# CSE5011 - Mobile Application Development

Contents

[CSE5011 - Mobile Application Development 1](#_Toc209001628)

[TASK A 1](#_Toc209001629)

[Mobile Operating Systems Comparison 1](#_Toc209001630)

[iOS Platform: The Walled Garden 1](#_Toc209001631)

[Strengths 1](#_Toc209001632)

[Limitations 3](#_Toc209001633)

[Android Platform: The Open Ecosystem 4](#_Toc209001634)

[Strengths 4](#_Toc209001635)

[Limitations 5](#_Toc209001636)

[Cross-Platform Considerations: The Middle Ground 6](#_Toc209001637)

[Progressive Web App (PWA) Approach 6](#_Toc209001638)

[Hybrid Approach (React Native, Flutter…) 7](#_Toc209001639)

[Development Tools Analysis 7](#_Toc209001640)

[The Dominance of Android Studio 7](#_Toc209001641)

[Build Tools and Dependency Management with Gradle 8](#_Toc209001642)

[Programming Languages Evaluation 9](#_Toc209001643)

[Java: The Established Foundation 9](#_Toc209001644)

[Kotlin: The Modern Successor 9](#_Toc209001645)

[Groovy: The Build Script Language 10](#_Toc209001646)

[Database Technology Assessment 10](#_Toc209001647)

[SQLite: The Default Embedded Database 10](#_Toc209001648)

[Performance Considerations 10](#_Toc209001649)

[Memory Management: Beyond Automatic Garbage Collection 11](#_Toc209001650)

[Database Performance: Optimizing Data Access 11](#_Toc209001651)

[Security Assessment 12](#_Toc209001652)

[Data Security: Protecting Data at Rest 12](#_Toc209001653)

[Mitigation Strategy: 12](#_Toc209001654)

[Platform Security: Leveraging Android's Built-in Defenses 13](#_Toc209001655)

[TASK B 14](#_Toc209001656)

[Entity-Relationship (ER) Diagram 14](#_Toc209001657)

[Database Design 15](#_Toc209001658)

[TASK C 17](#_Toc209001659)

[TASK E 37](#_Toc209001660)

# TASK A

## Mobile Operating Systems Comparison

## iOS Platform: The Walled Garden

Apple's iOS ecosystem is often described as a "walled garden"—a highly controlled environment that prioritizes security, uniformity, and a premium user experience.

### Strengths

* **Uniform User Experience:**
  + Apple exercises strict control over both its hardware (iPhone, iPad) and software (iOS). This vertical integration means developers target a limited, predictable set of device specifications and screen sizes.
  + This uniformity drastically simplifies the design, development, and quality assurance processes. A feature that works on a latest iPhone model will almost certainly work identically on a previous model, albeit with potential performance differences. This consistency builds strong user trust, as customers know exactly what to expect from an app regardless of which Apple device they use. It eliminates the frustration of apps behaving differently on various phones from the same brand.
* **Security:**
  + iOS is renowned for its robust security model, built on two pillars: the App Store review process and sandboxing.
  + Every app submitted to the App Store undergoes a rigorous, manual and automated review by Apple to check for compliance with its guidelines, including privacy violations, use of private APIs, and malicious code. This acts as a powerful gatekeeper. Furthermore, on the device itself, each app runs in its own "sandbox," a restricted environment where it cannot access data or resources from other apps without explicit user permission. This combination makes iOS highly resistant to malware and data breaches, a critical factor for applications handling sensitive user data (e.g., banking, travel, healthcare).
* **Monetization Potential:**
  + Statistical data consistently shows that iOS users generate higher average revenue per user (ARPU) than their Android counterparts, particularly in sectors like travel, luxury goods, and subscription services.
  + This trend is attributed to the demographic profile of iPhone users, who tend to have higher disposable incomes and a greater willingness to pay for premium apps, in-app purchases, and services. For a business focused on direct revenue generation, the iOS user base represents a highly valuable target market. An investment in developing for iOS can often yield a faster and higher return on investment (ROI) through sales.
* **Hardware Integration:**
  + iOS provides seamless and standardized access to Apple's proprietary hardware features.
  + Features like **Apple Pay** (for secure, one-tap payments), **Face ID/Touch ID** (for biometric authentication), and the **ARKit** framework (for augmented reality) are deeply integrated into the OS. For developers, this means they can implement these cutting-edge features easily and reliably, knowing they will work perfectly on all supported devices. This deep integration allows for the creation of sophisticated and secure user experiences that are difficult to replicate uniformly on the fragmented Android landscape.
* **Update Adoption:**
  + When Apple releases a new version of iOS, a very high percentage of users install the update within a few months.
  + Because Apple controls the update process and pushes it directly to devices (bypassing carriers and manufacturers), fragmentation is minimal. As a developer, this means you can confidently utilize the latest APIs and features introduced in a new iOS version shortly after its release, knowing that the vast majority of your user base will be able to access them. This accelerates innovation within apps and reduces the need to support very old operating system versions.

### Limitations

* **Market Share Restriction:**
  + By developing solely for iOS, an application is automatically unavailable to the ~73% of the global smartphone market that uses Android.
  + This is a significant strategic limitation. If your goal is maximum market penetration, brand awareness, or serving a global audience where Android devices dominate (especially in emerging markets), an iOS-only strategy will miss a substantial portion of potential users. The decision to prioritize iOS must be a conscious trade-off between audience quality and audience quantity.
* **Development Restrictions:**
  + Apple's App Store guidelines are strict and can be subjective. The company enforces specific rules regarding design, functionality, content, and business models.
  + This can lead to frustration for developers. Apps can be rejected for not meeting UI/UX standards, for using non-Apple payment systems for digital goods, or for content deemed inappropriate. This limits creativity and control, potentially preventing developers from implementing features that are possible on other platforms. The review process, while beneficial for security, also adds time to app updates and releases.
* **Cost Considerations:**
  + The barrier to entry for iOS development is financially higher.
  + To develop for iOS, a company or individual must invest in Apple hardware (a Mac computer is mandatory for using Xcode, the development environment). Furthermore, to publish an app on the App Store, an annual **Apple Developer Program** fee of $99 is required. While these costs may be negligible for a established company, they can be a significant hurdle for indie developers or startups.

## Android Platform: The Open Ecosystem

Google's Android platform champions openness and customization, leading to massive global adoption but introducing complexity.

### Strengths

* **Market Dominance:**
  + Android's overwhelming global market share is its single greatest advantage.
  + Reaching approximately 72% of all smartphone users, an Android application has the potential for unparalleled reach. This is crucial for applications aiming for mass-market adoption, such as communication tools, social networks, or utility apps. In many regions outside of North America and Western Europe, Android's market share is even higher, making it the default choice for global strategies.
* **Customization Flexibility:**
  + Android offers developers far greater freedom to customize the user interface and system interactions.
  + Unlike iOS, Android allows apps to become deeply integrated into the system. They can set default handlers for actions (e.g., "always open links in Chrome"), create custom home screen widgets, and alter system settings in ways iOS does not permit. This flexibility enables more innovative and powerful app designs that can tailor the phone's functionality to the user's specific needs.
* **Hardware Diversity:**
  + Android runs on thousands of devices from numerous manufacturers (Samsung, Google, Xiaomi, etc.) across all price points.
  + This diversity allows your application to reach users who cannot afford premium Apple devices. You can target everyone with a high-end foldable phone and a user with a budget-friendly device. For businesses, this also means they can develop custom apps for a wide range of enterprise-grade devices, barcode scanners, and other specialized hardware that often runs Android.
* **Development Freedom:**
  + The Android ecosystem is less restrictive. While Google Play is the primary store, users can "sideload" apps from other sources.
  + This openness allows for alternative app stores (e.g., the Samsung Galaxy Store, Amazon Appstore) and direct distribution of apps (e.g., for enterprise or internal use). The app review process for Google Play is generally faster and more automated than Apple's, though it has become stricter over time. Developers also have more freedom in implementing payment systems and app functionality.
* **Google Services Integration:**
  + Android offers native, first-party integration with the suite of Google services.
  + For apps that rely on maps, email, cloud storage, or authentication, this is a major benefit. Integrating **Google Maps** is seamless and feature-rich. **Google Drive** and **Firebase** (Google's backend platform) are deeply embedded, making it easier to build apps that sync data across devices. This provides a powerful toolkit for developers out of the box.

### Limitations

* **Fragmentation Challenges:**
  + This is Android's most significant technical challenge. Fragmentation occurs across two axes: **OS versions** and **device hardware**.
  + Unlike iOS, Android updates are controlled by device manufacturers and mobile carriers. This results in a wide mix of active OS versions (from Android 10 to the latest), meaning developers must often write code to support older APIs. Furthermore, the immense variety of devices means a vast range of screen sizes, resolutions, processor speeds, RAM amounts, and even hardware features (e.g., some devices lack a gyroscope). This makes comprehensive testing incredibly complex and expensive, as an app must be tested on dozens of device profiles to ensure consistent performance and avoid crashes.
* **Security Concerns:**
  + The open nature of Android makes it a more frequent target for malware.
  + The ability to install apps from unknown sources (sideloading) is a primary vector for malware distribution. While Google Play Protect scans apps on Google Play, it is not infallible. The responsibility for security shifts more towards the developer; they must be vigilant in implementing security best practices to protect user data, as the system-level protections, while robust, are not as uniformly enforced as on iOS.
* **Fragmented User Experience:**
  + The user experience of an Android app can vary significantly from device to device.
  + Different manufacturers (Samsung, OnePlus, Xiaomi) apply their own custom skins (One UI, OxygenOS, MIUI) on top of standard Android. These skins alter system fonts, colors, animations, and even overall navigation. This can cause an app's carefully designed interface to look and behave slightly differently on a Samsung phone versus a Google Pixel, leading to an inconsistent user experience.
* **Monetization Challenges:**
  + While the Android user base is larger, its users are generally less willing to pay upfront for apps and may spend less on in-app purchases.
  + The dominant monetization model on Android is often advertising-supported (freemium) rather than paid apps or direct purchases. This means an app may need a much larger user base on Android to generate the same revenue as a smaller, but more spending, user base on iOS. Converting users to paying customers requires different strategies.

## Cross-Platform Considerations: The Middle Ground

Cross-platform frameworks aim to balance the reach of Android with the development efficiency of a single codebase.

### Progressive Web App (PWA) Approach

* **Advantages:**
  + PWAs are websites that use modern web capabilities to deliver an app-like experience directly through a web browser.
  + Their biggest advantage is the **elimination of installation**. Users access the app by visiting a URL, removing the friction of downloading from an app store. They are inherently cross-platform (work on any device with a standards-compliant browser) and updates are instantaneous—the user always has the latest version the moment they reload the page. Development and maintenance costs are significantly lower as only one codebase is needed.
* **Limitations:**
  + The major trade-off is limited access to native device features and performance.
  + PWAs have restricted or no access to key device APIs like Bluetooth, NFC, advanced camera controls, or sensors like the gyroscope. They cannot be distributed through the Apple App Store or Google Play, limiting discoverability. While performance has improved dramatically, they still cannot match the smoothness and speed of a truly native application, especially for graphics-intensive tasks (e.g., complex games).

