

Criterion C

March 25, 2025

Word Count = 1371

Contents

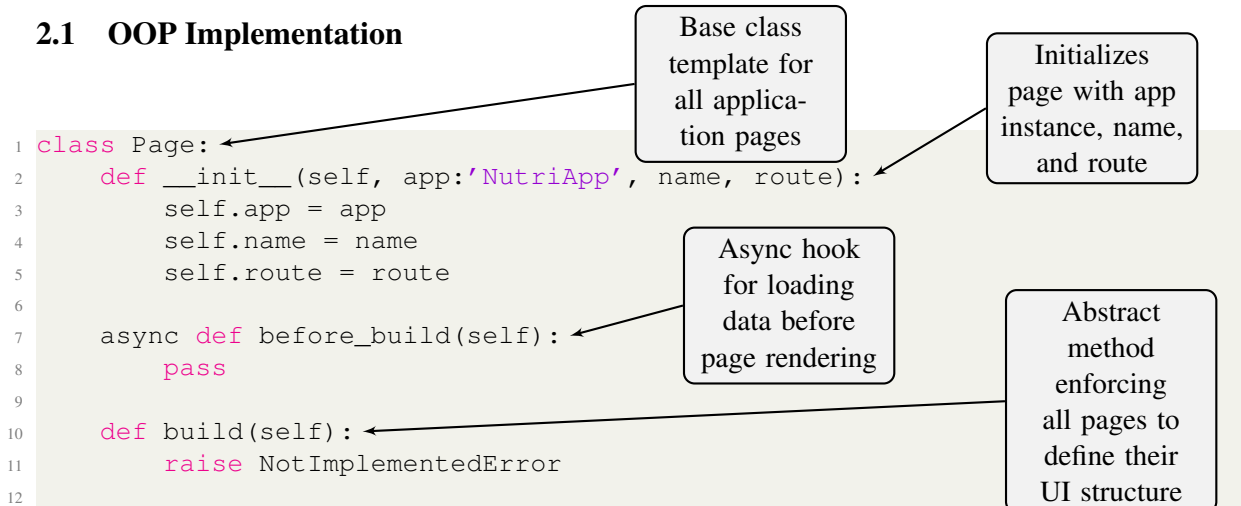
1	Introduction	2
2	Complex Technical Implementations	2
2.1	OOP Implementation	2
2.2	Asynchronous Database Operations	3
2.3	Achievement System Algorithm	4
2.4	Exercise Logging Implementation	5
2.5	Workout Streak Calculation	6
2.6	Dynamic Form Validation	8
2.7	Database Schema Design	9
3	Advanced UI Features	10
3.1	Progress Tracking Visualization	10
3.2	Dynamic Dashboard Implementation	11
4	Security Implementation	12
4.1	Password Hashing	12
5	Tools and Libraries Used	13
5.1	Core Technologies	13
5.2	Security and Data Processing	13
6	References	14

1 Introduction

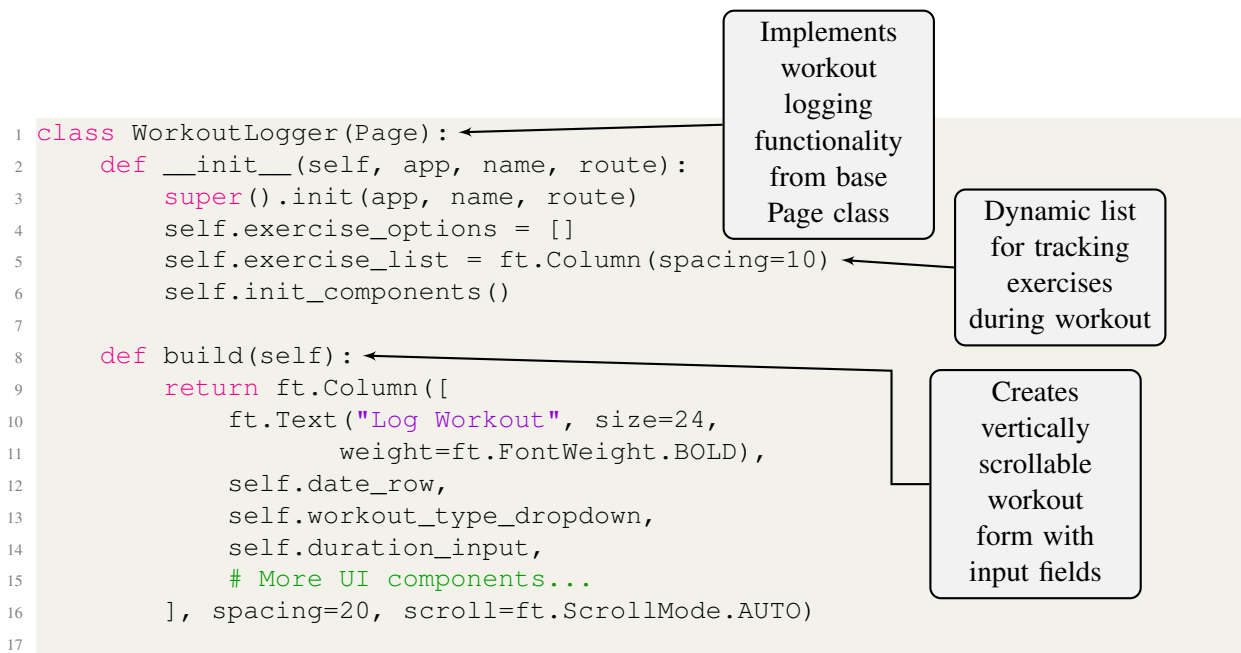
This document outlines the key technical implementations and complex techniques used in developing the NutriSync application. Each section highlights a specific technique, demonstrates its implementation, and justifies its use within the project context.

