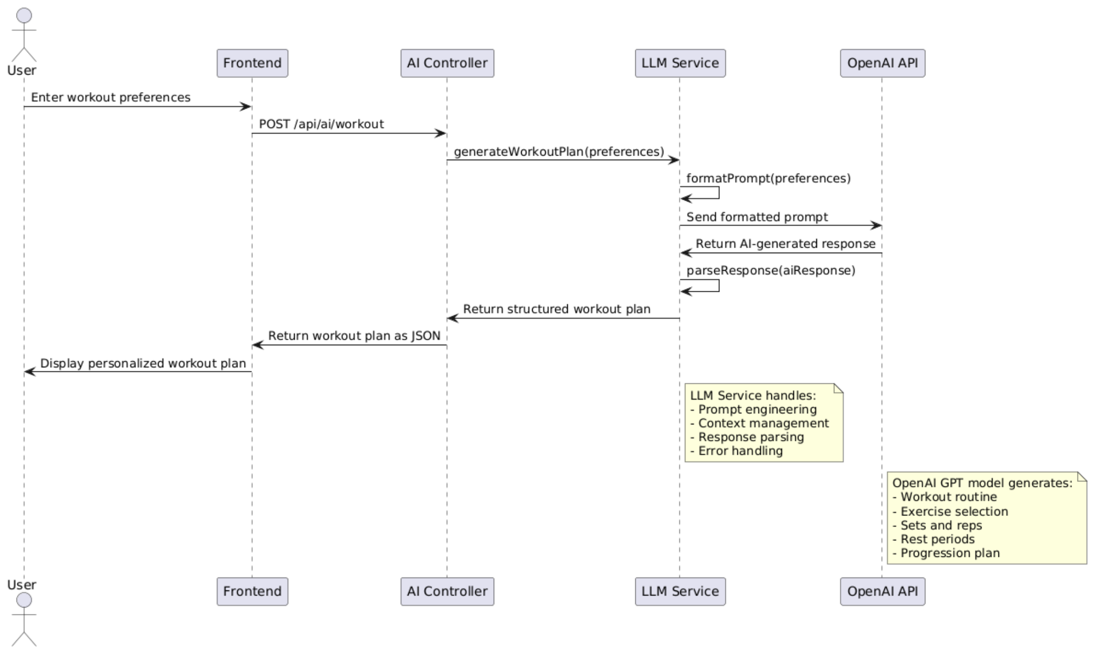

Figure 2.5: LLM Integration Architecture

2.2 Class/Component Diagrams

2.2.1 Backend Class Diagram

The backend follows an object-oriented approach with clear class responsibilities and relationships:

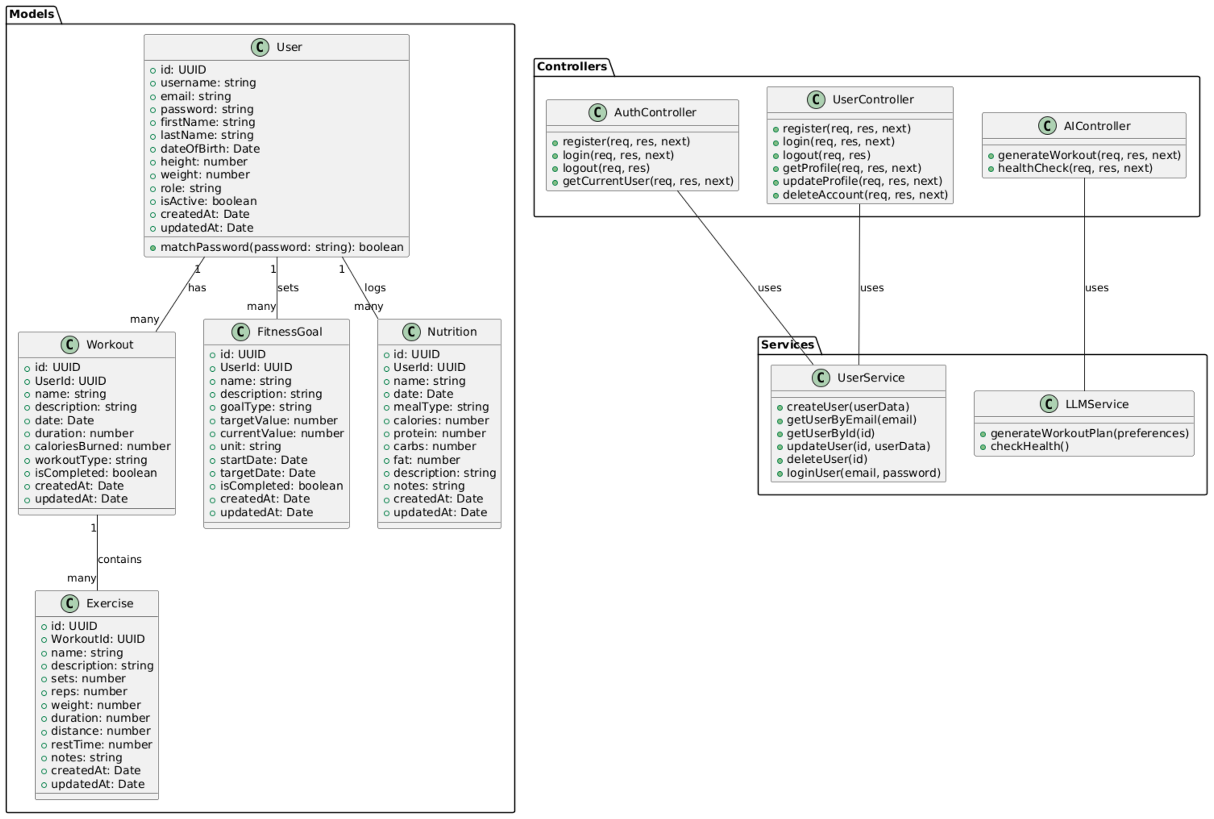

Figure 2.6: Backend Class Diagram

2.2.2 Frontend Component Diagram

The React frontend is structured around components with clear hierarchies and data flows:

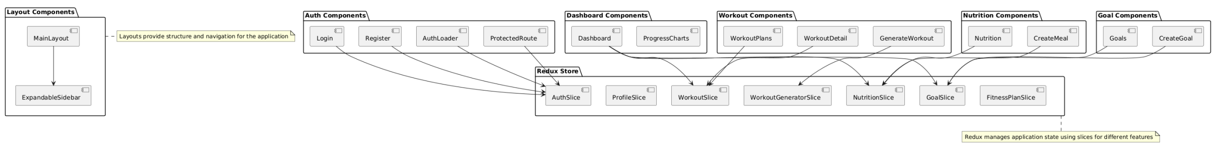

Figure 2.7: Frontend Component Diagram

2.2.3 Key Data Models

The application's core data models are designed to efficiently represent fitness domain concepts:

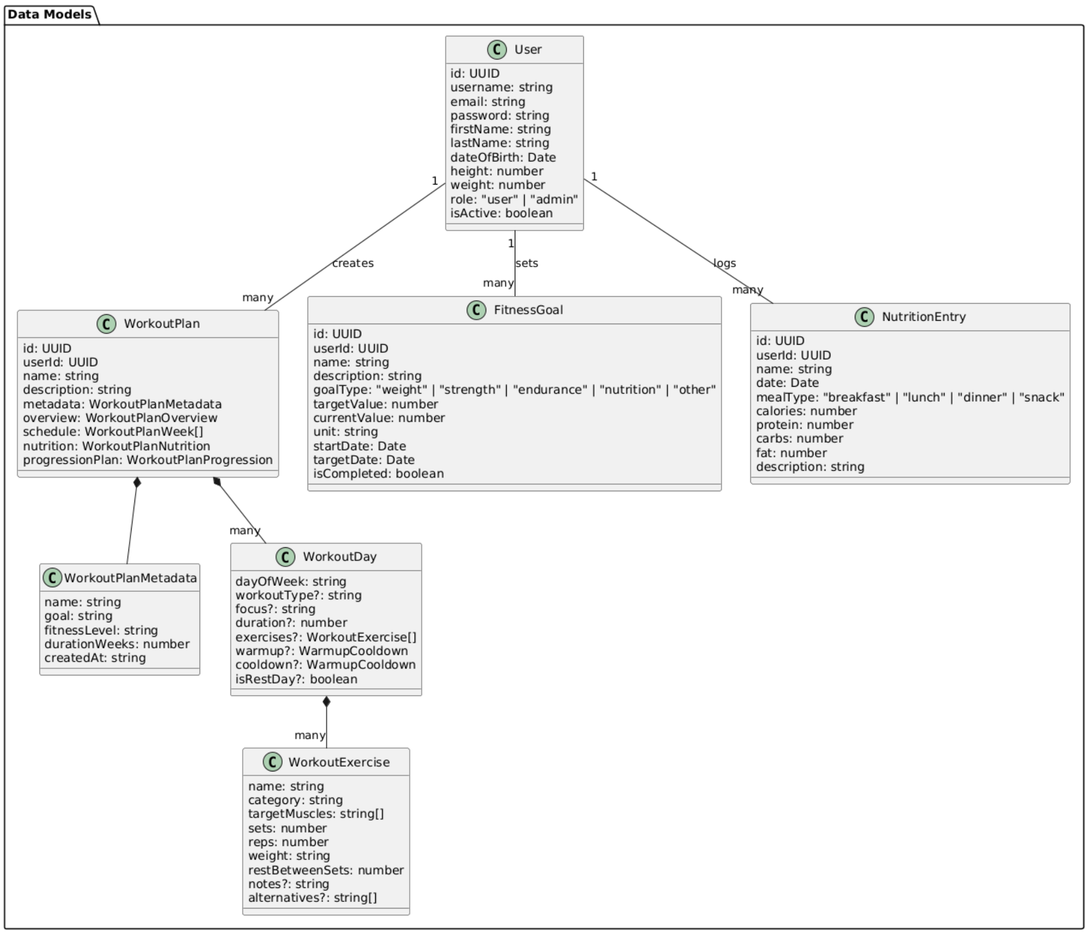

Figure 2.8: Key Data Models

2.3 Design Patterns Used

The Forward application implements several established design patterns to ensure code maintainability, scalability, and adherence to best practices.

2.3.1 MVC Pattern

The Model-View-Controller (MVC) pattern is implemented throughout the application to separate concerns and improve maintainability:

- **Model:** Sequelize models (User, Workout, Exercise, etc.) define the data structure and database interactions
- **View:** React components in the frontend represent the view layer, rendering data to users
- **Controller:** Express controllers (UserController, AIController, AuthController) handle incoming requests and coordinate responses

This separation allows independent development and testing of each layer while maintaining a clear flow of data and control.

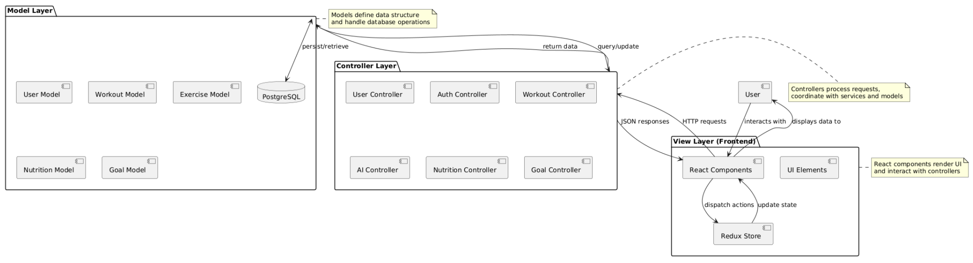

Figure 2.9: MVC Pattern Implementation

Implementation example from the UserController:

```
// Controller handling HTTP requests
exports.register = async (req, res, next) => {
  try {
    logger.info('Registration attempt received');
    const userData = req.body;

    // Call service layer (business logic)
    const result = await userService.createUser(userData);

    // Prepare and send response (view interaction)
    res.status(201).json({
      success: true,
      message: 'User registered successfully',
      data: result
    });
  } catch (error) {
    // Error handling
    next(error);
  }
};
```

2.3.2 Repository Pattern

The Repository pattern is implemented through Sequelize ORM, providing a clean abstraction over the database operations:

- **Repositories:** Sequelize models act as repositories, encapsulating data access logic
- **Service Layer:** Service classes use repositories to perform business operations
- **Domain Objects:** JavaScript objects represent the domain entities without exposing database details

This pattern isolates the application from database implementation details and enables easier testing with mock repositories.

Implementation example from the database configuration:

```
// Define DB object and register models
const db = {};
db.sequelize = sequelize;
db.Sequelize = Sequelize;

// Import models
db.User = require('../models/User')(sequelize, Sequelize.DataTypes);
db.Workout = require('../models/Workout')(sequelize);
db.Exercise = require('../models/Exercise')(sequelize, Sequelize.DataTypes);
// ...more models

// Setup Associations
Object.keys(db).forEach((modelName) => {
  if (db[modelName].associate) {
    db[modelName].associate(db);
  }
});
```

2.3.3 Observer Pattern

The Redux implementation in the frontend follows the Observer pattern:

- **Observable (Store):** The Redux store maintains the application state
- **Observers (Components):** React components subscribe to state changes
- **Actions:** Events that trigger state updates
- **Reducers:** Pure functions that determine how state changes in response to actions

This pattern creates a unidirectional data flow, making state changes predictable and trackable.

Implementation example from the Redux store configuration:

```
// Store configuration with multiple slices/reducers
export const store = configureStore({
  reducer: {
    auth: authReducer,
    profile: profileReducer,
    workouts: workoutsReducer,
    nutrition: nutritionReducer,
    goals: goalsReducer,
    workoutGenerator: workoutGeneratorReducer,
    fitnessPlans: fitnessPlanReducer,
  },
  middleware: (getDefaultMiddleware) =>
    getDefaultMiddleware({
      serializableCheck: {
        ignoredActionPaths: ['meta.arg', 'payload.date', ...],
      },
    }),
});
```

2.3.4 Middleware Pattern

The Express.js backend extensively uses the Middleware pattern for cross-cutting concerns:

- **Authentication:** Verifies user credentials and session validity
- **Error Handling:** Centralizes error processing and client responses
- **Logging:** Records application activities and requests
- **Request Parsing:** Transforms incoming data into application formats

This pattern creates a pipeline for request processing, with each middleware focusing on a specific aspect of request handling.