### Hybrid Approach (React Native, Flutter…)

* **Balance:**
  + Frameworks like React Native (Facebook) and Flutter (Google) allow developers to write code once in a single language (JavaScript/Dart) and compile it into native apps for both iOS and Android.
  + This approach can reduce development time and cost by **50-80%** compared to building two separate native apps. The resulting apps are not web views; they use native components, providing performance and a user experience that is very close to that of a truly native app. This is the most popular choice for startups and businesses that need to reach both platforms with a limited budget without sacrificing too much quality.

## Development Tools Analysis

The selection of development tools is a critical factor that directly impacts developer productivity, application performance, and the long-term maintainability of a project. The Android ecosystem, while open, has largely standardized around a primary toolset provided by Google.

### The Dominance of Android Studio

Android Studio stands as the unequivocal official Integrated Development Environment (IDE) for Android development. Its status as a first-party tool from Google guarantees that it is the first to receive support for new Android APIs, features, and best practices. This direct support is its most significant advantage, ensuring developers are always equipped with the most relevant and powerful tools for the platform.

The IDE is distinguished by its deep integration of Android-specific features. It includes a highly optimized emulator that can mimic a vast array of real device profiles, allowing developers to test their applications on different virtual devices without needing physical hardware for each one. Its advanced profiling tools are indispensable for performance tuning, providing real-time data on CPU usage, memory allocation, network activity, and energy consumption. This allows developers to identify and fix performance bottlenecks before they impact the end-user experience.

However, this power comes at a cost. Android Studio is a resource-intensive application that demands a powerful development machine with substantial RAM (16GB is highly recommended, 8GB is a bare minimum) and a fast solid-state drive (SSD) to run effectively. This hardware requirement can represent a significant investment for developers or organizations.

While alternative IDEs exist, they are generally not practical for professional Android development. IntelliJ IDEA, the platform upon which Android Studio is built, lacks the Android-specific plugins and tools by default. Visual Studio Code, while lightweight and popular for web development, requires extensive configuration and a suite of plugins to approach the functionality of Android Studio, making it an inefficient choice. Eclipse with the Android Development Tools (ADT) plugin is now completely deprecated and obsolete, serving only as a historical footnote.

### Build Tools and Dependency Management with Gradle

The build process for modern Android applications is almost exclusively handled by Gradle. This build system was selected by Google for its exceptional flexibility and power. Gradle allows development teams to create highly customized build configurations, enabling them to define different versions of an app (e.g., a free version with ads and a paid version without) from a single codebase, a process known as "build variants."

A key strength of Gradle is its focus on performance. It utilizes techniques like incremental builds (only recompiling what has changed) and build caching to significantly reduce compilation times, which is a major boost to developer productivity during the development and testing cycles.

Furthermore, Gradle integrates a robust dependency management system. It seamlessly handles the downloading and integration of external libraries from repositories like Maven Central, simplifying the process of adding complex functionality to an app without having to write everything from scratch. The primary challenge with Gradle is its learning curve; its configuration scripts can be complex and require dedicated expertise to master and troubleshoot. Older systems like Maven and Ant are largely irrelevant in contemporary Android development due to their lack of flexibility and inferior performance.

## Programming Languages Evaluation

The choice of programming language defines the developer experience, application safety, and long-term viability of the codebase. The Android platform has undergone a significant language evolution.

### Java: The Established Foundation

Java served as the original and primary language for Android for many years. Its key advantage lies in its mature ecosystem. There is a vast universe of well-documented libraries, extensive community knowledge, and a massive global pool of experienced developers familiar with the language. This makes finding talent and solving problems relatively easy. As a compiled language, it offers strong runtime performance and has a long history of backward compatibility, ensuring stability.

However, Java shows its age compared to modern languages. It is notoriously verbose, requiring a large amount of "boilerplate" code (repetitive code structures) to accomplish common tasks, which can slow down development and make code harder to read. A more critical drawback is its lack of built-in null safety. This means variables can unintentionally hold a null value, leading to frequent and often catastrophic NullPointerException crashes at runtime, which are a major source of bugs in Android applications.

### Kotlin: The Modern Successor

Kotlin has emerged as Google's preferred and officially supported language for Android development, and for compelling reasons. It is designed to fully interoperate with Java but addresses many of its shortcomings. The most immediate benefit is its conciseness; Kotlin can achieve the same functionality as Java with significantly less code, reducing boilerplate and improving code readability and maintainability.

Its most celebrated feature is built-in null safety. The Kotlin type system explicitly distinguishes between nullable and non-nullable references, forcing the developer to handle potential null cases at compile time. This proactively eliminates an entire class of runtime crashes, leading to more stable and reliable applications. Kotlin also introduces modern features that are highly beneficial for mobile development, such as coroutines for simplified and efficient asynchronous programming and extension functions for adding functionality to existing classes without inheritance.

For teams familiar with Java, the transition to Kotlin is relatively smooth due to its interoperability. A Kotlin application can leverage all existing Java libraries and even mix Kotlin and Java code within the same project, allowing for a gradual migration. The performance overhead of using Kotlin is negligible, making it a strictly superior choice for new Android projects.

### Groovy: The Build Script Language

It is important to distinguish Groovy from application languages. Groovy is primarily used within the Gradle build scripts (build.gradle files) that configure the build process. Its advantages here are its scripting capabilities and seamless interoperability with Java code, making it ideal for automating build tasks. However, it is not typically used for writing the application logic itself, as its interpreted nature is less performant than compiled Java or Kotlin.

## Database Technology Assessment

The choice of a local database is crucial for providing a fast, responsive, and reliable user experience, especially when the device is offline.

### SQLite: The Default Embedded Database

SQLite was selected as the foundational database technology for Android due to its compelling advantages for a mobile environment. Its most significant feature is that it is built directly into the Android operating system, requiring no additional setup or dependencies from the developer. It is an incredibly lightweight engine, consuming minimal CPU, memory, and battery resources, which are all critical constraints on mobile devices.

As a well-tested and ACID-compliant relational database, SQLite is highly reliable and ensures data integrity. Its primary strength is enabling full offline functionality; applications can store and retrieve data locally without any network connection, syncing with a remote server when a connection is re-established.

## Performance Considerations

Application performance is a cornerstone of user satisfaction. A slow, unresponsive, or resource-heavy application leads to frustration and abandonment. The current architecture presents several opportunities for significant optimization.

### Memory Management: Beyond Automatic Garbage Collection

While the Android Runtime (ART) employs an advanced garbage collector to automatically free unused memory, relying solely on this mechanism is a reactive and often insufficient strategy. The most common and pernicious performance issue in Android applications is the **memory leak**.

A memory leak occurs when an object is no longer needed but is still held in memory because it is unintentionally referenced by another object. In Android, the most frequent cause is holding a reference to an Activity or Context beyond its lifecycle. For example, a long-running background task holding onto an Activity context prevents that entire activity—and all its views and resources—from being garbage-collected after it is closed. Repeated leaks, especially of large objects, cause the application to consume increasing amounts of memory, eventually leading to sluggish performance, janky UI, and ultimately, an OutOfMemoryError crash.

The current manual management of database Cursors is another potential pitfall. If a Cursor is not explicitly closed after use, it leaks the underlying database resources and memory.

**Optimization Strategy:**  
A proactive approach to memory management is required. This involves adopting modern Android Jetpack components, specifically **ViewModel** and **Lifecycle-Aware Components**. The ViewModel class is designed to store and manage UI-related data in a lifecycle-conscious way. It survives configuration changes (like screen rotations) so data doesn't need to be reloaded, and it is automatically cleared when the associated Activity is permanently destroyed, preventing common context-related leaks.

Furthermore, integrating a tool like **LeakCanary** into the development and QA process is invaluable. This library automatically detects memory leaks in debug builds, providing a clear trace that pinpoints the exact source of the leak, allowing developers to fix issues before they reach production. Finally, all resources like Cursors, file streams, and network sockets must be explicitly closed in a finally block or by using the try-with-resources statement to ensure cleanup happens reliably.

### Database Performance: Optimizing Data Access

SQLite is highly performant, but its efficiency is entirely dependent on how it is used. Without optimization, database operations can become a major bottleneck, causing the UI thread to block and making the application feel unresponsive.

The single most important optimization for read operations is the use of **database indexing**. An index acts like a book's index, allowing the database to find data without performing a full table scan. Without an index on a frequently queried column (e.g., user\_id or product\_name), every query must read every row in the table, a process that becomes exponentially slower as the database grows.

For write operations, the most critical practice is the use of **batched transactions**. By wrapping multiple INSERT, UPDATE, or DELETE operations into a single transaction, the application instructs the database to perform them all at once. Without this, each operation is treated as its own transaction, forcing the database to write to disk repeatedly, which is a very slow process. Proper transaction management can improve write speed by orders of magnitude.

Finally, all database queries must be crafted carefully. Using SELECT \* should be avoided in favor of explicitly naming only the required columns (SELECT id, name). Complex JOIN operations should be analyzed to ensure they are necessary and efficient.

## Security Assessment

In mobile development, security cannot be an afterthought. It is a fundamental requirement to protect user data and maintain trust. The current implementation has several critical vulnerabilities that must be addressed before release.

### Data Security: Protecting Data at Rest

The most severe vulnerability in the current state is the storage of passwords or sensitive data in plain text or with basic encryption. If an attacker gains physical access to a rooted device or extracts the data via a backup, all user credentials are immediately compromised. This is a catastrophic failure of data protection.

Mitigation Strategy:  
Sensitive data must never be stored directly in an SQLite database without robust encryption. The recommended solutions are:

1. **Android Jetpack Security (EncryptedSharedPreferences):** For smaller datasets like API tokens, user preferences, or passwords, this library provides a simple and secure way to encrypt key-value pairs using a master key that is itself stored in the device's secure hardware (the Trusted Execution Environment - TEE), if available.
2. **SQLCipher:** For entire databases that require encryption, SQLCipher provides transparent, full-database encryption. It is an open-source extension to SQLite that encrypts every page of the database file, rendering it useless without the passphrase.

Furthermore, all data entering the system must be treated as untrusted. **Input validation** is crucial to prevent attacks like SQL Injection. While using parameterized queries (e.g., with SQLiteDatabase.query() or Room's DAOs) instead of concatenating raw SQL strings inherently mitigates this risk, validation ensures data conforms to expected formats.