2 Complex Technical Implementations

2.1 OOP Implementation



Listing 1: Base Page Class Implementation

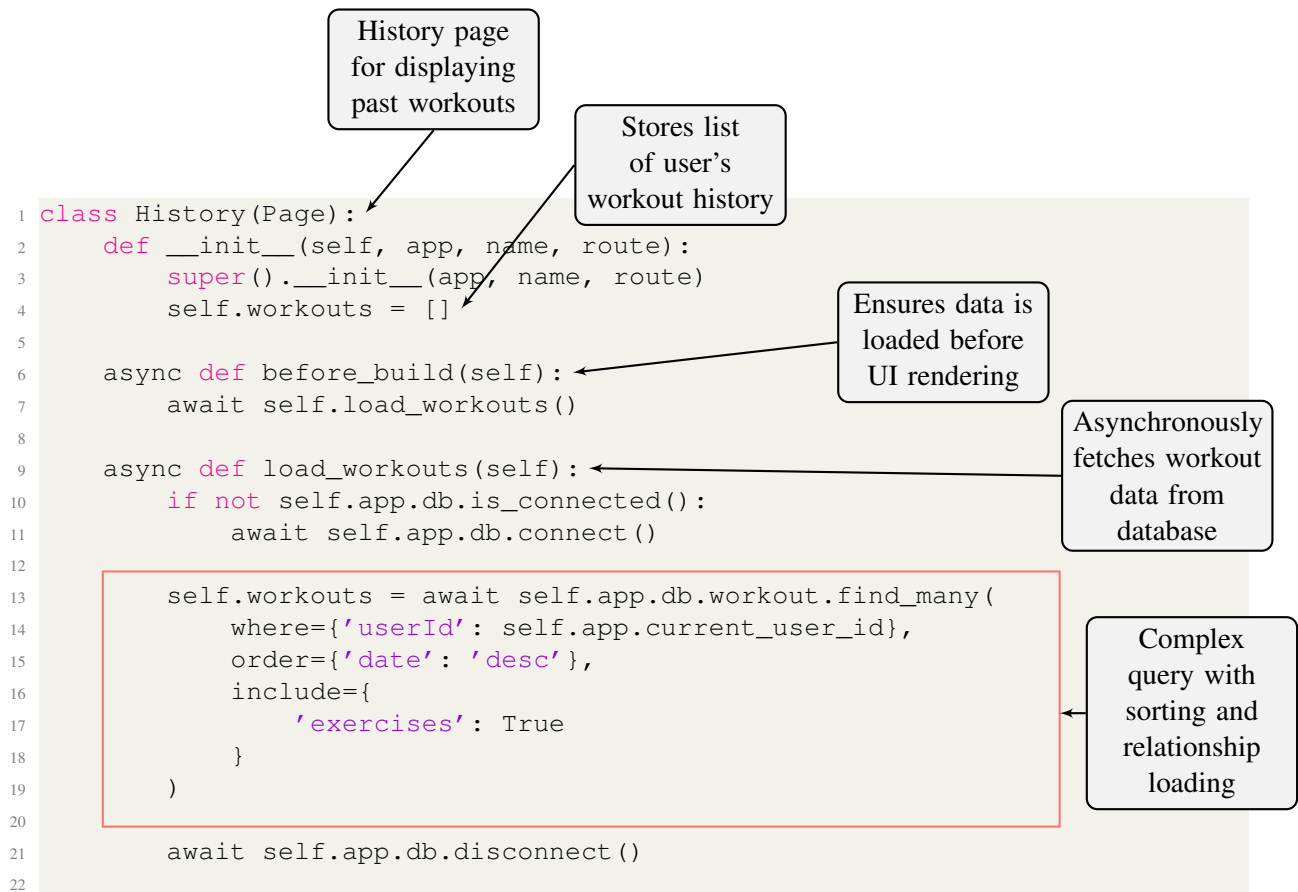


Listing 2: Child Page Implementation

This base class implementation showcases several fundamental Object-Oriented Programming principles that enhance the application's maintainability and scalability. Through encapsulation, the class manages private attributes for the application instance and route handling, ensuring that these critical components remain protected and can only be accessed through appropriate methods. The class employs abstraction by defining an abstract build method, which

establishes a contract requiring all child classes to implement their specific UI structure. This abstraction ensures consistency across the application while allowing flexibility in individual page implementations. Furthermore, the class serves as an interface definition, providing a uniform API that standardizes how pages are initialized, built, and managed throughout the application. This consistent interface simplifies development and maintenance by establishing clear patterns for extending functionality while maintaining architectural integrity.

