ADITUS - Secure Access Control System

M3 - Project Report

Miguel Vila – 107276
Project Repository – https://github.com/miguelovila/ua-cm-aditus

0. Project Overview

Aditus (Latin) means approach, access, entrance, or way in, deriving from ad (to) + ire (to go). It signifies a passage, opening, or opportunity to reach a place.

The project provides a modern solution for access control, replacing traditional keys and access cards with personal mobile devices. The system is built with security as a primary focus, utilizing cryptographic signatures to verify user identity and authorize access.

1. Introduction & Motivation

Access control is one of those problems that has existed forever but hasn't really evolved much beyond physical keys and magnetic cards. If you've ever been a student at a university or worked in an office building, you get handed a physical key or an access card, which you then have to carry around everywhere. Lose it, and you're locked out until someone can issue you a new one. Forget it at home, and your day just got a lot more complicated. Even worse, if someone finds your lost key, there's no way to know who's using it or when.

This is where Aditus comes in. The idea is simple: why carry a physical key when everyone already carries their phone everywhere? The motivation behind this project was to build a modern access control system that uses the devices people already have in their pockets and on their wrists. Instead of fumbling around for keys, you just pull out your phone or tap your smartwatch, and the door opens.

This isn't just about convenience. Traditional access control systems have real security problems. Physical keys can be copied without anyone knowing. Access cards can be cloned. Once someone has a copy, there's no audit trail showing who actually opened the door. Aditus addresses these issues by using cryptographic authentication. Every unlock attempt is logged with details about who tried to access which door, from which device, and whether they succeeded or failed. If someone's phone gets stolen, an administrator can immediately revoke access for that specific device without affecting the user's other devices or requiring everyone else to get new keys.

Another motivation was to explore wearable integration. Smartwatches have become incredibly popular, but they're underutilized for access control. Aditus includes a Wear OS companion app that lets users unlock doors directly from their wrist.

The result is a four-component system that includes a Flutter mobile app for Android smartphones, a Flutter companion app for Wear OS smartwatches, an ESP32-based door controller that handles the physical unlocking mechanism, and a Flask backend service that manages users, permissions, and access logs. Each component plays a specific role, and together they create a secure, and user-friendly access control solution.

2. Solution Overview

2.1. Requirements

Building a mobile access control system meant thinking carefully about what the system needs to do and what constraints it needs to operate within.

Starting with functional requirements, the system needed to handle user authentication in a secure way. Users should be able to log in with their email and password, and the system should issue JWT tokens that can be refreshed without requiring the user to log in again constantly. Beyond just logging in, users need a way to protect access to the app itself, which is why both PIN codes and biometric authentication are supported. If someone picks up your unlocked phone, they still shouldn't be able to open doors.

Device registration was another critical functional requirement. When a user logs in from a new phone or smartwatch, the app needs to generate a unique cryptographic keypair for that specific device and register it with the backend. The system should support multiple devices per user because people often have more than one phone or might upgrade their device. Each device needs to be independently manageable so that if one gets lost or stolen, it can be revoked without affecting the others.

The core functionality is door unlocking. The app needs to discover nearby doors using Bluetooth Low Energy, display them in a list sorted by signal strength so the closest doors appear first, and allow the user to initiate an unlock request. The unlock process involves communication between the mobile app, the ESP32 door controller, and the backend server to verify permissions and authenticate the request cryptographically.

Access control management was a requirement that became more complex as we thought through real-world scenarios. Administrators need to be able to create and manage user accounts since this isn't a public system where anyone can sign up. They need to organize users into groups, like "Engineering Students" or "Security Personnel", and then grant access permissions either to individual users or entire groups. The system also needs to support exceptions, which act like a blacklist. For example, maybe the Engineering Students group has access to a lab, but one specific student had their access revoked due to breaking lab rules. The exception system handles that scenario.

Logging every access attempt was also a requirement. The system records who tried to unlock which door, from which device, whether they succeeded or failed, why they failed if they did. This creates an audit trail that can be used for security monitoring or investigating incidents. Users should be able to see their own access history.

The smartwatch companion app is a stripped down version of the smartphone app. It first needs to pair with the user's smartphone through a simple code-based system. Once paired, the watch should be able to unlock doors independently without needing the phone nearby, which means it needs its own cryptographic keys and can communicate directly with ESP32 controllers via BLE.

2.2. Features

The features of Aditus can be understood by explaining what different user types can accomplish with the system.