For any future network communication, security is paramount. All communication must occur over **HTTPS** (TLS) to prevent eavesdropping and man-in-the-middle attacks. For highly sensitive applications, **Certificate Pinning** should be implemented to ensure the app only communicates with the intended server, even if a device has a malicious certificate installed. Authentication should be handled via modern, token-based systems like **OAuth 2.0**, avoiding the transmission and storage of actual passwords.

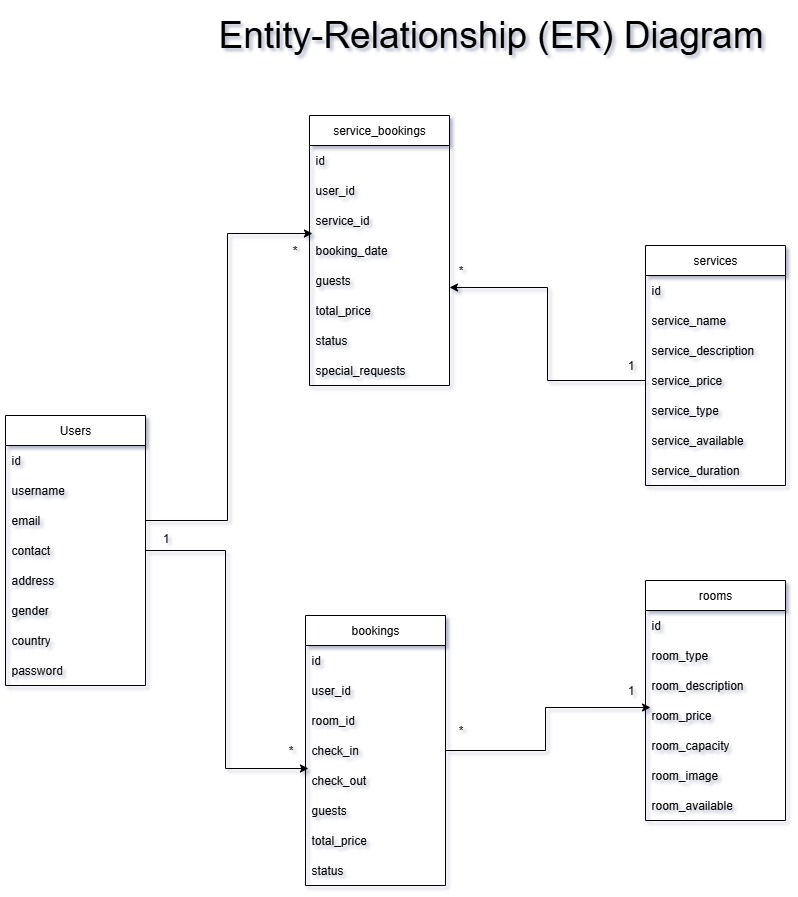
### Platform Security: Leveraging Android's Built-in Defenses

Android provides a robust security model that applications must consciously leverage. The **permission system** is a key feature. Instead of requesting all permissions at install time, the application should request sensitive permissions (e.g., location, camera, contacts) at runtime, precisely when they are needed. This follows the principle of least privilege and allows the user to understand why a permission is necessary, building trust.

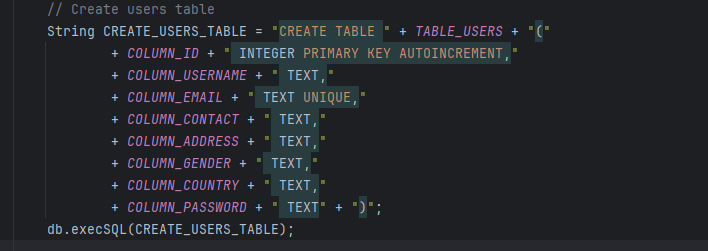
The Android **application sandbox** is a fundamental security feature that isolates each application's data and code execution from every other application. This means that even if a vulnerability is found in this application, it is contained and cannot easily compromise other apps or the system itself. Adhering to Android's best practices ensures this isolation is maintained.