Implementation example from the Express application setup:

```
// Middleware setup in Express
app.use(cors({
  origin: process.env.CLIENT_URL || 'http://localhost:3000',
  credentials: true,
  methods: ['GET', 'POST', 'PUT', 'DELETE', 'OPTIONS'],
  allowedHeaders: ['Content-Type', 'Authorization']
}));

app.use(express.json({ limit: '2mb' }));
app.use(express.urlencoded({ extended: true }));
app.use(cookieParser());

// Development logging
if (process.env.NODE_ENV === 'development') {
  app.use(morgan('dev'));
}

// Error handling middleware
app.use(notFound);
app.use(errorHandler);
```

2.3.5 Factory Pattern

The application employs the Factory Pattern for creating model instances, particularly in the database configuration:

- **Model Creation:** Models are instantiated through factory functions
- **Configuration Separation:** Creation logic is separated from the model's business logic
- **Consistent Initialization:** Ensures models are created with proper configuration

Implementation example from model initialization:

```
// Models are created through factory functions
db.User = require('../models/User')(sequelize, Sequelize.DataTypes);
db.Workout = require('../models/Workout')(sequelize);
db.Exercise = require('../models/Exercise')(sequelize, Sequelize.DataTypes);
```

2.3.6 Command Pattern

The Redux implementation in the application follows the Command Pattern:

- **Actions as Commands:** Redux actions encapsulate operations as objects
- **Command Dispatch:** Actions are dispatched to execute operations
- **State Transformation:** Reducers execute the commands on the state

Example from workout generation functionality:

```
// Creating a command to generate a workout
export const generateWorkout = createAsyncThunk(
  'workoutGenerator/generate',
  async (preferences: WorkoutPreferences, { rejectWithValue, getState }) => {
    try {
      const state: any = getState();
      const userProfile = state.profile?.profile || {};

      const preferencesWithProfile = {
        ...preferences,
        userProfile: {
          weight: userProfile.weight || null,
          height: userProfile.height || null,
          // More properties...
        }
      };

      // Execute the command by calling the service
      const response = await generateWorkoutPlan(preferencesWithProfile);
      return response;
    } catch (error: any) {
      return rejectWithValue(error.message || 'Failed to generate workout plan');
    }
  }
);
```

2.3.7 Adapter Pattern

The LLM integration service acts as an adapter, normalizing data between the application and the external API:

- **Interface Translation:** Converts internal data formats to LLM-compatible requests
- **Response Transformation:** Processes LLM outputs into application-compatible formats
- **Error Normalization:** Standardizes error handling across different APIs

Example from the LLM service:

```
function createWorkoutPlanPrompt(preferences) {
  // Convert application preferences to LLM prompt format
  const { fitnessGoal, experienceLevel, workoutDaysPerWeek /* etc */ } = preferences;

  // Create formatted prompt for the external API
  return `
You are an expert fitness coach creating a personalized workout plan...
  `;
}
```

2.3.8 Composite Pattern

The React component structure utilizes the Composite Pattern:

- **Component Hierarchy:** Components can contain other components
- **Uniform Treatment:** Parent components work with child components through consistent interfaces
- **Recursive Composition:** Components can be nested to any depth

Example from the UI component structure:

```
// A composite component that contains other components
const WorkoutPlan: React.FC<WorkoutPlanProps> = ({ workoutPlan, onSave, onEdit, readOn
  return (
    <Card>
      <CardHeader title={workoutPlan.metadata.name} />
      <CardContent>
        {/* Child components are composed together */}
        <Box>
          <Typography>{workoutPlan.overview.description}</Typography>
          <EquipmentList equipment={workoutPlan.overview.recommendedEquipment} />
        </Box>
        {/* Week schedule as a composite of day components */}
        {workoutPlan.schedule.map((week) => (
          <WeekSchedule key={week.week} weekData={week} />
        ))}
      </CardContent>
    </Card>
  );
};
```

2.3.9 Decorator Pattern

Express middleware functions in the backend operate as decorators:

- **Function Enhancement:** Middleware adds functionality to route handlers
- **Transparent Wrapping:** Original route handlers are unaware of middleware
- **Dynamic Behavior Addition:** Authentication, logging, and validation are added without modifying routes

Example from route handling with middleware:

```
// Authentication middleware decorates the route handler
router.post('/workouts',
  authenticate,                   // Decorator for authentication
  validate(workoutSchema),        // Decorator for validation
  logRequest,                     // Decorator for logging
  workoutController.createWorkout // The core route handler being decorated
);
```

2.4 SOLID Principles Application

The Forward application adheres to SOLID principles to ensure maintainable, extensible, and testable code.