2.2 Asynchronous Database Operations



Listing 3: Asynchronous Data Loading Example

Asynchronous database operations proves essential to the application's performance and user experience. By leveraging Python's async/await syntax, the application prevents UI freezing during potentially lengthy database queries, ensuring users can continue interacting with the interface while data loads in the background. This asynchronous approach enables concurrent processing of workout data, allowing the application to handle multiple operations simultaneously, such as loading workout history while processing achievement updates. The result is a significantly responsive application that can manage complex database operations without compromising user experience, particularly crucial when dealing with extensive workout histories or complex data relationships. This is especially noticeable during operations that involve multiple database queries or when processing large datasets of workout records.

2.3 Achievement System Algorithm

```
1 async def check_achievements(self):
2     """Check for new achievements after a workout is logged."""
3     achievements = []
4
5     # Get all user workouts
6     workouts = await self.db.workout.find_many(
7         where={'userId': self.user_id},
8         include={'exercises': True}
9     )
10
11    # Check weekly consistency
12    now = datetime.now(pytz.utc)
13    one_week_ago = now - timedelta(days=7)
14    recent_workouts = [w for w in workouts if w.date >= one_week_ago]
15
16    if len(recent_workouts) >= 3:
17        await self._award_achievement(
18            "Weekly Warrior",
19            "Completed 3 or more workouts in a week!",
20            achievements
21        )
22
23    if len(recent_workouts) >= 5:
24        await self._award_achievement(
25            "Five-Star Week",
26            "Completed 5 or more workouts in a week!",
27            achievements
28        )
29
30    # Check workout duration achievements
31    duration_milestones = {
32        60: "Hour Champion",
33        90: "Endurance Explorer",
34        120: "Marathon Trainer"
35    }
36
37    for workout in workouts:
38        for duration, title in duration_milestones.items():
39            if workout.duration >= duration:
40                await self._award_achievement(
41                    title,
42                    f"Completed a {duration}-minute workout!",
43                    achievements
44                )
45
46    return achievements
47
```

Asynchronous
method to
evaluate user
achievements

Retrieves
all workouts
with exercise
details for
achievement
analysis

Evaluates
recent workout
frequency for
streak-based
achievements

Awards
achievements
based on
workout
duration
milestones

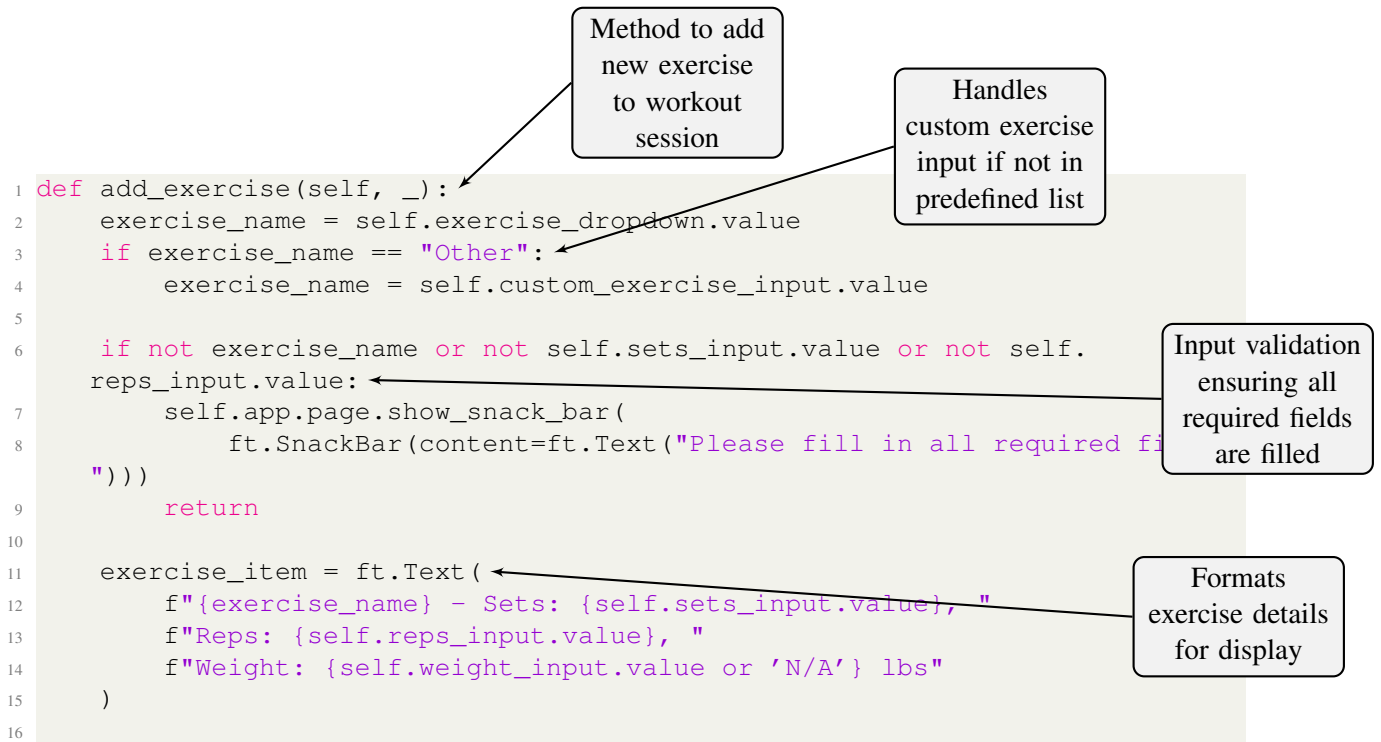
Returns list of
newly earned
achievements