2.2.1. Regular Users

Regular users begin their journey with authentication. The app prompts them to log in with credentials provided by an administrator, since self-registration isn't allowed. Upon successful login, the backend returns JWT tokens that enable subsequent API calls. Users then set up a six-digit PIN code for app protection and can optionally enable biometric authentication for convenience.

Device registration happens automatically after first login. The app generates an RSA-2048 keypair, storing the private key securely on the device while sending the public key to the backend. Users can view their registered devices in the account section, rename them for clarity, or revoke access by deleting devices they no longer use or got lost.

The core functionality is **door unlocking**. Users tap the scan button to discover nearby doors via BLE. The app displays doors sorted by signal strength and cross-references them with backend data to show meaningful names and locations. When a user selects a door, the app initiates a BLE connection to the ESP32 controller and executes the cryptographic challenge-response flow.

The **access history** screen shows all past unlock attempts with timestamps, door names, and failure reasons when applicable. **Account management** features allow users to update profile information, change passwords, modify their PIN, and customize theme settings including light, dark, or system-default modes with optional custom color schemes.

2.2.2. Admin Users

Administrators access additional capabilities through an admin tab. **User management** displays all system users with their names, emails, and roles. Administrators can **view detailed information about users** including group memberships, registered devices, accessible doors, and any exceptions. They can **create new accounts**, **edit user** details, change roles between user and admin, or **delete accounts entirely**.

Group management lets administrators organize users efficiently. They can create groups with names and descriptions, add multiple members at once, and view which doors each group can access. This approach scales better than individual permission management, particularly for scenarios like granting an entire class access to a lab.

Door management handles **registration of physical doors in the system**. Administrators enter door names, location descriptions, and ESP32 MAC addresses. Doors can be marked active or inactive for temporary access control without deletion. The access control screen implements a permission system where **administrators can grant access to individual users or entire groups, and create exceptions that deny access even when other rules would allow it**. The system evaluates rules with exceptions taking highest priority, followed by direct access, then group access.

2.2.3. Smartwatch Usage

The smartwatch companion app provides quick door access capabilities. Pairing uses a six-digit code generated on the smartphone and entered on the watch so the watch is associated with the user. The watch validates this code with the backend, receives credentials, and generates its own keypair independent from the phone. Once paired, **users can unlock the closest door directly from their wrist without needing their phone present**.

3. System Architecture

3.1. High-Level Architecture

Aditus is built around four main components. The **mobile app** runs on the user's smartphone, the **smartwatch app** runs on their wrist, the **ESP32 controller** is physically installed on each door, and the **Flask backend service** runs on a server somewhere in the cloud. Each component has a specific job and communicates with the others using different protocols depending on what makes sense for that particular interaction.

The mobile and smartwatch apps communicate with the backend using HTTPS. The apps authenticate these requests using JWT tokens that are included in the Authorization header.

Between the mobile devices and the ESP32 door controllers, the communication happens over Bluetooth Low Energy. BLE was chosen because it's designed for short-range communication with low power consumption, which is perfect for this use case. The ESP32 advertises itself as a BLE peripheral, and the mobile app acts as a central device that scans for and connects to nearby doors.

The ESP32 controllers also communicate with the backend over HTTP, but unlike the mobile apps that might be on cellular networks or different WiFi networks, the ESP32s are on a local WiFi network and need to reach out to the backend server over the internet. They authenticate using a shared API key configured in the firmware when the device is set up for simplification.

Data flows through the system in a few different patterns depending on what's happening. When a user logs in, data flows from the mobile app to the backend and back. The backend validates credentials, checks the database, and returns tokens and user information. When unlocking a door, data flows in a more complex pattern: the mobile app uses BLE discover nearby doors and to talk to the ESP32 of the specific door, which then talks to the backend again to verify permissions and get the user's public key, and finally the result flows back through BLE to the mobile app. This ensures that the ESP32 always has the latest permission information and public keys without having to store them locally and **makes it possible for the mobile device (smartphone or smartwatch) to unlock doors even without internet access since all communication happens through BLE**.

The architecture is intentionally distributed rather than having everything go through a central point. The mobile app doesn't proxy communication between the ESP32 and the backend. Instead, the ESP32 has its own direct connection to the backend. This means that even if the mobile app loses internet connectivity, the ESP32 can still complete the verification process.

3.2. Backend Architecture

The backend is built with Flask, chosen for its simplicity and focus on RESTful API development. The system doesn't need server-side rendering or complex form handling, just a JSON API that the mobile apps and ESP32 controllers can communicate with over HTTPS.

SQLAlchemy provides the ORM layer that maps Python objects to database tables. During development, SQLite stores the database in a single file, while production deployment can use PostgreSQL with minimal code changes. The schema consists of six main tables that handle users, devices, doors, groups, access control, and logging.