2.4.1 Single Responsibility Principle

Each class and module in the application has a single responsibility:

- **Controllers:** Handle HTTP requests and responses only
- **Services:** Contain business logic separate from HTTP concerns
- **Models:** Represent data structure and database operations
- **Routes:** Define API endpoints without handling business logic

Example from the AIController that delegates workout generation to a specialized service:

```
exports.generateWorkout = async (req, res, next) => {
  try {
    const preferences = req.body;

    if (!preferences) {
      throw new ApiError(400, 'Workout preferences are required');
    }

    logger.info(`Generating workout plan with preferences: `, preferences);

    // Call LLM service - each service has a single responsibility
    const workoutPlan = await llmService.generateWorkoutPlan(userPreferences);

    res.status(200).json(workoutPlan);
  } catch (error) {
    logger.error(`Workout generation error: ${error.message}`);
    next(error);
  }
};
```

2.4.2 Open/Closed Principle

The application is designed to be open for extension but closed for modification:

- **Redux Slices:** New features can be added as new slices without modifying existing ones
- **API Routes:** New endpoints can be added without changing existing route handlers
- **Middleware:** Additional middleware can be inserted into the request pipeline
- **Models:** Sequelize allows extending models through associations and hooks

Example from the routes configuration that allows adding new routes without modifying existing ones:

```
// API routes
app.use('/api', apiRoutes);
app.use('/api/auth', authRoutes);
app.use('/api/users', userRoutes);
app.use('/api/workouts', workoutRoutes);
app.use('/api/nutrition', nutritionRoutes);
app.use('/api/goals', goalRoutes);
app.use('/api/health-check', healthRoutes);
app.use('/api/ai', aiRoutes);
app.use('/api/fitnessPlan', fitnessPlanRoutes);
```

2.4.3 Liskov Substitution Principle

The application uses inheritance and polymorphism in a way that subclasses can be used in place of their parent classes:

- **Error Classes:** Custom error types extend the base Error class while maintaining expected behavior
- **Component Inheritance:** React components use composition patterns for reuse
- **Route Handlers:** Different route handlers maintain consistent response formats

Example of extending the Error class while maintaining substitutability:

```
// ApiError can be used anywhere a standard Error is expected
class ApiError extends Error {
  constructor(statusCode, message, errors = []) {
    super(message);
    this.statusCode = statusCode;
    this.errors = errors;
    this.name = 'ApiError';
  }
}
```

2.4.4 Interface Segregation Principle

The application uses fine-grained interfaces rather than general-purpose ones:

- **React Props:** Components accept only the props they need
- **API Endpoints:** Each endpoint accepts only the data relevant to its function
- **Redux Selectors:** Select only the state slices needed by components

Example of a component with focused props interface:

```
// Component only requires the specific props it needs
interface WorkoutPlanProps {
  workoutPlan: WorkoutPlanData;
  onSave?: (plan: WorkoutPlanData) => void;
  onEdit?: (planId: string) => void;
  readOnly?: boolean;
}

const WorkoutPlan: React.FC<WorkoutPlanProps> = ({
  workoutPlan,
  onSave,
  onEdit,
  readOnly = false
}) => {
  // Component implementation
};
```

2.4.5 Dependency Inversion Principle

The application depends on abstractions rather than concrete implementations:

- **Service Injection:** Controllers depend on service interfaces, not implementations
- **Redux Hooks:** Components use hooks to access store without direct dependencies
- **Configuration:** Environment-specific settings are abstracted through configuration objects

Example of dependency abstraction through service injection:

```
// Controller depends on the interface of the service, not its implementation
const userService = require('../services/userService');
const llmService = require('../services/llmService');

// The controller doesn't know how these services are implemented internally
exports.generateWorkout = async (req, res, next) => {
  try {
    const preferences = req.body;
    const workoutPlan = await llmService.generateWorkoutPlan(preferences);
    res.status(200).json(workoutPlan);
  } catch (error) {
    next(error);
  }
};
```

Chapter 3

Testing Document

3.1 Testing Methodology

3.1.1 Testing Approach

The Forward application implements a pragmatic testing strategy focusing on key application functionality. Our approach prioritizes testing critical paths and core features, particularly the AI workout generation functionality that serves as the centerpiece of our application:

- **Script-Based Testing:** Automated scripts that verify essential functionality
- **API Endpoint Verification:** Tests ensuring our endpoints respond correctly
- **Infrastructure Validation:** Tests confirming our application structure is correct
- **Component Testing:** Tests for React UI components

Our testing methodology emphasizes validating the LLM integration, which is the most critical and complex aspect of our application.