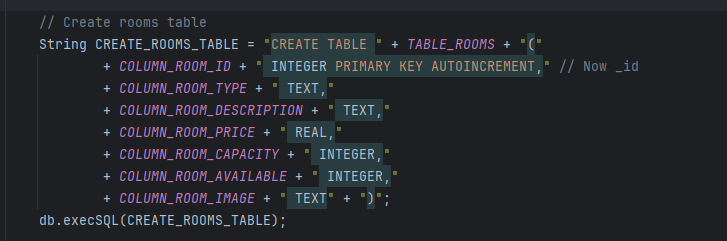
# TASK B

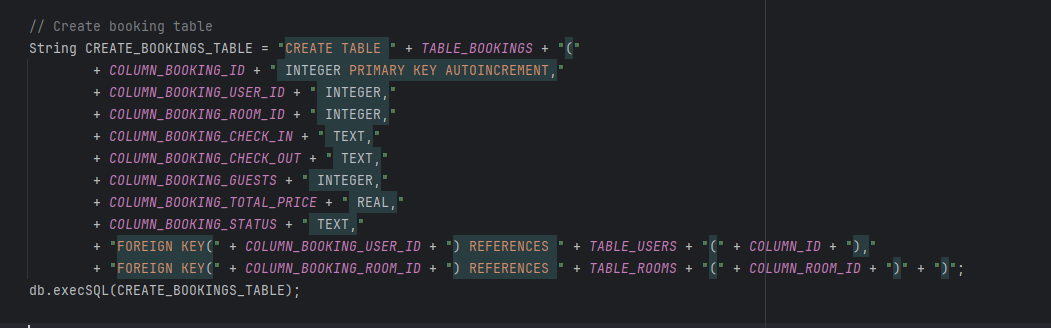
## Entity-Relationship (ER) Diagram

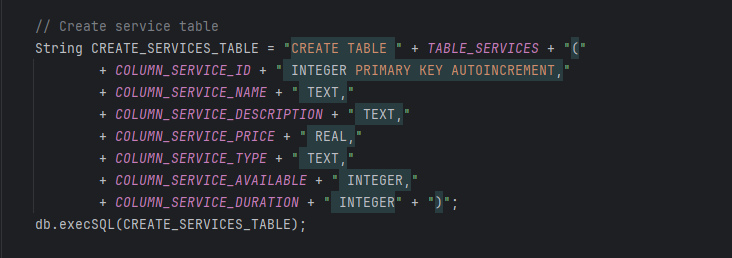
****

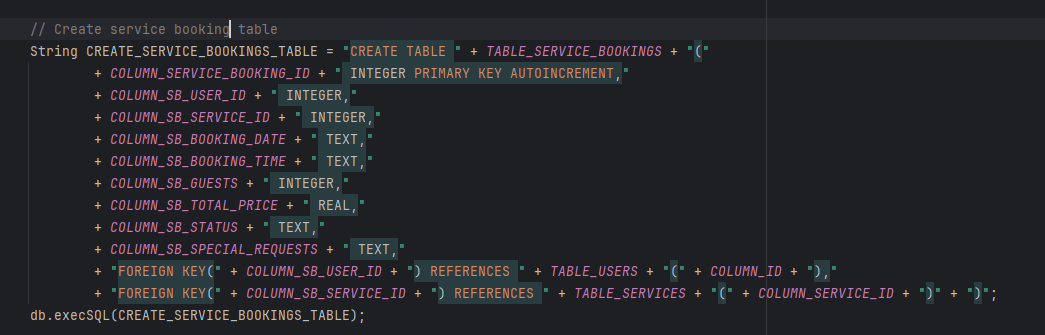