Access control uses a priority-based system implemented across four tables. Direct user permissions and group permissions grant access, while user exceptions and group exceptions deny it. When an ESP32 queries whether a user can access a door, the backend checks exceptions first, then direct permissions, then group permissions. This hierarchy gives administrators fine-grained control over who can access which doors.

3.3 Mobile App Architecture

The Flutter apps for both smartphone and smartwatch follow similar architectural patterns, though the smartwatch app is simplified due to the constraints of the Wear OS platform and smaller screen size. The smartphone app is where the full architecture is most visible, so that's what I'll focus on here.

3.3.1 Clean Architecture Pattern

The app is structured using clean architecture principles, meaning that the code is organized into three main layers: **presentation, domain, and data**. Each layer has a specific responsibility and depends on the layers below it but not above it.

The **presentation layer is what the user interacts with**. It includes all the Flutter widgets that render the UI, the screens that represent different pages of the app, and the BLoC classes that manage state for those screens. The presentation layer is allowed to know about the domain layer but doesn't know anything about the data layer directly. If a screen needs to fetch data, it asks the domain layer through a use case.

The **domain layer contains the business logic of the application**. This is where use cases live, which are classes that represent specific actions a user can take, like "register a device" or "unlock a door." Each use case is a focused piece of logic that might involve multiple steps. For example, the register device use case generates a keypair, saves the private key to secure storage, and calls the repository to register the public key with the backend.

The **data layer is where the actual implementation happens**. Repositories in the data layer **implement the interfaces defined in the domain layer**. They handle API calls, data transformation, error handling, etc... The data layer also includes model classes that represent the JSON structures returned by the API and service classes that handle specific tasks like BLE communication or cryptographic operations.

The reason for this separation is maintainability (testing doesn’t apply here since I do not have any tests implemented). But, if we want to test a use case, we can just simply mock the repository interface without needing to make real API calls.

In practice, the architecture looks like this: A screen creates a BLoC and provides it through a BlocProvider. The screen listens to the BLoC's state stream and rebuilds the UI when the state changes. When the user interacts with the UI, like tapping a button, the screen dispatches an event to the BLoC. The BLoC receives the event and calls a use case from the domain layer. The use case interacts with repositories, which make API calls or perform other data operations. The results flow back up through the layers, eventually resulting in a new state being emitted by the BLoC, which triggers a UI rebuild.

3.3.2 Feature-Based Organization

The app is organized by feature. This means all the code related to authentication lives in a features/auth directory, all the code related to device management lives in features/device, and so on. Within each feature directory, the code is then organized by layer, so you have auth/data, auth/domain, and auth/presentation.

This organization made it easier to understand and modify features because all the related code is co-located around the same folders. If you need to add a new field to the login form, you know exactly where to look: features/auth/presentation.

Shared code that doesn't belong to a specific feature lives in the core directory. This includes things like the API service classes that handle HTTP communication, the BLE service singleton that manages Bluetooth operations, cryptographic utilities, secure storage services, theme definitions, and reusable widgets.

3.3.3 Widget Tree & UI Structure

The app's widget tree starts with a MaterialApp widget wrapped in a BlocProvider. Actually, it's wrapped in a MultiBlocProvider because there are two BLoCs that need to be available throughout the entire app: ThemeCubit for managing the app's theme settings and AuthBloc for managing authentication state.

The MaterialApp is configured with both a theme and darkTheme, which are provided by the AppTheme class based on the current theme preferences. It also uses DynamicColorBuilder from the dynamic_color package to support Material You theming on recent Android versions. This builder gives us access to the system's dynamic color scheme, which is derived from the user's wallpaper.

The MaterialApp's home property is set to an AppInitializer widget, which is a simple widget that checks the authentication state and decides which screen to show. If the AuthBloc is in the AuthUnauthenticated state, it shows the LoginScreen. If it's in the AuthSuccess state, it uses the AppRouter to determine whether the user needs to set up a PIN, register a device, or go straight to the home screen.

The HomeScreen is the main screen users see after logging in. It uses a NavigationBar at the bottom with destinations for Unlock, History, Groups, Account, and Admin (if the user is an admin).

Widgets are composed from smaller, reusable pieces. The core/ui/widgets directory includes things like ErrorState and EmptyState widgets that display standard messages, GravatarAvatar for displaying user profile pictures, and various other helpers

3.4 ESP32 Firmware

The ESP32 firmware is written in C++ using the Arduino framework, which provides a simpler programming model. The Arduino framework abstracts away a lot of the low-level details of the ESP32's hardware while still giving access to all the important features like WiFi, BLE, and cryptography.