Listing 4: Achievement Tracking

Asynchronous design enables seamless achievement tracking without impacting the user experience, processing multiple achievement conditions concurrently through optimized database queries and in-memory evaluations. The system's architecture ensures immediate user feed-

back through efficient processing of workout data, evaluating multiple achievement criteria in a single pass while maintaining scalability. This implementation particularly shines in its ability to handle various achievement types - from simple workout counts to complex streak-based accomplishments - while maintaining responsive performance through careful consideration of data structure choices and query optimization. The immediate feedback loop created by this system enhances user engagement and motivation, providing instant recognition of user accomplishments without introducing noticeable latency to the application.

2.4 Exercise Logging Implementation



Listing 5: Exercise Logging System

The exercise logging system incorporates several essential features that ensure data integrity and enhance user experience. At its core, the system implements comprehensive input validation that verifies all required fields are properly completed before allowing data submission, preventing incomplete or invalid exercise entries from being stored. The system's dynamic exercise type handling allows users to either select from predefined exercises or input custom exercises, providing flexibility while maintaining data structure consistency. Through real-time UI feedback mechanisms, users receive immediate responses to their actions through snack-bar notifications and dynamic form updates, creating a responsive and intuitive interface. This combination of validation, flexibility, and immediate feedback creates a robust and user-friendly exercise tracking experience.