## Database Design



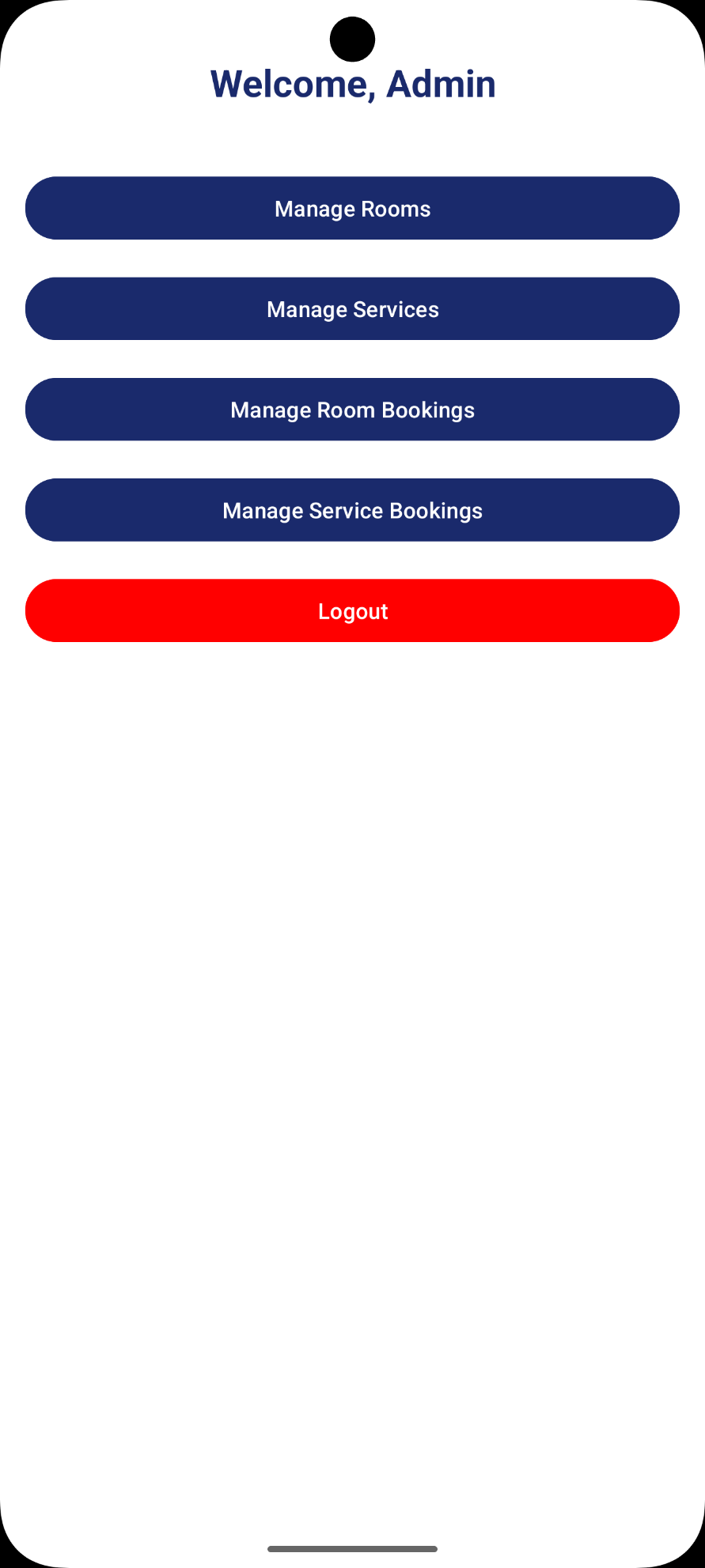
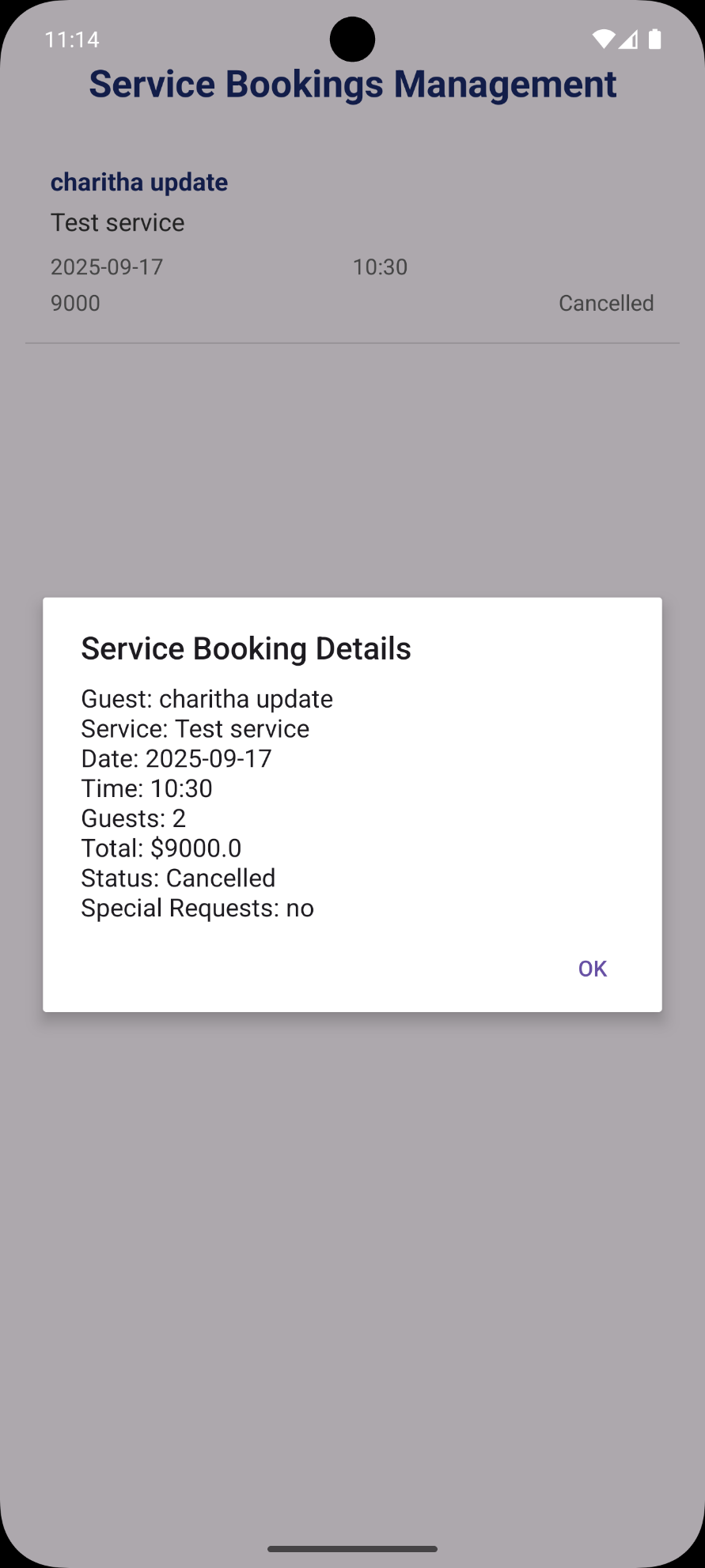
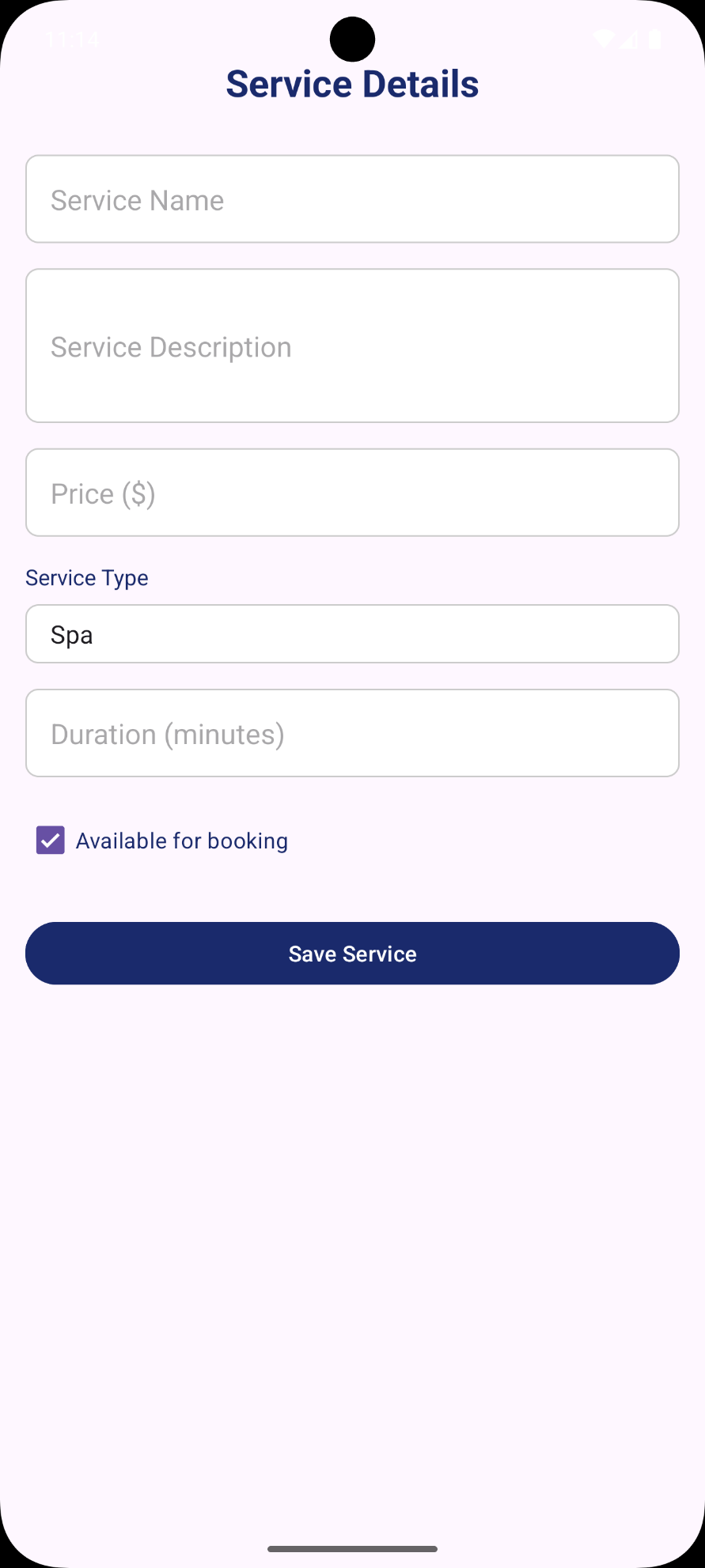
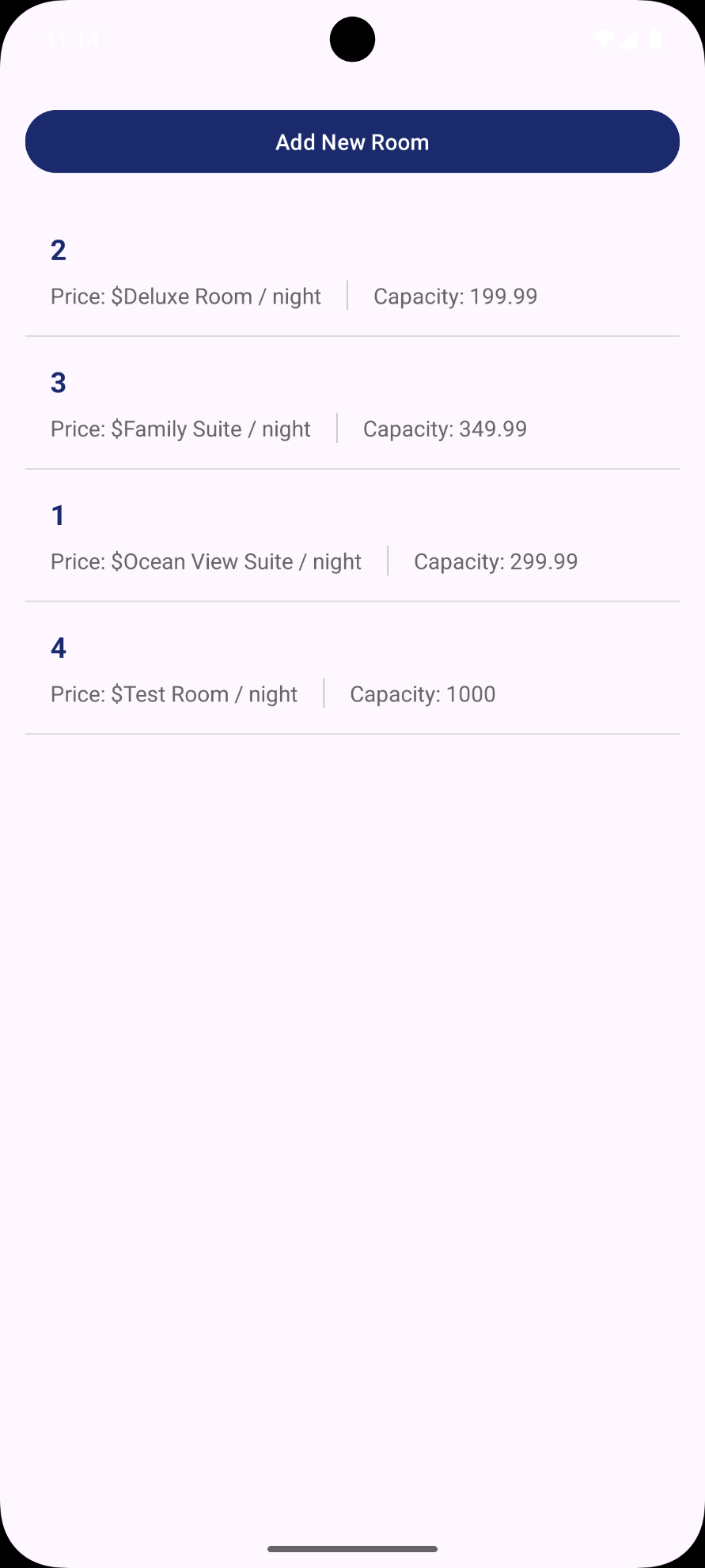
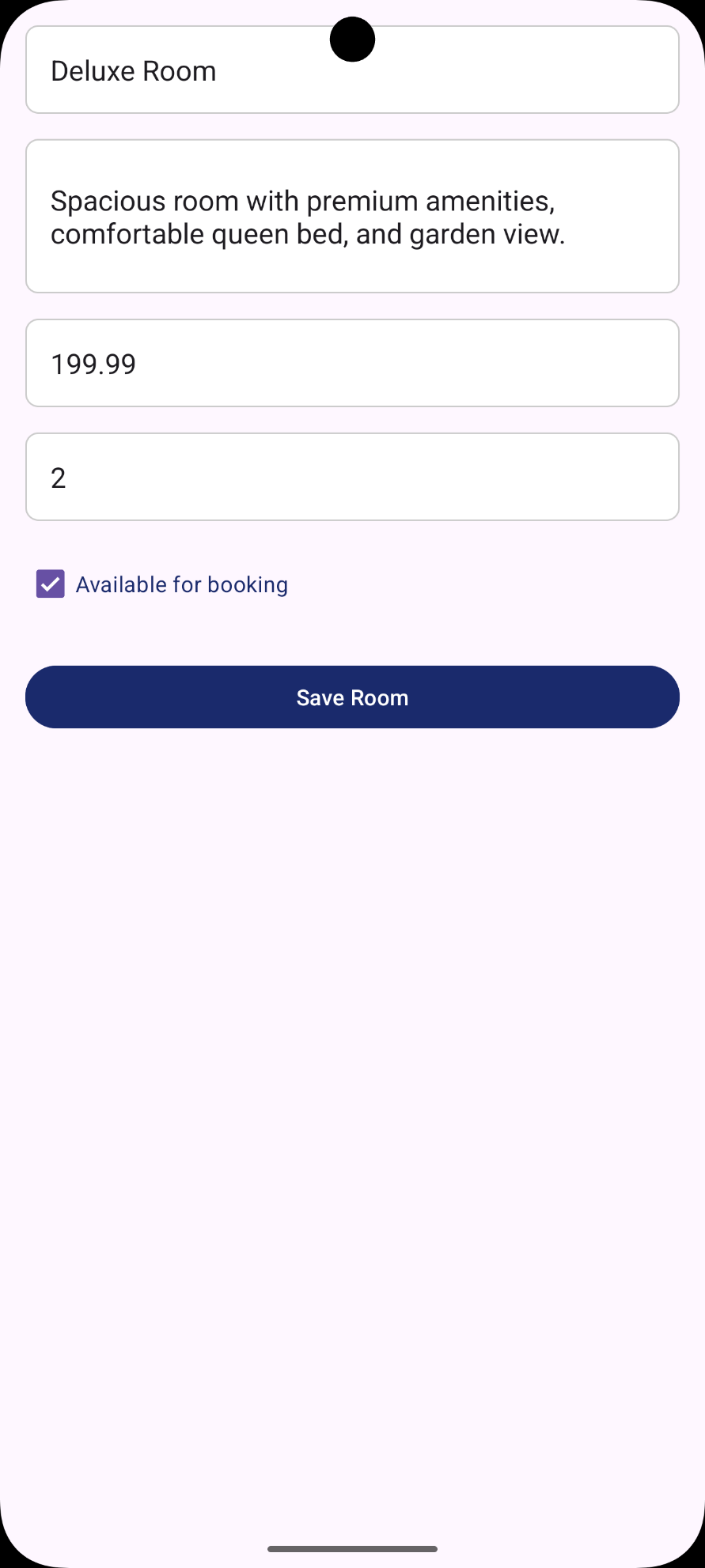