The firmware's main responsibilities are to advertise itself as a BLE peripheral so mobile devices can discover it, handle BLE connections and characteristic reads/writes, communicate with the backend over WiFi to verify access permissions and fetch public keys, perform RSA signature verification using the mbedtls library, control the physical door lock mechanism, and send access logs back to the backend after each unlock attempt.

The BLE implementation uses the BLEDevice, BLEServer, and BLEService classes from the ESP32 BLE Arduino library. During setup, the firmware creates a BLE server and defines a service with a specific UUID that mobile apps filter for when scanning. This service includes four characteristics, each with its own UUID. The ID characteristic is writable by the client and receives a JSON payload containing the user_id and device_id. The challenge characteristic is readable and notifiable, allowing the ESP32 to send the random challenge to the mobile app. The signature characteristic is writable and receives the signed challenge from the mobile app. And the status characteristic is notifiable, letting the ESP32 send the final result back to the app.

Each characteristic has callbacks that are triggered when a client writes to it or reads from it. The most important callback is on the ID characteristic. When the ESP32 receives a write to this characteristic, it parses the JSON to extract the user_id and device_id, then immediately makes an HTTP request to the backend's /api/doors/check-access endpoint to verify whether this user should be allowed to access this specific door. The backend responds with either allowed: true or allowed: false along with a reason. If access is denied, the process stops and a denied status is sent back to the mobile app. If access is allowed, the ESP32 continues to the next step.

The next step is generating a random challenge. The ESP32 uses the random() function to generate a six-digit number, which is converted to a string. This challenge is then written to the challenge characteristic and the characteristic's notify flag is set, which triggers a notification to the mobile app.

The ESP32 then enters a waiting state where it expects the mobile app to write a signature to the signature characteristic within a timeout period. When the signature arrives, the firmware makes another HTTP request to the backend to fetch the device's public key. The endpoint is /api/devices/{device_id}/public-key and requires the ESP32 API key for authentication. The backend returns the public key in PEM format as a JSON string. The ESP32 then uses mbedtls functions to parse this PEM-encoded key and verify the signature.

The signature verification process involves initializing an mbedtls RSA context, parsing the public key into that context, decoding the base64-encoded signature received from the mobile app, and running the mbedtls RSA PKCS1 verification function with the challenge as the input and the signature as the expected output. If verification succeeds, the door is unlocked. If it fails, access is denied.

Unlocking the door is represented in the code by a placeholder function that would control a GPIO pin connected to an electronic lock. In a real deployment, this might trigger a relay that temporarily releases a magnetic lock or sends a signal to a servo motor. The current implementation just prints a message to the serial console and blinks an LED to indicate success.

4. State Management - BLoC Pattern

4.1 Why BLoC?

State management is the most important architectural decision in any Flutter application. For Aditus, the BLoC pattern was chosen for several specific reasons that align with the project's requirements and complexity.

BLoC stands for Business Logic Component, and the core idea is to separate business logic from the UI completely. The UI should only be responsible for rendering what it's told to render and reporting user interactions. All the actual logic, like making API calls, validating data, and coordinating between different services, lives in BLoC classes. This separation makes the codebase much easier to test because you can test business logic without needing to instantiate any widgets or run the Flutter framework at all.

4.2 Authentication BLoC - Just an Example

The AuthBloc is one of the most critical BLoCs in the application because it manages the user's authentication state and drives navigation throughout the app. It's provided at the root of the widget tree using MultiBlocProvider, making it accessible from anywhere in the app without needing to pass it down through constructors.

The AuthBloc handles four events: **AuthInitializeRequested, AuthLoginRequested, AuthLogoutRequested, and AuthForgotPinRequested.** The initialize event is dispatched when the app starts up, and it's responsible for checking if the user has valid credentials stored on the device. If they do, the user is automatically logged in without having to enter their password again. If not, they're shown the login screen.

When AuthInitializeRequested is received, the BLoC first emits an AuthInitializing state to indicate that work is being done. It then checks secure storage for access and refresh tokens. If both tokens exist, the BLoC attempts to fetch the current user's information from the backend using the access token. This API call serves two purposes: it retrieves the user data that the app needs, and it validates that the token is still valid. If the request succeeds, the BLoC emits an AuthSuccess state containing the user object and tokens. If the request fails with a 401 unauthorized error, the token might be expired, so the BLoC attempts to use the refresh token to get a new access token. If that succeeds, it tries fetching the user data again. If any of this fails, the BLoC clears the stored tokens and emits AuthUnauthenticated.