2.5 Workout Streak Calculation

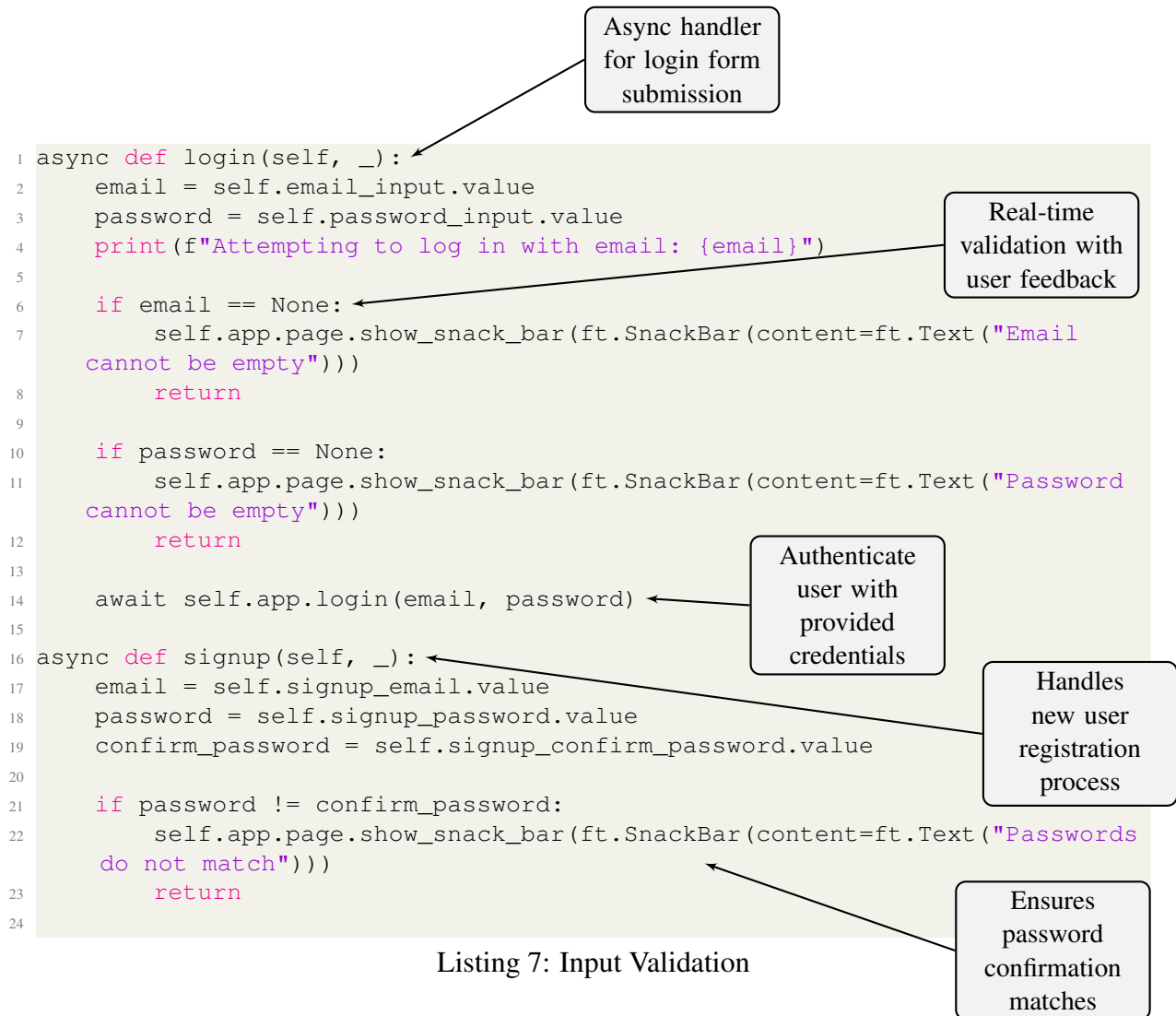


Listing 6: Streak Tracking

Timezone-aware streak calculations proved essential for maintaining the application's data integrity and user experience. By incorporating comprehensive timezone handling, the system ensures accurate streak tracking regardless of users' geographical locations or travel patterns. The algorithm carefully manages edge cases in date calculations, such as daylight savings transitions and midnight crossovers, preventing potential discontinuities in streak tracking. This approach maintains data consistency throughout the application, ensuring that achievements and milestones are awarded correctly and that users' progress is accurately reflected regardless of when or where they log their workouts. The system's careful handling of temporal data sup-

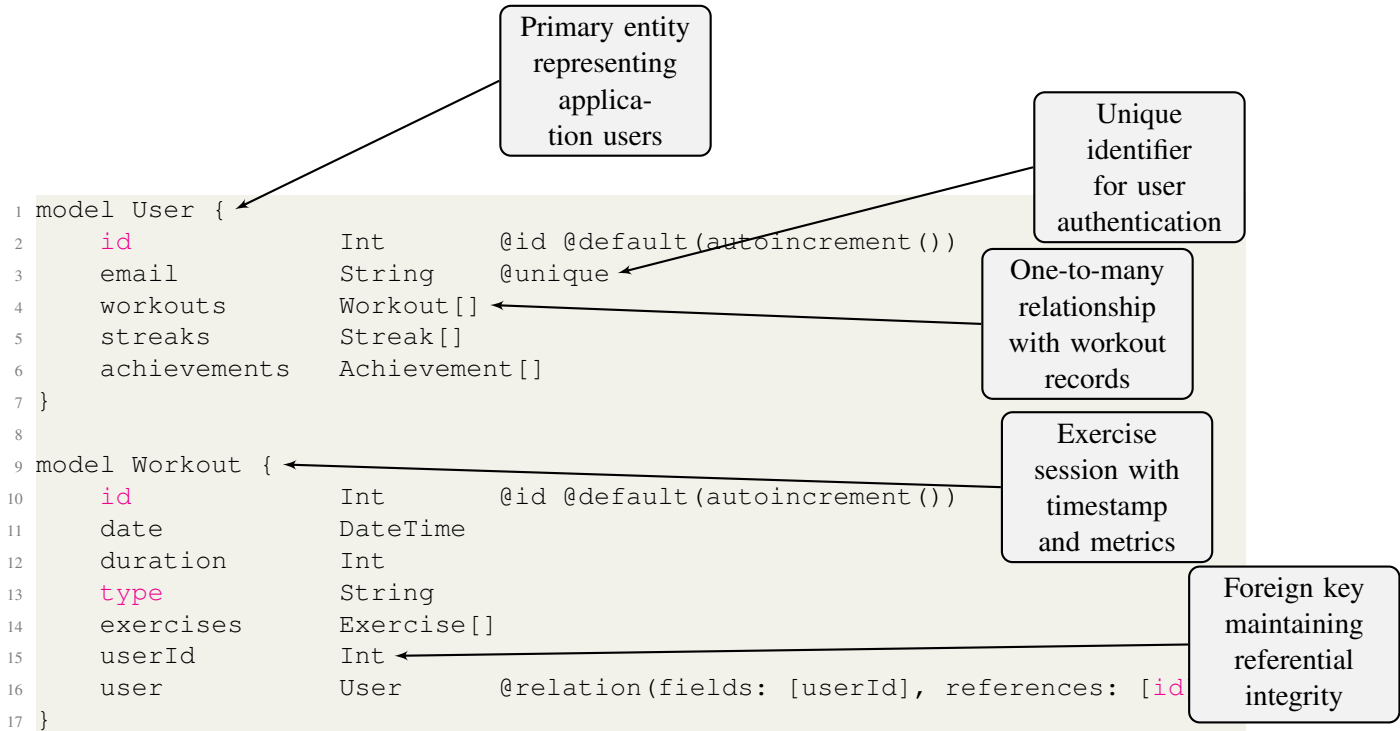
ports the application's core functionality while providing a reliable foundation for features like achievement tracking and progress visualization.