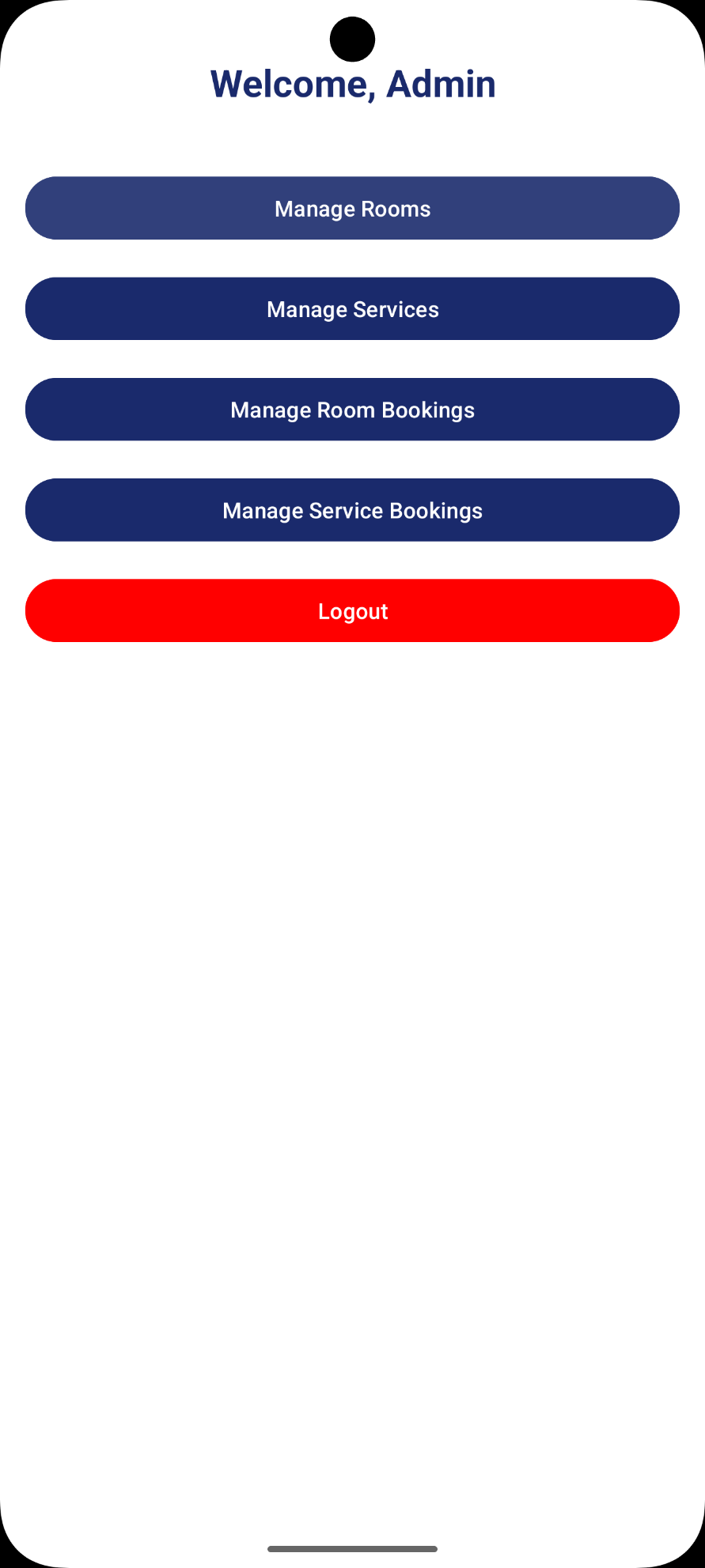
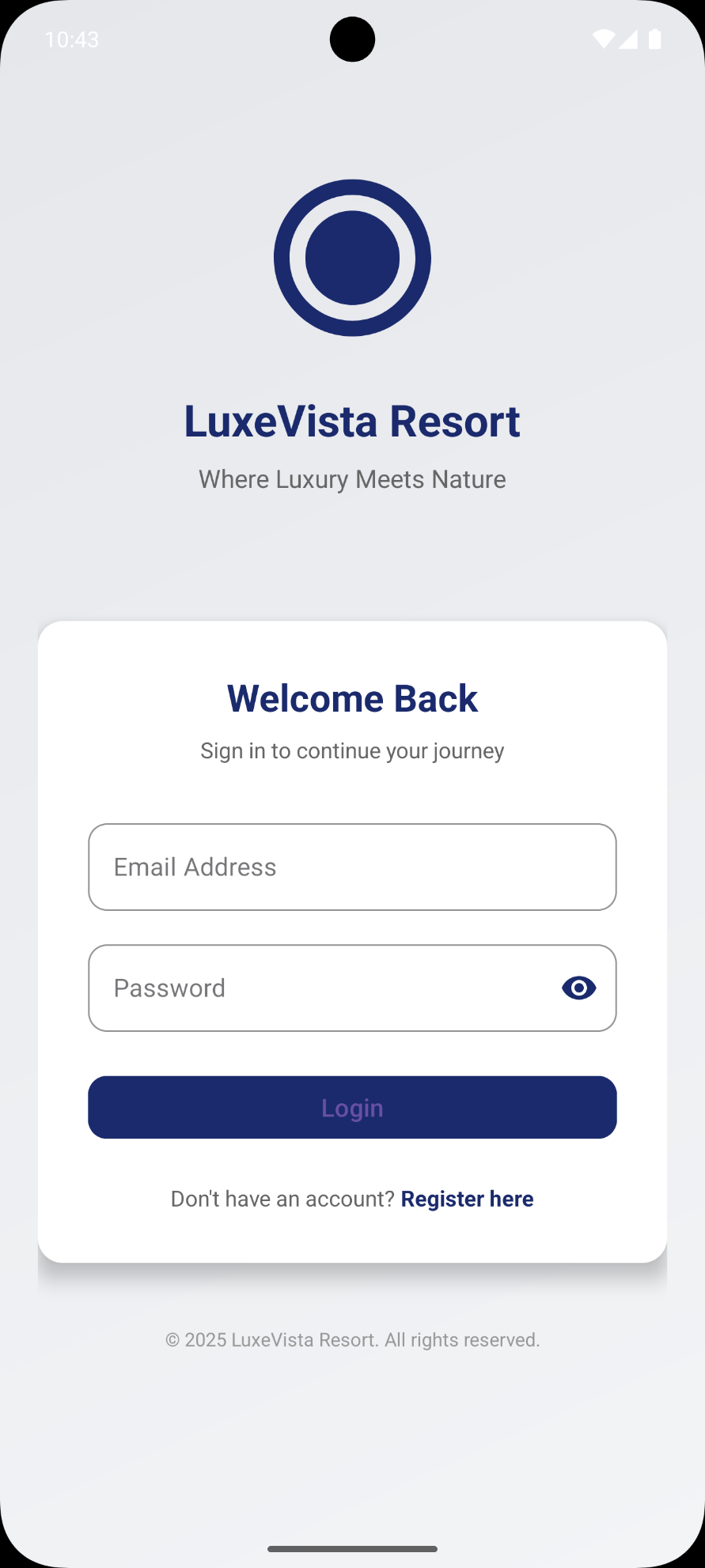
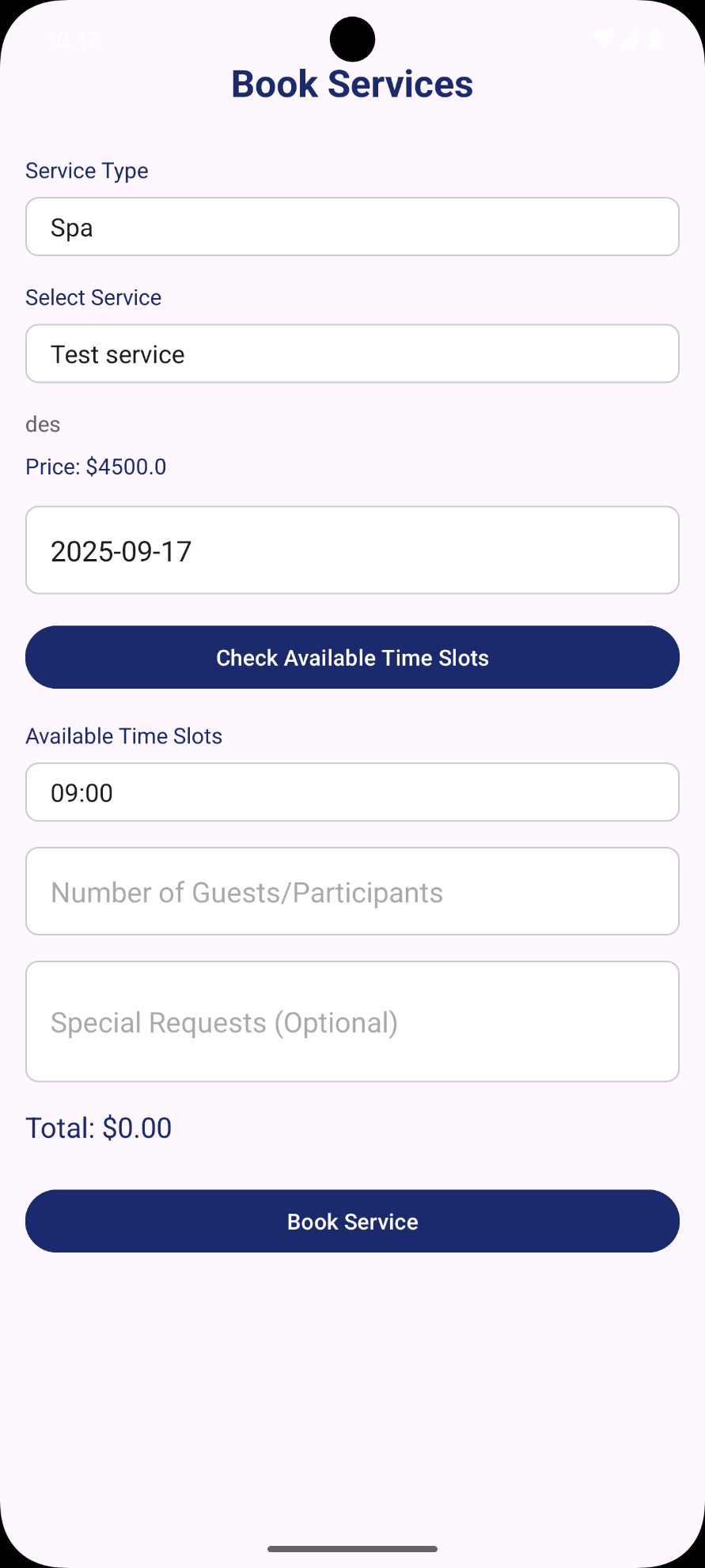
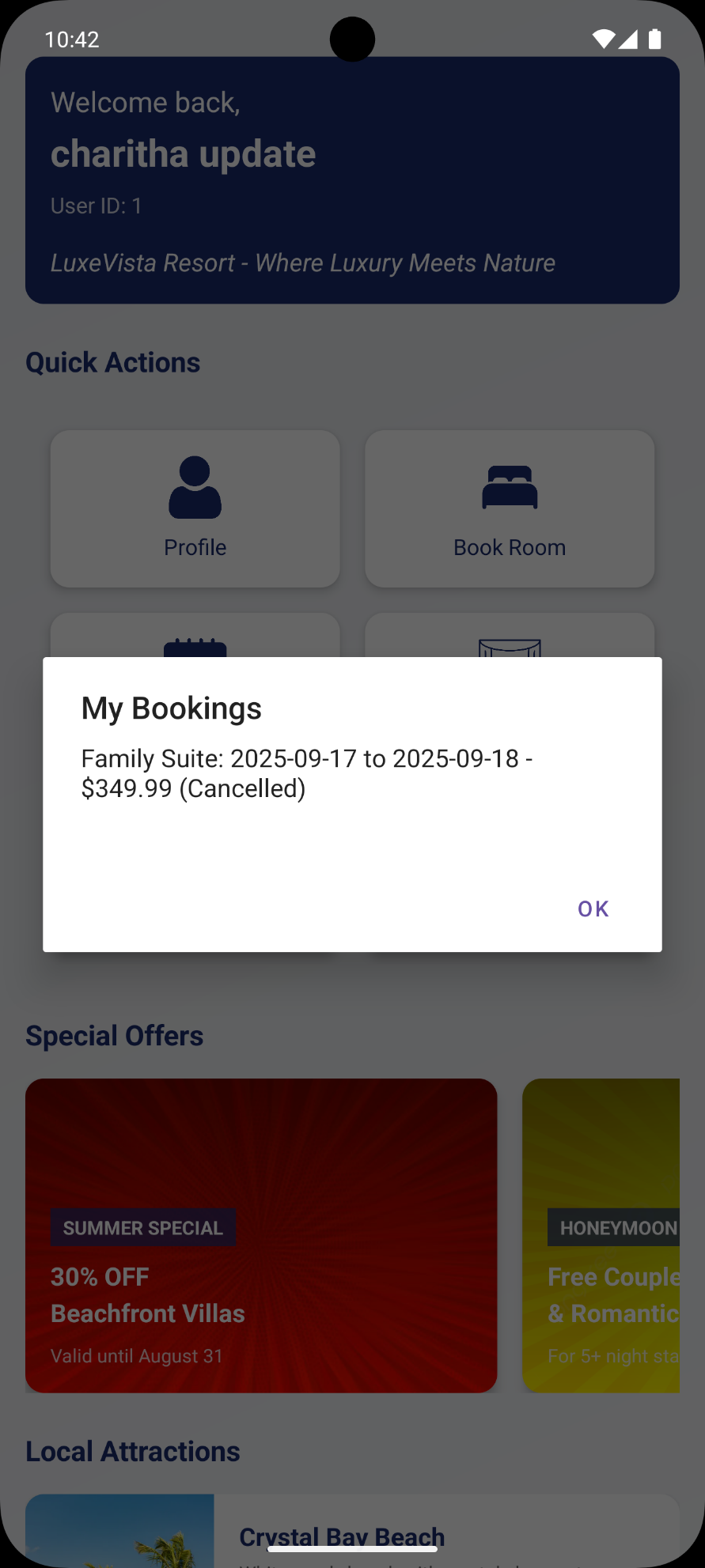
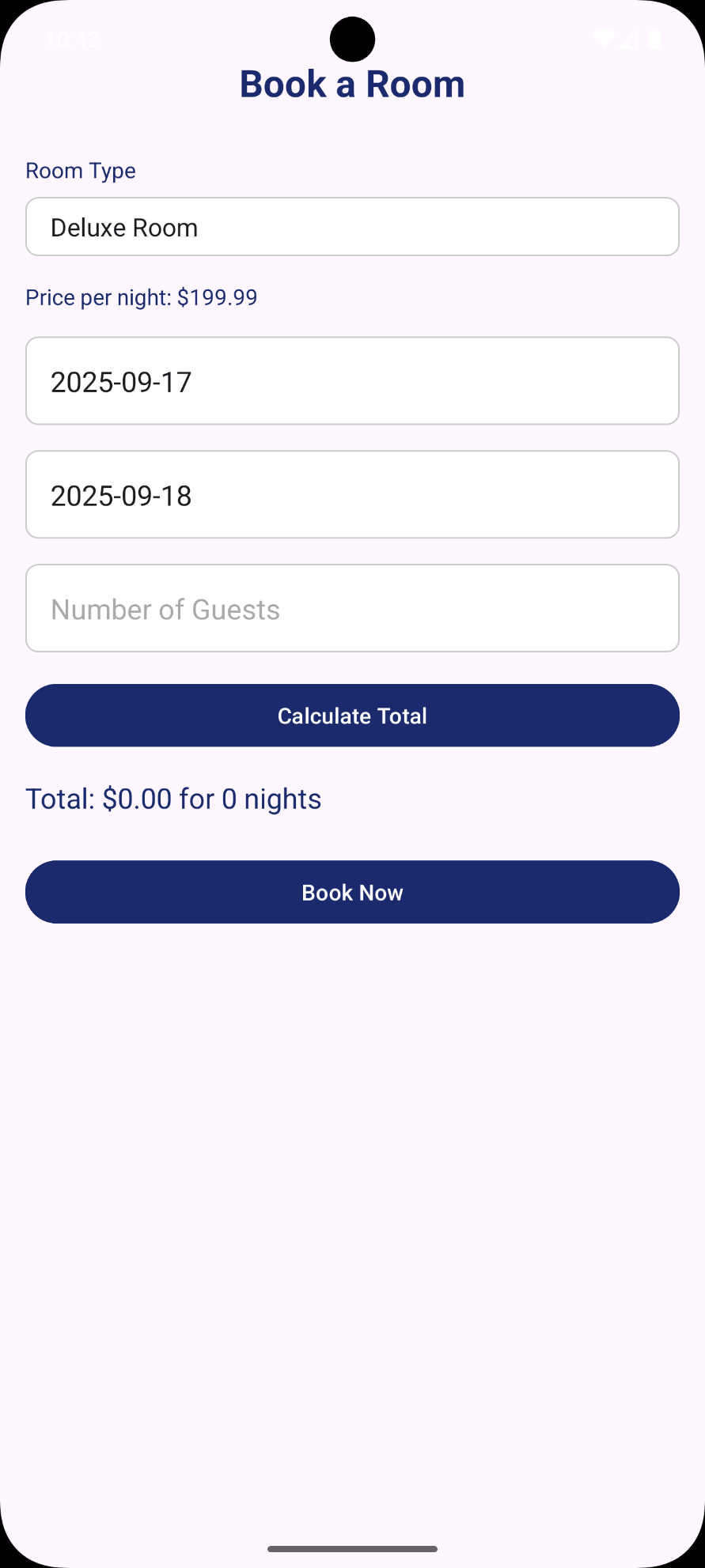