The login flow is more straightforward. When AuthLoginRequested is received with an email and password, the BLoC emits AuthLoading to show a loading indicator in the UI. It then calls the AuthApiService to make a POST request to the backend's login endpoint. The backend validates the credentials and returns tokens and user data if they're correct. The BLoC saves these to secure storage and emits AuthSuccess. If the login fails, an AuthFailure state is emitted with an error message that the UI can display.

The AuthBloc demonstrates several important patterns. It manages multiple async operations with proper error handling, making sure that network failures or invalid responses don't crash the app. It coordinates with multiple services: the API service for network calls and the secure storage service for persisting data. And it drives navigation implicitly through state changes rather than explicitly calling navigation methods.

4.3 BLoC Best Practices Used

Several BLoC best practices are evident throughout the Aditus codebase, and understanding why these practices matter helps explain the decisions made during implementation.

Every BLoC follows the single responsibility principle. The AuthBloc only handles authentication, not device management or door unlocking. The DoorUnlockBloc only handles unlocking doors, not discovering them. This keeps each BLoC focused and prevents them from growing into giant classes that do everything.

The BLoCs don't know anything about the UI. They don't import any Flutter widgets, don't know what a BuildContext is, and don't trigger navigation directly. All they do is emit states. This maintains the separation of concerns. The UI layer is responsible for responding to state changes, whether that means showing different widgets or navigating to different screens.