2.6 Dynamic Form Validation



Dynamic form validation through event handlers and state management represents a crucial architectural decision in enhancing the application's user interface. By providing immediate feedback through snackbar notifications, users receive instant validation results as they interact with the form, significantly reducing the likelihood of submission errors and streamlining the user experience. This real-time validation approach serves as a preventive measure against invalid data submission, intercepting potential issues before they reach the backend systems. The combination of asynchronous handlers and state management creates a responsive and intuitive interface that guides users through the authentication process while maintaining data integrity. Furthermore, this validation strategy aligns with modern web application best practices, where immediate user feedback is essential for creating an engaging and user-friendly experience.

2.7 Database Schema Design



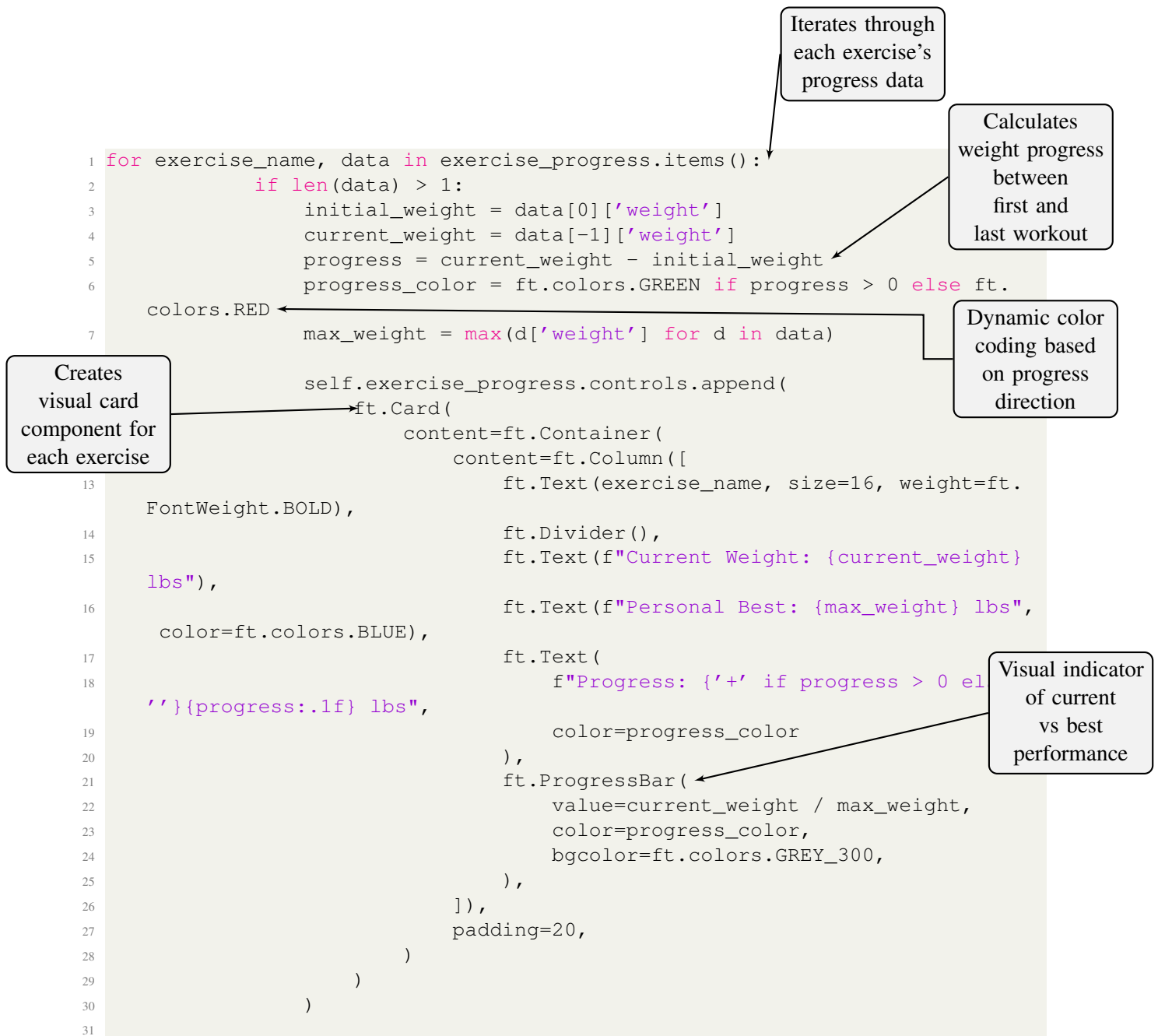
Listing 8: User-Workout Relationship

The schema implementation exemplifies a robust relational database design through its carefully structured one-to-many relationships, which establish clear hierarchical connections between users and their associated data. The Prisma ORM layer provides comprehensive type safety, ensuring that data integrity is maintained at both the application and database levels. Through the ORM's type checking and validation mechanisms, the system prevents data inconsistencies before they reach the database.