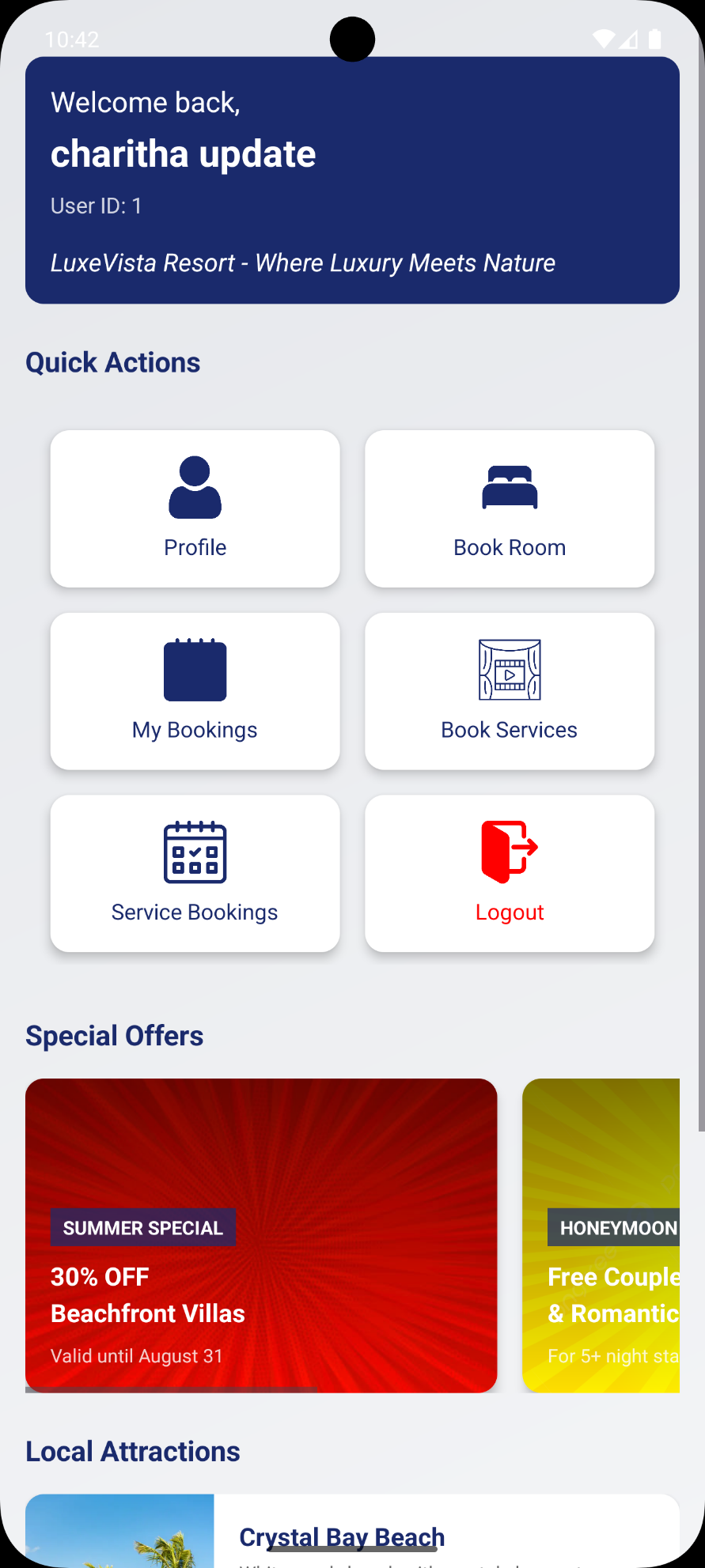
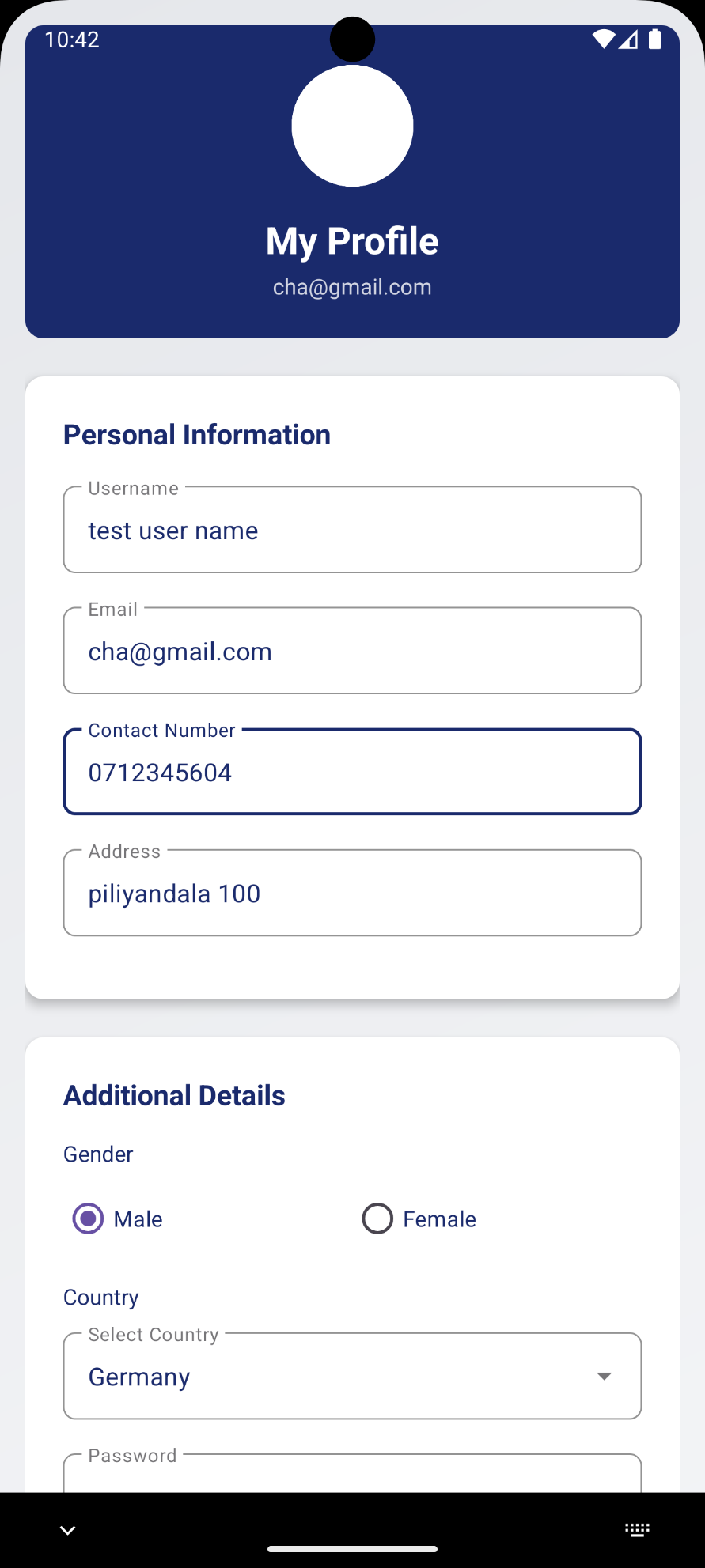
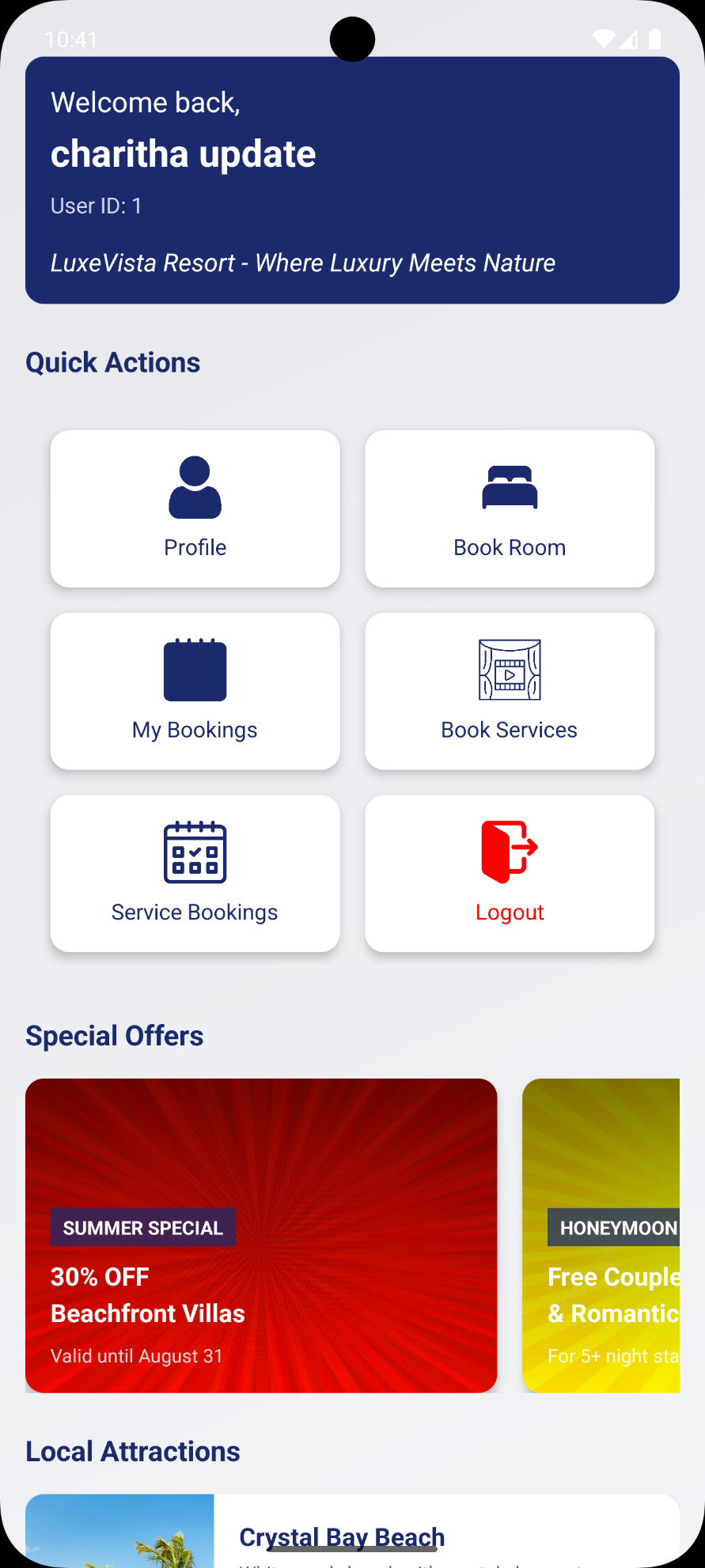
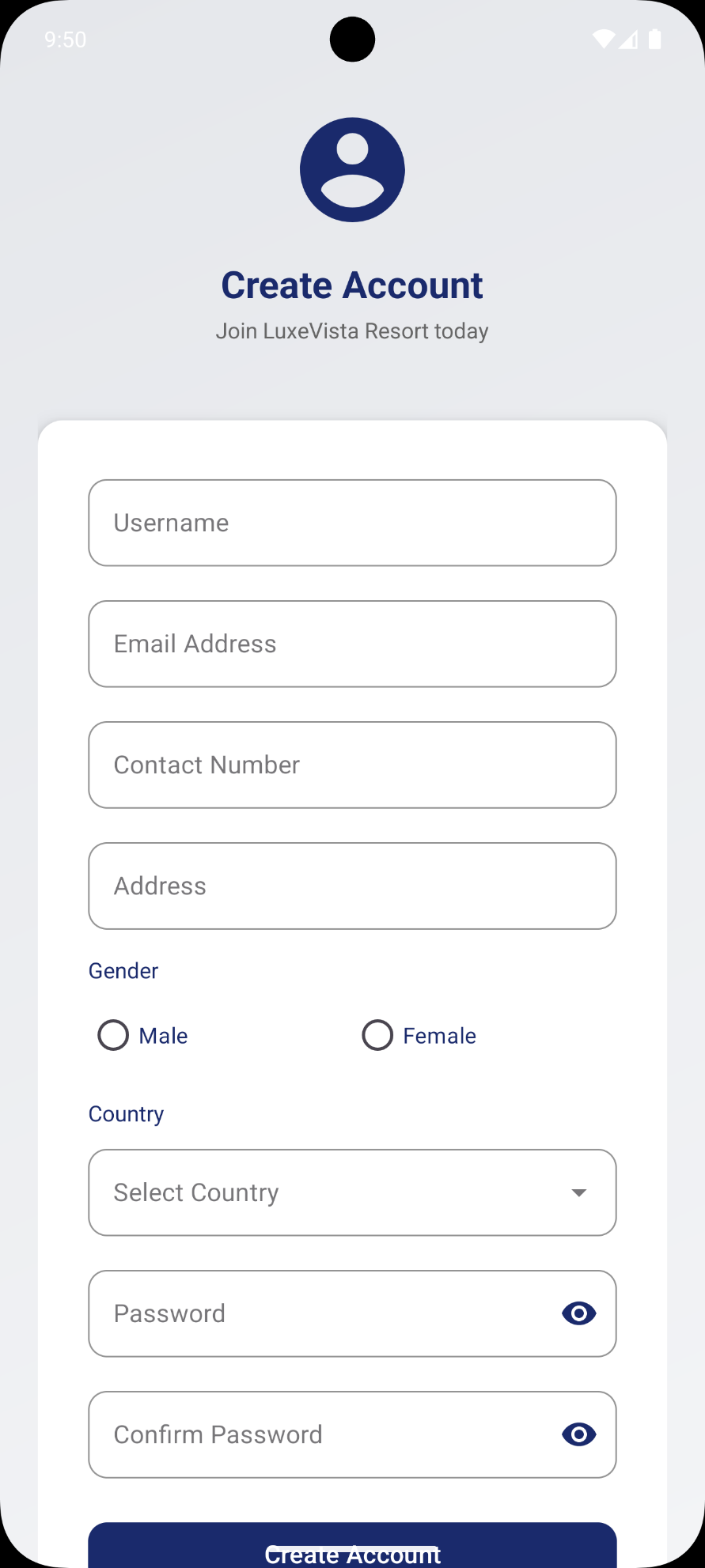
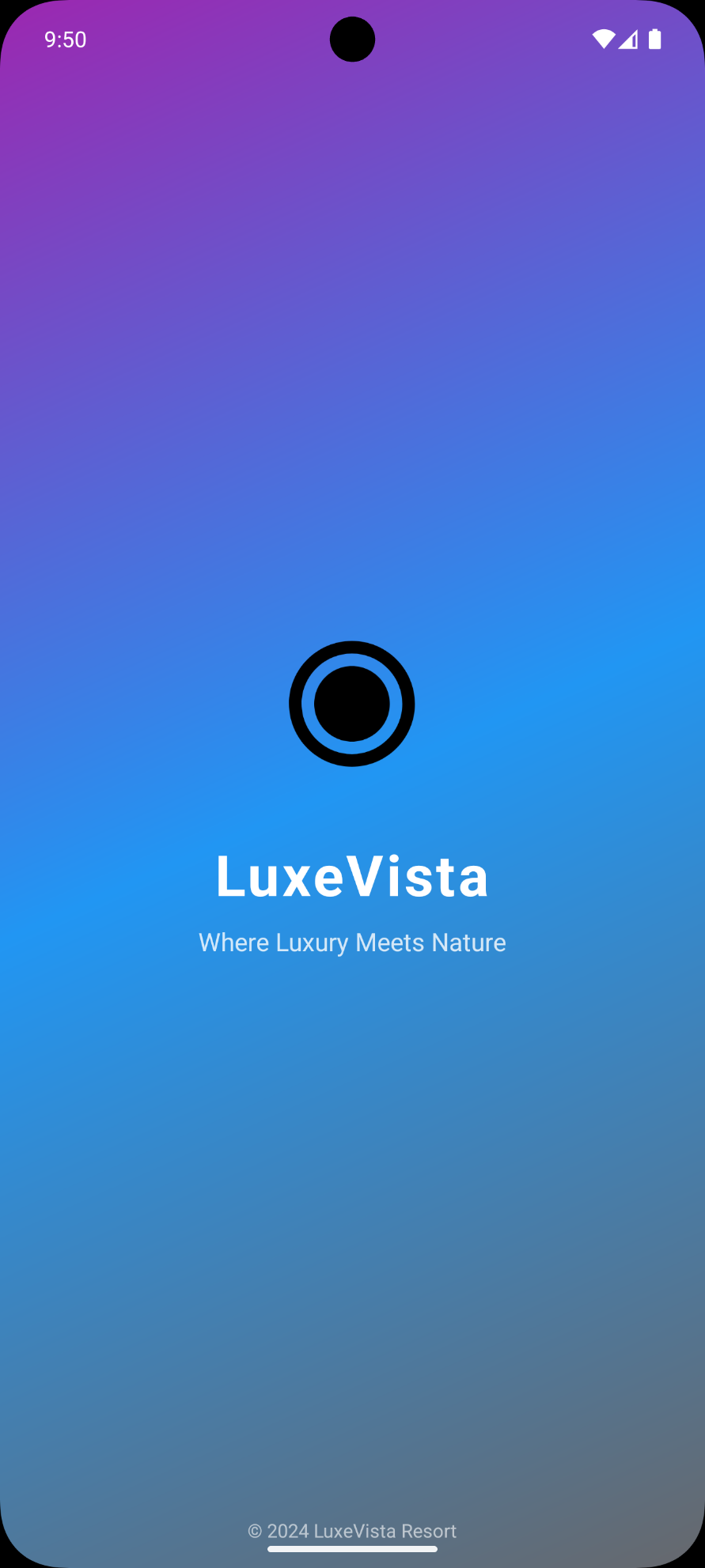