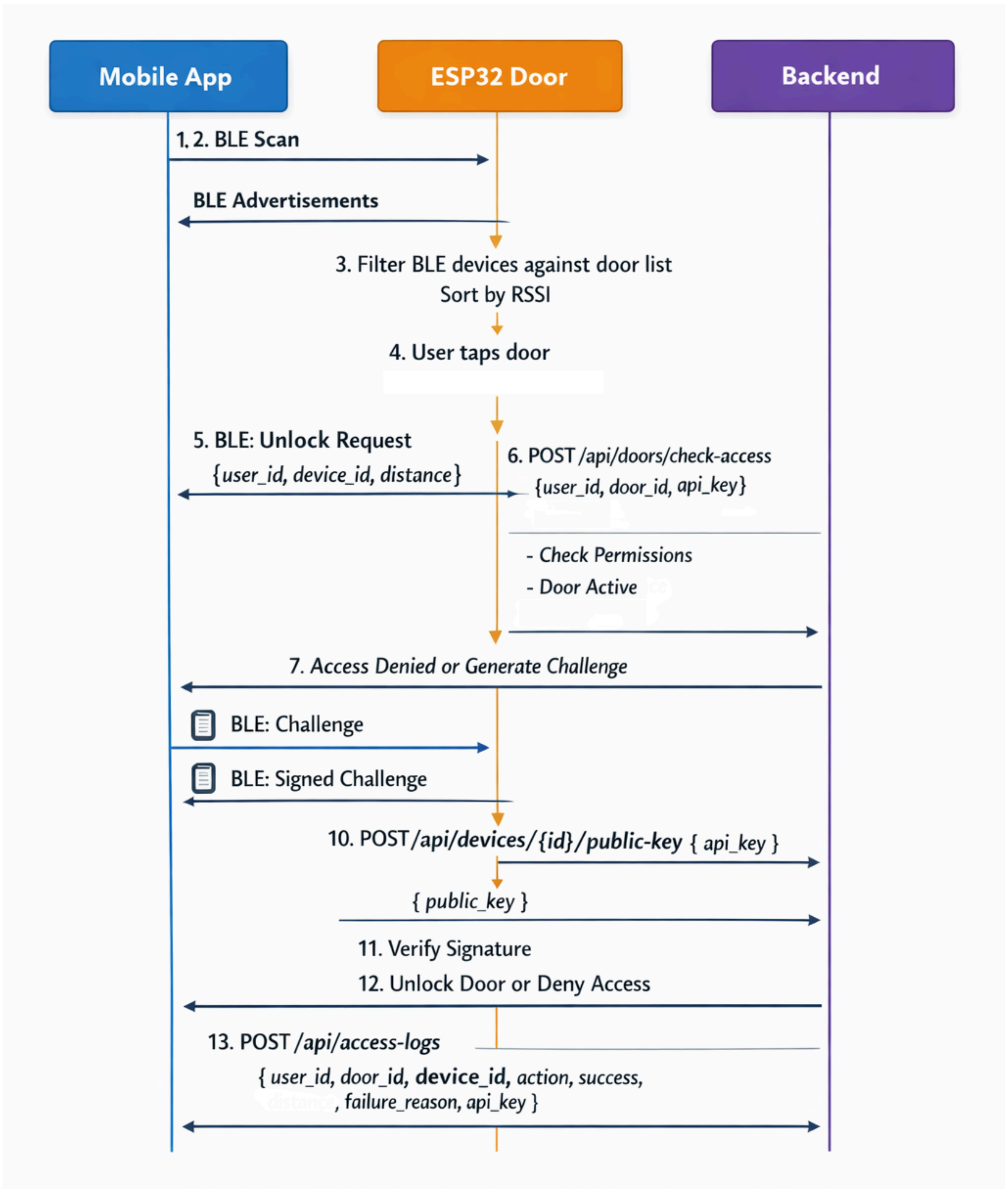
5. Some Technical Details

5.1 Smartphone Door Unlock Flow

The door unlock process is the most complex interaction in the entire system, involving all four components and multiple communication protocols.

The process actually begins when the user opens the app and enters their PIN or uses biometric authentication, the AuthBloc initializes and validates their stored tokens. If the tokens are valid, the app transitions to the home screen. This is the first checkpoint: only authenticated users can proceed.

When the user navigates to the unlock tab and taps the scan button, the app starts a BLE scan filtered by the door service UUID. The BLE service on the mobile device calls **FlutterBluePlus.startScan with the service UUID 4fafc201-1fb5-459e-8fcc-c5c9c331914b** and sets the scan mode to low latency for continuous updates. As the scan runs, the ESP32 controllers within range advertise themselves, and their BLE advertisements are picked up by the mobile device. Each advertisement includes the device's MAC address and signal strength (RSSI). The results are sorted by RSSI so the closest door appears at the top of the list.



Mobile App
ESP32 Door
Backend
1, 2. BLE Scan
BLE Advertisements
3. Filter BLE devices against door list
Sort by RSSI
4. User taps door
5. BLE: Unlock Request
{user_id, device_id, distance}
6. POST /api/doors/check-access
{user_id, door_id, api_key}
- Check Permissions
- Door Active
7. Access Denied or Generate Challenge
BLE: Challenge
BLE: Signed Challenge
10. POST /api/devices/{id}/public-key { api_key }
{ public_key }
11. Verify Signature
12. Unlock Door or Deny Access
13. POST /api/access-logs
{ user_id, door_id, device_id, action, success,
, failure_reason, api_key }

Image 1: BLE unlocking flow

When the user selects a door and taps unlock, several things happen rapidly. First, the app initiates a BLE connection to the ESP32. The BLE service calls connectToDevice with the door's MAC address, which was obtained from the backend. The ESP32, which has been advertised as a BLE peripheral, accepts the connection. The mobile device then performs service discovery to enumerate the ESP32's GATT services and characteristics. It finds the door service and its four characteristics: ID, challenge, signature, and status.

The app subscribes to notifications on both the challenge and status characteristics before doing anything else. This is important because if the ESP32 sends a notification before the app is listening, it would be lost. The subscription is done by calling setNotifyValue(true) on each characteristic, which sends a BLE command to the ESP32 telling it to enable notifications.

With notifications set up, the app creates a JSON payload containing the user_id and device_id. The user_id identifies which user is making the request, and the device_id identifies which of that user's devices is being used. This matters because each device has its own cryptographic keys. The JSON is encoded as UTF-8 bytes and written to the ID characteristic using the BLE write operation.

The ESP32 receives this write on the ID characteristic, which triggers a callback in the firmware. The callback parses the JSON to extract the user_id and device_id, then immediately makes an HTTP POST request to the backend at /api/doors/check-access. This request includes the user_id, the door_id (which the ESP32 knows because it's configured with which door it controls), the distance the mobile app calculated, and the ESP32's API key for authentication.

The backend processes this access check request by querying the database. It first checks for exceptions. If the user is in a group that's blacklisted from this door, access is denied with the reason "exception_group". If the user themselves is blacklisted, it's "exception_user". If there are no exceptions blocking access, the backend checks for direct access grants. If there's a record in the allowed_users table linking this user to this door, access is granted with the reason "direct_access". If there's no direct grant, the backend checks if the user is in any groups that have access to this door. If so, access is granted with "group_access". If none of these conditions are met, access is denied with "no_permission".

The backend's response is sent back to the ESP32 as JSON. If access is denied, the ESP32 writes a denial reason to the status characteristic, triggers a notification to send it to the mobile app, and logs the failed attempt to the backend. The process ends here, and the user sees an error message explaining why they were denied.

