Criterion C

March 25, 2025

Word Count = 1371

Contents

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

```python
class Page:
    def __init__(self, app:'NutriApp', name, route):
        self.app = app
        self.name = name
        self.route = route

    async def before_build(self):
        pass

    def build(self):
        raise NotImplementedError

```

Base class template for all application pages

Initializes page with app instance, name, and route

Async hook for loading data before page rendering

Abstract method enforcing all pages to define their UI structure

Listing 1: Base Page Class Implementation

```python
class WorkoutLogger(Page):
    def __init__(self, app, name, route):
        super().init(app, name, route)
        self.exercise_options = []
        self.exercise_list = ft.Column(spacing=10)
        self.init_components()

    def build(self):
        return ft.Column([
            ft.Text("Log Workout", size=24,
                   weight=ft.FontWeight.BOLD),
            self.date_row,
            self.workout_type_dropdown,
            self.duration_input,
            # More UI components...
        ], spacing=20, scroll=ft.ScrollMode.AUTO)

```

Implements workout logging functionality from base Page class

Dynamic list for tracking exercises during workout

Creates vertically scrollable workout form with input fields

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

History page for displaying past workouts

Stores list of user's workout history

Ensures data is loaded before UI rendering

Asynchronously fetches workout data from database

Complex query with sorting and relationship loading

```python
class History(Page):
    def __init__(self, app, name, route):
        super().__init__(app, name, route)
        self.workouts = []

    async def before_build(self):
        await self.load_workouts()

    async def load_workouts(self):
        if not self.app.db.is_connected():
            await self.app.db.connect()

        self.workouts = await self.app.db.workout.find_many(
            where={'userId': self.app.current_user_id},
            order={'date': 'desc'},
            include={
                'exercises': True
            }
        )

        await self.app.db.disconnect()

```

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

```python
async def check_achievements(self):
        """Check for new achievements after a workout is logged."""
        achievements = []

        # Get all user workouts
        workouts = await self.db.workout.find_many(
            where={'userId': self.user_id},
            include={'exercises': True}
        )

        # Check weekly consistency
        now = datetime.now(pytz.utc)
        one_week_ago = now - timedelta(days=7)
        recent_workouts = [w for w in workouts if w.date >= one_week_ago]

        if len(recent_workouts) >= 3:
            await self._award_achievement(
                "Weekly Warrior",
                "Completed 3 or more workouts in a week!",
                achievements
            )

        if len(recent_workouts) >= 5:
            await self._award_achievement(
                "Five-Star Week",
                "Completed 5 or more workouts in a week!",
                achievements
            )

        # Check workout duration achievements
        duration_milestones = {
            60: "Hour Champion",
            90: "Endurance Explorer",
            120: "Marathon Trainer"
        }

        for workout in workouts:
            for duration, title in duration_milestones.items():
                if workout.duration >= duration:
                    await self._award_achievement(
                        title,
                        f"Completed a {duration}-minute workout!",
                        achievements
                    )

        return achievements
```

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 feedback 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

Method to add new exercise to workout session

Handles custom exercise input if not in predefined list

```python
def add_exercise(self, _):
    exercise_name = self.exercise_dropdown.value
    if exercise_name == "Other":
        exercise_name = self.custom_exercise_input.value

    if not exercise_name or not self.sets_input.value or not self.reps_input.value:
        self.app.page.show_snack_bar(
            ft.SnackBar(content=ft.Text("Please fill in all required fi")))
        return

    exercise_item = ft.Text(
        f"{exercise_name} - Sets: {self.sets_input.value}, "
        f"Reps: {self.reps_input.value}, "
        f"Weight: {self.weight_input.value or 'N/A'} lbs"
    )

```

Input validation ensuring all required fields are filled

Formats exercise details for display

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 snackbar 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

```
# Get current streak
streak = await self.db.streak.find_first(
    where={'userId': self.user_id},
    order={'endDate': 'desc'}
)

# Check total workouts achievements
total_workouts = len(workouts)
workout_milestones = {
    1: "First Workout",
    5: "Getting Started",
    10: "Dedicated Athlete",
    25: "Fitness Enthusiast",
    50: "Workout Warrior",
    100: "Centurion",
    200: "Double Centurion",
    365: "Year-Round Athlete"
}

for count, title in workout_milestones.items():
    if total_workouts >= count:
        await self._award_achievement(
            title,
            f"Completed {count} workouts!",
            achievements
        )

# Check streak achievements
if streak:
    streak_milestones = {
        3: "Three-Day Streak",
        7: "Week Warrior",
        14: "Two-Week Terror",
        30: "Monthly Master",
        60: "Consistency King",
        90: "Quarterly Champion",
        180: "Half-Year Hero",
        365: "Year of Dedication"
    }