# TASK C



# TASK E

#### Test Case 1: User Registration

Test ID TC-001

Objective Verify new user can register successfully

Preconditions Fresh application install

Test Steps 1. Launch app

2. Navigate to Register screen

3. Fill all required fields

4. Click Register button

Expected Result User registered successfully, redirected to Login screen

Actual Result Pass - User created successfully

Status Completed

#### Test Case 2: User Login

Test ID TC-002

Objective Verify registered user can login

Preconditions User account exists in database

Test Steps 1. Launch app

2. Enter valid credentials

3. Click Login button

Expected Result Successful login, redirected to Guest Dashboard

Actual Result : Pass - Login successful, dashboard loaded

Status Completed

#### Test Case 3: Admin Login

Test ID TC-003

Objective Verify admin login functionality

Preconditions Admin account exists (hardcoded)

Test Steps 1. Launch app

2. Enter admin credentials

3. Click Login button

Expected Result Successful admin login, redirected to Admin Dashboard

Actual Result Pass - Admin dashboard accessible

Status Completed

#### Test Case 4: Room Booking

Test ID TC-004

Objective Verify user can book a room

Preconditions User logged in, rooms available

Test Steps 1. Navigate to Room Booking

2. Select room, dates, guests

3. Confirm booking

Expected Result Booking confirmed, appears in My Bookings

Actual Result Pass - Booking created successfully

Status Completed

#### Test Case 5: Service Booking

Test ID TC-005

Objective Verify user can book a service

Preconditions User logged in, services available

Test Steps 1. Navigate to Service Booking

2. Select service, date, time

3. Confirm booking

Expected Result Service booking confirmed, appears in My Service Bookings

Actual Result Pass - Service booking created

Status Completed

#### Test Case 6: Profile Update

Test ID TC-006

Objective Verify user can update profile information

Preconditions User logged in

Test Steps 1. Navigate to Profile

2. Update contact information

3. Save changes

Expected Result Profile updated successfully, changes persisted

Actual Result Pass - Profile data updated correctly

Status Completed

#### Test Case 7: Admin Room Management

Test ID TC-007

Objective Verify admin can manage rooms

Preconditions Admin logged in

Test Steps 1. Navigate to Manage Rooms

2. Add new room

3. Edit existing room

4. Delete room

Expected Result Room operations successful, changes reflected in database

Actual Result Pass - CRUD operations working for rooms

Status Completed

#### Test Case 8: Admin Service Management

Test ID TC-008

Objective Verify admin can manage services

Preconditions Admin logged in

Test Steps 1. Navigate to Manage Services

2. Add new service

3. Edit existing service

4. Delete service

Expected Result Service operations successful, changes reflected

Actual Result Pass - Services managed successfully

Status Completed

#### Test Case 9: Booking Cancellation (User)

Test ID TC-009

Objective Verify user can cancel bookings

Preconditions User has active bookings

Test Steps 1. Navigate to My Bookings

2. Select booking to cancel

3. Confirm cancellation

Expected Result Booking status updated to "Cancelled"

Actual Result Pass - Booking cancellation working

Status Completed

#### Test Case 10: Booking Cancellation (Admin)

Test ID TC-010

Objective Verify admin can cancel any booking

Preconditions Admin logged in, bookings exist

Test Steps 1. Navigate to Manage Bookings

2. Select booking to cancel

3. Confirm cancellation

Expected Result Booking status updated to "Cancelled"

Actual Result Pass - Admin cancellation working

Status Completed