If access is allowed, the ESP32 generates a random six-digit challenge. The random number generator is seeded during the ESP32's startup, so each challenge is different. The challenge is converted to a string and written to the challenge characteristic, and a notification is triggered. The mobile app receives this notification through the stream it subscribed to earlier.

When the challenge arrives at the mobile app, it's passed to the CryptoService along with the device's private key. The CryptoService uses the pointycastle library to perform RSA signing with SHA-256 hashing. The process involves hashing the challenge string using SHA-256 to produce a fixed-size digest, then encrypting that digest with the private key to produce the signature. This signature is essentially proof that the mobile device possesses the private key corresponding to the public key the backend has on file. The signature is base64-encoded and sent back to the ESP32 by writing it to the signature characteristic.

The ESP32 receives the signature and now needs to verify it. But it doesn't have the user's public key stored locally because keys might be revoked or updated. So it makes another HTTP request to the backend at /api/devices/{device_id}/public-key with the ESP32 API key. The backend looks up the device in the database and returns its public key as a PEM-encoded string.

The ESP32 uses the mbedtls library to parse the PEM string and extract the RSA public key. It then performs signature verification, which is the inverse of signing. The signature is decrypted using the public key to reveal what should be the SHA-256 hash of the challenge. The ESP32 also computes the SHA-256 hash of the original challenge it sent. If these two hashes match, the signature is valid, which proves the mobile device has the correct private key.

If verification succeeds, the ESP32 writes "AUTHORIZED" to the status characteristic and triggers a notification. It also activates the door lock mechanism, which in a real deployment would be a GPIO pin toggling a relay or sending a signal to an electronic lock. The mobile app receives the authorized status and shows a success message. If verification fails, the ESP32 writes a denial reason and the process ends with an error.

5.1 Smartwatch Pairing

Pairing a smartwatch to the Aditus system could have been done through direct Bluetooth pairing between the phone and watch, but that approach has limitations. The watch needs credentials to authenticate with the backend, and those credentials shouldn't be transferred over Bluetooth between the devices without encryption. The chosen approach uses a pairing code system similar to what many two-factor authentication apps use, and it's both more secure and more flexible.

The flow starts on the smartphone app. The user navigates to the account section and finds a smartwatch pairing option. When they tap it, the app generates a random six-digit numeric code. This code is generated using a secure random number generator to ensure it's not predictable. The app then makes an HTTP POST request to the backend at /api/pairing-sessions with the user's JWT token and the generated code. The backend creates a record in the PairingSession table with the code, the user's ID, and an expiration timestamp set to a few minutes in the future.

The smartphone app displays the six-digit code to the user in large numbers on the screen. The user is instructed to enter this code on their smartwatch. They open the smartwatch app, which detects that the watch hasn't been paired yet and shows a pairing screen. The user enters the six digits using the watch's input method, which on Wear OS is typically a numeric keypad optimized for small screens.

When the user submits the code, the smartwatch app makes an HTTP POST request to the backend at /api/pairing-sessions/validate with the code and no authentication. This is one of the few endpoints that doesn't require a JWT token because the watch doesn't have one yet. The backend looks up the code in the PairingSession table. If the code doesn't exist, the request fails with "invalid code". If the code exists but has expired, it fails with "code expired". If the code is valid and hasn't expired, the backend returns the user's credentials, including their user ID, email, and full name. The backend also generates a new access token and refresh token specifically for the watch and returns those.

The smartwatch app receives these credentials and stores them in the watch's secure storage using flutter_secure_storage. On Wear OS, this uses the Android KeyStore system, which keeps sensitive data encrypted and protected. The watch now has authentication tokens that it can use to make API calls to the backend.

Next, the smartwatch app generates its own RSA-2048 keypair using the same cryptographic library the smartphone app uses, pointycastle. The process is identical: generate random prime numbers, create the public and private keys, encode them as PEM. The private key is saved to secure storage and never leaves the watch. The public key is sent to the backend to register the watch as a new device.
Now, the watch is completely independent of the smartphone. Even if the phone is turned off, dead, or at home, the watch can still unlock doors. It establishes its own BLE communication with ESP32 controllers. The only thing the phone was needed for was the initial pairing to transfer credentials securely.

If the user later wants to unpair the watch, they can do it from either the smartphone app or the watch itself. The unpair action deletes the device record from the backend, which immediately revokes the watch's ability to unlock doors.

5.2 Data Persistence

Data persistence in Aditus happens at multiple levels, from temporary in-memory caching to permanent storage in the backend database.