```

Retrieves user's current workout streak from database

Defines achievement levels based on total workouts completed

Awards achievements when workout count hits milestones

Defines achievements for maintaining workout streaks

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

2.6 Dynamic Form Validation

Async handler for login form submission

```python
async def login(self, _):
    email = self.email_input.value
    password = self.password_input.value
    print(f"Attempting to log in with email: {email}")

    if email == None:
        self.app.page.show_snack_bar(ft.SnackBar(content=ft.Text("Email cannot be empty")))
        return

    if password == None:
        self.app.page.show_snack_bar(ft.SnackBar(content=ft.Text("Password cannot be empty")))
        return

    await self.app.login(email, password)

async def signup(self, _):
    email = self.signup_email.value
    password = self.signup_password.value
    confirm_password = self.signup_confirm_password.value

    if password != confirm_password:
        self.app.page.show_snack_bar(ft.SnackBar(content=ft.Text("Passwords do not match")))
        return

```

Real-time validation with user feedback

Authenticate user with provided credentials

Handles new user registration process

Ensures password confirmation matches

Listing 7: Input 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

```
model User {
    id             Int       @id @default(autoincrement())
    email          String    @unique
    workouts       Workout[]
    streaks        Streak[]
    achievements   Achievement[]
}

model Workout {
    id             Int       @id @default(autoincrement())
    date           DateTime
    duration       Int
    type           String
    exercises      Exercise[]
    userId         Int
    user           User      @relation(fields: [userId], references: [id
}
```

Primary entity representing application users

Unique identifier for user authentication

One-to-many relationship with workout records

Exercise session with timestamp and metrics

Foreign key maintaining referential integrity

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

```
for exercise_name, data in exercise_progress.items():
            if len(data) > 1:
                initial_weight = data[0][’weight’]
                current_weight = data[-1][’weight’]
                progress = current_weight - initial_weight
                progress_color = ft.colors.GREEN if progress > 0 else ft.colors.RED
                max_weight = max(d[’weight’] for d in data)

                self.exercise_progress.controls.append(
                    ft.Card(
                        content=ft.Container(
                            content=ft.Column([
                                ft.Text(exercise_name, size=16, weight=ft.FontWeight.BOLD),
                                ft.Divider(),
                                ft.Text(f"Current Weight: {current_weight} lbs"),
                                ft.Text(f"Personal Best: {max_weight} lbs", color=ft.colors.BLUE),
                                ft.Text(
                                    f"Progress: {’+’ if progress > 0 el
’’}{progress:.1f} lbs",
                                    color=progress_color
                                ),
                                ft.ProgressBar(
                                    value=current_weight / max_weight,
                                    color=progress_color,
                                    bgcolor=ft.colors.GREY_300,
                                ),
                            ]),
                            padding=20,
                        )
                    )
                )

```

Iterates through each exercise’s progress data

Calculates weight progress between first and last workout

Dynamic color coding based on progress direction

Creates visual card component for each exercise

Visual indicator of current vs best performance

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

```
weekly_progress = ft.Card(
    content=ft.Container(
        content=ft.Column([
            ft.Row([
                ft.Icon(ft.icons.CALENDAR_TODAY),
                ft.Text("This Week's Progress", weight=ft.FontWeight.BOLD)
            ]),
            ft.ProgressBar(
                width=None,
                value=self.weekly_progress / self.weekly_target,
                height=20
            ),
            ft.Text(
                f"{self.weekly_progress} out of {self.weekly_target} visits completed",
                size=14,
                color=ft.colors.GREY
            ),
        ]),
        padding=20,
    ),
)

```

Container structure for organizing progress elements

Visual representation of weekly workout completion ratio

Dynamic text display showing current progress status

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

```python
# Hash the password
hashed_password = bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())

# Add user to the database
if not self.app.db.is_connected():
    await self.app.db.connect()
try:
    new_user = await self.app.db.user.create(
        data={
            'email': email,
            'password': hashed_password.decode('utf-8'),  # Store the hashed password as a string
        }
    )
    self.app.page.show_snack_bar(ft.SnackBar(content=ft.Text("Sign up successful! Please log in.")))
    self.toggle_signup(None)  # Switch back to login form
except Exception as e:
    self.app.page.show_snack_bar(ft.SnackBar(content=ft.Text(f"Error during sign up: {str(e)}")))

```

Converts hashed binary to string for database storage

Provides immediate user feedback on successful registration

Listing 11: Pasword 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.