3.1.2 Tools and Frameworks

The project leverages the following testing tools:

- **Jest:** For component and unit testing
- **React Testing Library:** For testing React components
- **Axios:** For testing API endpoints directly
- **Node.js Scripts:** For custom testing functionality
- **Redux Mock Store:** For testing Redux state management

3.2 Testing Implementation

3.2.1 Infrastructure Testing

We implemented a directory structure validation script to ensure that all required directories and files exist before the application runs:

```
// Directory Structure Check
const requiredFiles = [
  'app.js',
  'server.js',
  'controllers/aiController.js',
  'routes/aiRoutes.js',
  'services/llmService.js',
  'utils/logger.js',
  'utils/errors.js'
];

const requiredDirs = [
  'config',
  'controllers',
  'db',
  'middleware',
  'models',
  'routes',
  'services',
  'utils'
];

// Check directories
for (const dir of requiredDirs) {
  const dirPath = path.join(BASE_DIR, dir);
  if (fs.existsSync(dirPath) && fs.statSync(dirPath).isDirectory()) {
    console.log(‘ Directory exists: ${dir}‘);
  } else {
    console.log(‘ Missing directory: ${dir}‘);
  }
}
```

This ensures that our application's structure remains consistent, which is particularly important for the modular architecture we've implemented.

3.2.2 API Endpoint Testing

We created a dedicated script to test the critical API endpoints, particularly focusing on the workout generation functionality:

```
async function runTests() {
  log.header('API ENDPOINTS TEST');
  log.info(`Testing against API at ${API_URL}`);
  const startTime = Date.now();
  const results = {};

  // Test server health endpoint
  log.header('1. Server Health Check');
  results.serverHealth = await testEndpoint('Server Health', 'GET', '/health');

  // Test AI health endpoint
  log.header('2. AI Service Health Check');
  results.aiHealth = await testEndpoint('AI Service Health', 'GET', '/api/ai/health')

  // Test workout generation endpoint
  log.header('3. Workout Plan Generation');
  results.workoutGeneration = await testEndpoint(
    'Workout Generation',
    'POST',
    '/api/ai/workout',
    sampleWorkoutPreferences
  );

  // Summarize results
  log.header('TEST SUMMARY');
  const duration = ((Date.now() - startTime) / 1000).toFixed(2);
  const allSuccess = Object.values(results).every(r => r.success);
}
```

This testing approach allows us to quickly verify that our endpoints are functioning correctly and producing the expected responses.

3.2.3 LLM Integration Testing

The most critical part of our application is the LLM integration for workout plan generation. We developed a dedicated testing script for this functionality:

```
async function testLlmConnection() {
  console.log('\n--- Testing direct LLM connection ---');

  try {
    console.log(`Sending request to: ${LLM_URL}`);
    const response = await axios.post(LLM_URL, {
      model: 'mistral-7b-instruct-v0.2',
      messages: [
        {
          role: 'user',
          content: 'Respond with "LLM service is working properly"'
        }
      ],
      temperature: 0.7,
      max_tokens: 50
    }, {
      timeout: 10000, // 10 seconds timeout
      headers: { 'Content-Type': 'application/json' }
    });

    console.log('LLM Response:', response.data.choices[0].message.content);
    console.log(' Direct LLM connection successful');

    return true;
  } catch (error) {
    console.error(' LLM connection failed:', error.message);
    return false;
  }
}

async function testWorkoutGeneration() {
  console.log('\n--- Testing Workout Generation ---');

  try {
    const response = await axios.post(`${API_URL}/api/ai/workout`, samplePreferences)

    console.log(' Workout generation successful');
    console.log('Generated Plan Type:', typeof response.data);

    // Save the generated plan to a file for inspection
    const outputFile = path.join(OUTPUT_DIR, `workout-plan-${Date.now()}.json`);
    fs.writeFileSync(outputFile, JSON.stringify(response.data, null, 2));
    console.log(`Workout plan saved to: ${outputFile}`);

    return true;
  } catch (error) {
    console.error(' Workout generation failed:', error.message);
    return false;
  }
}
```

This test verifies both the direct connection to the LLM service and the workout generation functionality through our API, which is the core value proposition of our application.

3.2.4 Database Testing

For database testing, we implemented a configuration script that creates a clean test database environment:

```
async function setupTestDb() {
  try {
    // Sync database with force flag to recreate tables
    await sequelize.sync({ force: true });
    logger.info('Test database synced successfully');
  } catch (error) {
    logger.error(`Error setting up test database: ${error.message}`);
    throw error;
  }
}

async function teardownTestDb() {
  try {
    await sequelize.close();
    logger.info('Test database connection closed successfully');
  } catch (error) {
    logger.error(`Error closing test database connection: ${error.message}`);
    throw error;
  }
}
```

This ensures that our tests can run against a clean database instance, providing isolation between test runs.