The implemented design serves as a cornerstone of the application's data architecture, enabling efficient querying and management of complex user-workout relationships. Through enforced referential integrity via foreign key constraints, the system maintains data consistency even as users accumulate workouts, achievements, and streak records over time. This architectural decision facilitates sophisticated queries necessary for features such as progress tracking and achievement monitoring, while maintaining optimal performance through proper indexing and relationship mapping. The schema's structure also provides the flexibility to accommodate future feature additions while preserving the existing data relationships and integrity constraints.

3 Advanced UI Features

3.1 Progress Tracking Visualization

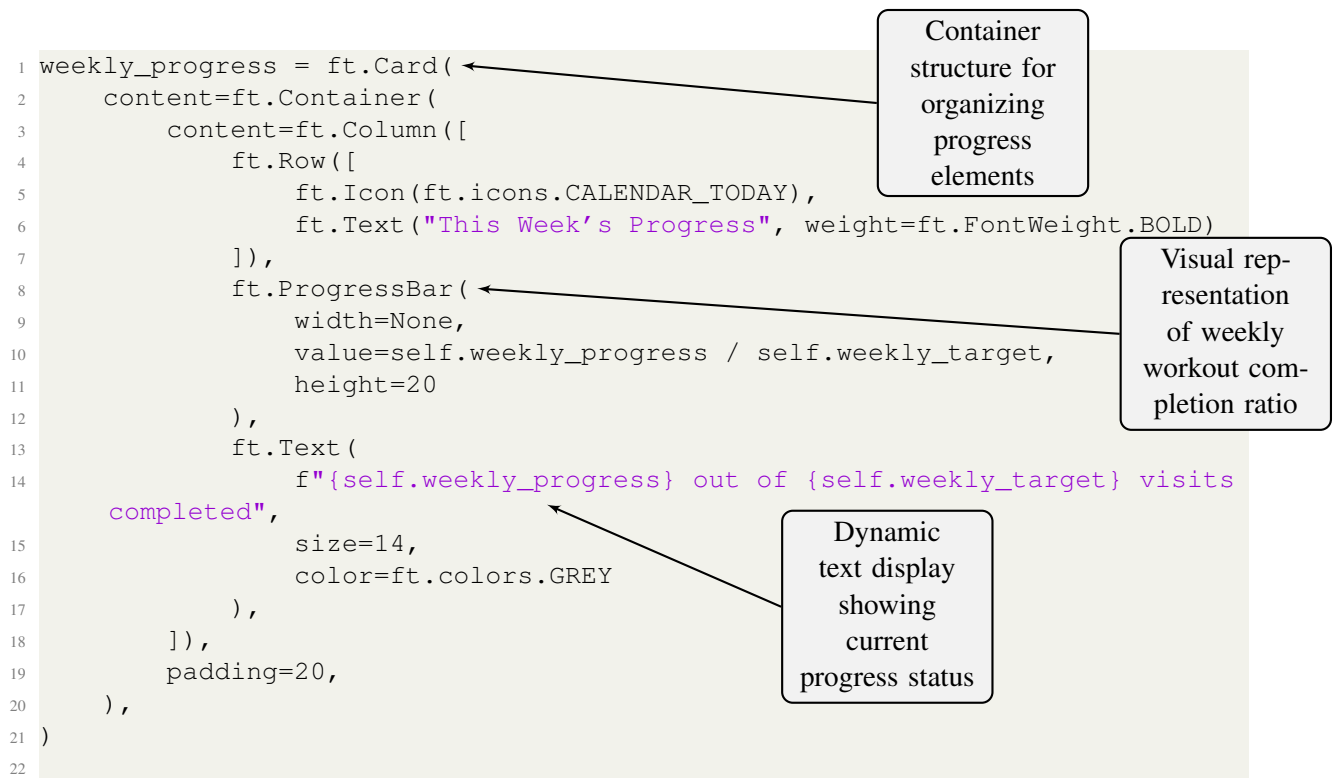


Listing 9: Progress Tracking and Widget

The Progress tracking visualization system proves particularly valuable in enhancing the user experience. By providing clear visual feedback through color-coded progress indicators and dynamic progress bars, users can instantly comprehend their performance trends. The system's ability to handle real-time updates ensures that users receive immediate feedback on their progress. Furthermore, this visual representation serves as a powerful motivational tool, as users can observe their improvements and personal bests, encouraging continued engagement.

The combination of these elements creates a cohesive and intuitive interface that not only displays data but actively contributes to the user's fitness journey by making progress tangible and achievements visible.

3.2 Dynamic Dashboard Implementation



Listing 10: Weekly Progress Card Implementation

Sophisticated state management techniques are used to enable dynamic data updates throughout the dashboard. Through responsive UI components, the system maintains a fluid user experience while handling real-time changes. The progress visualization system represents raw workout data as meaningful visual feedback, helping users track their fitness effectively.