On mobile devices, the most sensitive data is stored using flutter_secure_storage, which provides platform-specific secure storage mechanisms. On Android, this uses EncryptedSharedPreferences, which stores data in an encrypted format using keys derived from the Android KeyStore. The KeyStore is hardware-backed on devices that support it, meaning the encryption keys are stored in a secure enclave that's physically separate from the main processor.

The data stored in secure storage includes JWT tokens (both access and refresh), the user's cryptographic keypair (private key and public key in PEM format), the device ID returned by the backend during registration, the PIN hash (not the PIN itself, but a hashed version), user data like email and full name, and biometric authentication preferences.

The app doesn't do much caching of API responses. Most screens fetch fresh data from the backend every time they're opened. This keeps the implementation simple and ensures users always see current data, but it does mean more network requests. The one exception is the AuthBloc, which stores the user object in memory after fetching it from the backend during initialization.

On the backend, all persistent data lives in the SQLAlchemy database. During development, this is SQLite, which stores the entire database in a single file. SQLite is convenient for development because there's no server to run, but it has limitations around concurrent writes. For production deployment, the recommendation is to use PostgreSQL.

5.3 UI/UX Design Decisions

The user interface of Aditus is built around Material Design 3 (my personal taste really), which is Google's latest design system. MD3 emphasizes personal expression through dynamic color, smooth animations, and accessible layouts. The decision to use MD3 instead of sticking with Material Design 2 or creating a custom design system came down to being so much easier.

Theme **switching between light and dark modes is handled by the ThemeCubit,** which is a simplified version of a BLoC that manages a single piece of state. Users can choose between light, dark, or system-default themes. The system-default option uses ThemeMode.system, which tells Flutter to follow the device's dark mode setting. When the user switches the device to dark mode system-wide, the app automatically switches without requiring any action from the user.

6. Overall Assessment

6.1 What Was Achieved

Looking at the project, it accomplishes what it set out to do: replace physical keys with a secure mobile-based access control solution. The project delivers a working system where users can unlock doors using either their smartphone or smartwatch through cryptographic authentication. This isn't just a proof of concept or a prototype with missing pieces. It's a functional system that handles the entire lifecycle from user registration through device management to actual door access.

The authentication system works as intended. Users can log in with their credentials, and the app maintains their session using JWT tokens that refresh automatically without bothering the user. The addition of PIN and biometric protection provides an extra security layer that makes sense for a mobile app. Even if someone picks up an unlocked phone, they can't use Aditus without the PIN or fingerprint.

Device registration achieves the goal of making **each smartphone or smartwatch a unique authentication device**. The RSA key generation happens locally, private keys never leave the device, and the backend only stores public keys. This architecture means that compromising the backend doesn't give an attacker the ability to forge signatures. Each device is independently registered and can be independently revoked.

The door unlock functionality demonstrates that the challenge-response cryptographic system works in practice. The access logging creates an audit trail that can be reviewed later. The admin features provide management capabilities. Creating users, organizing them into groups, and configuring door access permissions all work as you'd expect from an admin dashboard.

The smartwatch integration is interesting because it demonstrates that the architecture is flexible enough to support multiple device types. The pairing system using codes is simple and secure. Once paired, the watch operates completely independently of the phone, which is exactly what you want. Being able to unlock a door from your wrist when your phone is in your bag or pocket is a real usability improvement.

From a software engineering perspective, the project demonstrates clean architecture principles in a real application. The separation between presentation, domain, and data layers is consistent throughout the codebase. The BLoC pattern is applied consistently across all features, making the code predictable and maintainable. The feature-based organization keeps related code together, which makes it easier to understand and modify specific functionality.

6.2 Issues & Limitations

This project has some issues and limitations that are worth discussing. The most significant known issue is that smartwatch unlocking occasionally crashes the ESP32. This doesn't happen with smartphone unlocking, only with the watch. Through testing, the problem appears to be related to the Pixel Watch BLE stack and how it handles certain operation sequences. The issue manifests as the ESP32 completely hanging and requiring a reset.Everything points to device-specific BLE driver behavior rather than application problem.

The ESP32 doesn't have any secure storage for its WiFi credentials or API key. These are hardcoded in the firmware, which means anyone who can read the flash memory can extract them. In a production deployment, we would want to use the ESP32's secure boot and flash encryption features to protect this data. The current approach is fine for development and testing but not for production.

There's no offline mode for door unlocking (ESP32 needs to be able to always contact the backend). If the backend is unreachable, doors can't be unlocked even if the user has valid local credentials. A more robust system might issue time-limited offline tokens that the ESP32 could verify without contacting the backend.

The admin interface is functional but basic. There's no search or filtering on large lists. You can't bulk edit permissions.