3.2.5 Frontend Component Testing

For frontend testing, we implemented tests for key React components using Jest and React Testing Library. Here's an example of our test for the WorkoutForm component:

```
describe('WorkoutForm Component', () => {
  const mockOnSubmit = jest.fn();
  const mockOnCancel = jest.fn();

  beforeEach(() => {
    jest.clearAllMocks();
  });

  it('renders the form with empty fields', () => {
    render(
      <WorkoutForm
        onSubmit={mockOnSubmit}
        onCancel={mockOnCancel}
      />
    );

    // Check form elements are present
    expect(screen.getByLabelText(/Workout Name/i)).toBeInTheDocument();
    expect(screen.getByLabelText(/Description/i)).toBeInTheDocument();
    expect(screen.getByLabelText(/Duration/i)).toBeInTheDocument();
    expect(screen.getByLabelText(/Date/i)).toBeInTheDocument();
    expect(screen.getByLabelText(/Workout Type/i)).toBeInTheDocument();
  });

  it('submits the form with valid data', async () => {
    render(
      <WorkoutForm
        onSubmit={mockOnSubmit}
        onCancel={mockOnCancel}
      />
    );

    // Fill form fields
    fireEvent.change(screen.getByLabelText(/Workout Name/i), {
      target: { value: 'Evening Workout' }
    });

    // Submit form
    fireEvent.click(screen.getByRole('button', { name: /Save/i }));

    // Check if onSubmit was called with correct data
    await waitFor(() => {
      expect(mockOnSubmit).toHaveBeenCalled();
    });
  });
}
```

We also implemented tests for the workout generation components which are critical to our application's functionality:

```
describe('GenerateWorkout Component', () => {
  let store: any;

  beforeEach(() => {
    // Reset all mocks before each test
    mockGenerateWorkout.mockReset();
    mockResetWorkoutGenerator.mockReset();
    mockSetPreferences.mockReset();
    mockSaveGeneratedWorkout.mockReset();
    store = mockStore({
      workoutGenerator: {
        loading: false,
        error: null,
        workoutPlan: null,
        success: false
      }
    });
  });

  it('renders the component with form fields', () => {
    render(
      <Provider store={store} children={undefined}>
        <GenerateWorkout />
      </Provider>
    );

    // Check if form elements are present
    expect(screen.getByText(/Generate Custom Workout Plan/i)).toBeInTheDocument();
    expect(screen.getByText(/Primary Fitness Goal/i)).toBeInTheDocument();
    expect(screen.getByText(/Experience Level/i)).toBeInTheDocument();
  });
}
```

3.2.6 Redux Testing

Our application uses Redux for state management, and we've implemented tests for key Redux functionality, particularly for the workout generation feature:

```
describe('GeneratedWorkoutDisplay Component', () => {
  it('renders the workout plan data correctly', () => {
    render(<GeneratedWorkoutDisplay workoutPlanData={mockWorkoutPlan} />);

    // Check if main elements are rendered
    expect(screen.getByText('Custom Test Plan')).toBeInTheDocument();
    expect(screen.getByText(/Muscle Gain/i)).toBeInTheDocument();
    expect(screen.getByText(/Intermediate/i)).toBeInTheDocument();
    expect(screen.getByText(/This is a test workout plan/i)).toBeInTheDocument();
  });

  it('handles save button correctly', () => {
    const onSaveMock = jest.fn();
    render(<GeneratedWorkoutDisplay workoutPlanData={mockWorkoutPlan} onSave={onSaveMo

    // Click the save button
    const saveButton = screen.getByTitle('Save Workout');
    fireEvent.click(saveButton);
    // Fill the name and click save
    fireEvent.change(screen.getByLabelText('Workout Plan Name'), {
      target: { value: 'My Saved Workout' }
    });
    fireEvent.click(screen.getByText('Save'));

    // Verify onSave was called with correct params
    expect(onSaveMock).toHaveBeenCalledWith('My Saved Workout', mockWorkoutPlan.worko
  });
}
```

3.3 Testing Results and Analysis

3.3.1 Health Check Results

Our health check endpoint tests consistently pass, indicating that the server is running correctly and responding to basic requests. This establishes a baseline for all other tests.

3.3.2 API Endpoint Results

The API endpoint tests show that our core API functionality is working correctly:

Endpoint	**Status**
Server Health	PASS
API Endpoints	PASS
Workout Generation	PASS

Table 3.1: API Endpoint Test Results

3.3.3 LLM Integration Results

The LLM integration tests demonstrate that our application can successfully connect to the LLM service, generate prompts, and process the responses:

Test	**Status**
LLM Connection	PASS
Prompt Generation	PASS
Response Processing	PASS

Table 3.2: LLM Integration Test Results

3.3.4 Component Test Results

Our React component tests verify that the user interface is rendering correctly and responding to user interactions appropriately:

Component	Rendering	Interaction
WorkoutForm	PASS	PASS
GenerateWorkout	PASS	PASS
GeneratedWorkoutDisplay	PASS	PASS