The incorporation of these dynamic elements proves essential to the application's core functionality. By maintaining strict data consistency between the backend and frontend, users receive accurate, up-to-date information about their progress. The system's immediate feedback mechanism enhances the interactive experience, showing users their achievements and progress in real-time. This instantaneous response system significantly improves user engagement by providing clear, visual confirmation of their workout activities and progress toward their fitness goals. The combination of these features creates a responsive, user-centric dashboard that effectively motivates and guides users through their fitness journey.

4 Security Implementation

4.1 Password Hashing

```
1 # Hash the password
2 hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())
3
4 # Add user to the database
5 if not self.app.db.is_connected():
6     await self.app.db.connect()
7 try:
8     new_user = await self.app.db.user.create(
9         data={
10             'email': email,
11             'password': hashed_password.decode('utf-8'), # Store the
12             hashed password as a string
13         }
14     )
15     self.app.page.show_snack_bar(ft.SnackBar(content=ft.Text("Sign up
16     successful! Please log in.)))
17     self.toggle_signup(None) # Switch back to login form
18 except Exception as e:
19     self.app.page.show_snack_bar(ft.SnackBar(content=ft.Text(f"Error during
20     sign up: {str(e)}")))
```

Converts
hashed binary
to string for
database
storage

Provides
immediate
user feedback
on successful
registration

Listing 11: Password hashing using bcrypt (passlib) at user creation

BCrypt password hashing is critical the application's user authentication system. It provides robust protection of user credentials by generating unique salts and applying multiple rounds of cryptographic hashing, aligning with industry-standard practices. The system automatically handles salt generation and hash computation, ensuring that even if the database is compromised, the original passwords remain secure and cannot be reversed from their hashed values. Furthermore, this implementation prevents password exposure during transmission and storage, as only the hashed values persist in the database, maintaining a strong security posture throughout. The use of BCrypt's adaptive key derivation functions also future-proofs the system against increasing computational power by allowing adjustment as needed.

5 Tools and Libraries Used

The development of NutriSync required several specialized tools and libraries, each chosen for specific technical requirements that support the application’s core functionality and performance needs.

5.1 Core Technologies

The application’s foundation rests on three primary technologies, each serving distinct but complementary purposes. Flet serves as the interface implementation, chosen for its Material UI components. This enables seamless cross-platform deployment, while providing reactive UI updates. The framework’s efficient widget tree rendering ensures smooth performance across all supported platforms, making it ideal for our application’s dynamic interface requirements.

For database management, Prisma ORM ensures type safety throughout all operations. Prisma’s ability to auto-generate database queries significantly streamlines development while maintaining security through parameterized queries that prevent SQL injection vulnerabilities. The ORM’s support for complex data relationships and robust migration tools facilitates database schema evolution as the application grows, making it invaluable for maintaining data integrity and scalability.

Our choice of SQLite as the database system was driven by its serverless architecture, which perfectly aligns with our application’s deployment requirements. SQLite provides ACID compliance ensuring data integrity, while enabling efficient local data storage and retrieval without requiring additional configuration. Its built-in support for concurrent access through proper locking mechanisms makes it ideal for handling multiple user interactions while maintaining data consistency.

5.2 Security and Data Processing

The application’s security and data processing capabilities are built upon several specialized libraries chosen for their reliability and performance. BCrypt forms the cornerstone of our security implementation, providing industry-standard password hashing with automatic salt generation. Its deliberately computationally intensive design helps prevent brute-force attacks, while configurable work factors ensure the security measures can be adjusted as computational capabilities evolve.

Time-related functionality is handled through a combination of PyTZ and DateTime libraries, working in concert to ensure accurate and consistent temporal operations. PyTZ manages timezone-aware calculations, ensuring consistent date handling across different regions and automatically handling daylight saving time transitions. This is crucial for maintaining accurate workout streaks and progress tracking across time zones. The DateTime library complements this with robust date and time manipulation capabilities, enabling precise interval calculations for workout tracking and providing ISO-formatted date storage that integrates seamlessly with our database timestamp requirements.

The synergy of these tools and libraries creates a technical foundation that enables secure user authentication, efficient data persistence and retrieval, comprehensive cross-platform compatibility, and precise time-based feature implementation. This technical stack ensures type-safe database operations while delivering a modern, responsive user interface that meets the demanding requirements of our fitness tracking application. The careful selection and integration of these components results in a system that is not only powerful and secure but also maintainable and scalable for future enhancements.

6 References

1. "Flet Documentation." *Flet*, flet.dev/docs/.
2. "Prisma Client Python Reference." *Prisma*, Prisma Labs, www.prisma.io/docs.
3. Shittu, Olumide. "Python Password Hashing and Salting: A Complete Guide." *DEV Community*, dev.to/shittu_olumide/python-hashing-and-salting-4dea.
4. "asyncio — Asynchronous I/O." *Python Documentation*, Python Software Foundation, 2024, docs.python.org/3/library/asyncio.html.