Table 3.3: Component Test Results

3.4 Test Validation

3.4.1 Test Coverage by Feature

Our testing approach ensures coverage of the most critical aspects of the application:

Feature	Test Coverage
Server Health	Complete
API Endpoints	Complete
LLM Integration	Complete
Workout Generation	Complete
UI Components	Partial
Redux State Management	Partial

Table 3.4: Test Coverage by Feature

3.4.2 Testing Challenges

During our testing implementation, we encountered several challenges:

- **LLM Variability:** The inherent variability of LLM responses made it challenging to create deterministic tests
- **Asynchronous Testing:** Testing asynchronous operations, particularly the LLM integration, required careful timeout handling
- **Redux Testing Complexity:** Testing the Redux state management required mocking complex state interactions

3.4.3 Testing Limitations

Our current testing implementation has some limitations:

- **Focused Coverage:** We prioritized testing critical paths over comprehensive coverage
- **Manual Verification:** Some aspects of the LLM integration require manual verification due to the variable nature of LLM responses
- **Limited E2E Testing:** Full end-to-end testing would require more complex setup with browser automation

3.5 Future Testing Improvements

In future development iterations, we plan to enhance our testing in the following areas:

- **Expanded Unit Testing:** Increase coverage of service and controller functions
- **Comprehensive Integration Testing:** Add more tests for interactions between system components
- **End-to-End Testing:** Implement true E2E tests using tools like Cypress
- **Performance Testing:** Add tests to verify system performance under load
- **Automated Test Pipeline:** Integrate tests into CI/CD pipeline for automatic execution

Chapter 4

Conclusions and Challenges

4.1 Project Summary

The Forward application represents a successful implementation of a modern fitness management system powered by artificial intelligence. Our team successfully delivered a comprehensive solution that addresses the core needs of fitness enthusiasts while leveraging cutting-edge technology to provide personalized workout experiences.

Key accomplishments of the project include:

- **AI-Powered Workout Generation:** We successfully integrated with LLMs to create personalized workout plans based on user preferences, fitness goals, and physical constraints. This feature represents the core innovation of our application and demonstrates the potential for AI to transform fitness planning.
- **Comprehensive Fitness Management:** Beyond workout generation, we implemented a full suite of fitness management features including goal tracking, nutrition logging, and progress visualization, creating a holistic fitness platform.
- **Modern Technical Architecture:** The application follows industry best practices with a cleanly separated frontend and backend, well-defined layers of responsibility, and structured data models. This architecture ensures maintainability and scalability as the application grows.
- **Robust Security Model:** User data protection was implemented through proper authentication, authorization, and data validation, ensuring user information remains secure.
- **Cross-Device Accessibility:** The responsive design ensures users can access their fitness information from any device, supporting continuous fitness tracking regardless of location.

The Forward project demonstrates how modern web technologies and artificial intelligence can converge to create personalized fitness experiences that adapt to individual needs and goals. By combining structured fitness management with AI-generated content, the application offers a unique value proposition that addresses gaps in current fitness applications.

4.2 Challenges Encountered

Throughout the development of Forward, our team faced and overcame several significant challenges:

- **LLM Integration Complexity:** Integrating with LLM services presented unique challenges, particularly in crafting effective prompts that would consistently generate well-structured workout plans. We discovered that prompt engineering is both an art and a science, requiring extensive experimentation to achieve reliable results.
- **Response Parsing and Standardization:** LLM outputs can vary significantly in format and content, making it challenging to parse and standardize responses into structured data for the application. We developed robust parsing mechanisms with fallback strategies to handle variations in response format.
- **Data Model Design:** Creating a flexible yet consistent data model to represent diverse fitness concepts (workouts, exercises, nutrition, goals) required careful consideration. Each domain has its own specific requirements and relationships that needed to be balanced with overall system cohesion.
- **State Management Complexity:** Managing application state across multiple features with interdependencies (e.g., how user profile information affects workout generation) introduced complexity in our Redux implementation. We addressed this through careful slice organization and strategic use of selectors and thunks.
- **Testing LLM-Dependent Functionality:** Creating deterministic tests for features that rely on non-deterministic LLM responses proved challenging. We developed specialized testing approaches that focused on validating the processing logic rather than specific content.
- **Balancing Personalization and Structure:** Finding the right balance between highly personalized content and maintaining a consistent structure for workout plans required iterative refinement of our prompts and processing logic.
- **Performance Optimization:** LLM requests can be time-consuming, potentially affecting user experience. We implemented strategies like request batching, caching, and asynchronous processing to maintain application responsiveness.

These challenges pushed our team to develop creative solutions and deepened our understanding of both fitness domain concepts and modern application development practices. The solutions we developed not only addressed immediate problems but also enhanced the overall architecture and user experience of the Forward application.