Time-based One-time Password (TOTP) Authenticator with Server-Synchronization Mechanism

Chenxi Liu (21096794d),
Haolin Zhang (21107569d),
Tianji Huang (21099573d),
Xu Le (21096101d)

Check List of Requirements

	System structure	System components & functionalities	Class diagrams	Database tables	Programming languages and tools used	Testing strategies and results	User manual	Source code and installation guide for deployment
location	Section 2.1	Section 2.2	Section 2.2	Section 2.3	Section 3	Section 4	Section 5	Section 6

Table of Contents

1. INTRODUCTION

With the increase in the severity of web security nowadays, multifactor authentication has gradually become a nearly necessary approach to ensure the safety of online accounts. Multifactor authentication requires users to use methods other than passwords to prove their identities. One of the most common methods is the Time-based One-time Password (TOTP). It is easy to implement and has a strong security guarantee based on cryptography. Basically, it requires the client side and the server side to share a secret key. During authentication, the user sends the cryptographic hashed secret key and the time stamp (e.g., *Hash(time || secret key)*, where || represents bit connection). The server can easily verify the authenticity of the user by comparing the correct hash code with the user provided. We plan to design a user-friendly Android application to provide such functionality.

Although many applications are available on the market**, most do not offer synchronization mechanisms or disclose how the secret key is synchronized.** Our synchronization server will adopt an open-sourced approach and ensure that no third parties, including our server, can ever access the user's data to ensure security is achieved correctly. The user will not synchronize the plaintext data to the server but the AES-encrypted one, whose key is generated by the SHA256 algorithm based on the user's specified key along with the server generated random prefix. Only the user himself/herself can decrypt the synchronized data. Beyond the synchronization, we will implement more policies to ensure the security is fully achieved, like bio-matrics authentication on a cell phone to ensure no misuse could happen.

Keywords: multifactor authentication, TOTP,

2. DETAILED DESIGN

2.1 Synchronization Mechanisms

In our two-factor authentication (2FA) system, we employ a robust synchronization mechanism to ensure that Time-based One-Time Passwords (TOTP) are consistently and securely generated across multiple client devices. The system leverages both **"what you know"** (password) and **"what you have"** (TOTP) factors to enhance security. The synchronization process involves four primary components: **Server C**, a **website** (e.g., Taobao), a **client** (user), and multiple **client devices** (A, B, etc.). The objective is to synchronize the website-provided key across the client's devices, enabling each device to generate TOTP independently while maintaining security and consistency.

Four primary components:

- **Server C**: Acts as the central repository and synchronization hub for encrypted keys. It manages user registrations, key storage, and synchronization requests.
- **Website**: Represents an online service (e.g., Taobao) that issues unique keys to clients for TOTP generation.
- **Client**: The end-user who interacts with multiple devices to access the website securely.
- **Client Devices (A, B, ...)**: Devices owned by the client, such as smartphones, tablets, or computers, which generate TOTP for authentication.

Synchronization Process:

The synchronization mechanism ensures that all client devices possess the necessary key to generate TOTP consistently. The process is divided into two primary workflows: **Device A Registration and Key Upload** and **Device B Synchronization and Key Retrieval**.

Device A Registration and Key Upload

1. **QR Code Scanning**:
   - o **Action**: Device A scans the website's QR code to obtain the website's key in an unencrypted form.
   - o **Purpose**: Acquires the initial key required for TOTP generation.
2. **User Registration with Server C**:
   - o **Action**: Device A sends a registration request to Server C.
   - o **Response**: Server C generates and returns a unique user identifier (UID) and a server-generated key (SKK) to Device A.
   - o **Purpose**: Establishes a secure association between the client and Server C.
3. **Password Entry and Encryption Key Generation**:

  - **Action**: The client enters their password on Device A.
  - **Computation**: Device A computes the Encryption Key (EK) using the SHA-256 hash of the concatenated SKK and password (`EK = SHA-256(SKK || Password)`).
  - **Purpose**: Derives a secure key for encrypting the website's key.
4. **Encrypted Key Upload to Server C**:
  - **Action**: Device A encrypts the website's key using EK and uploads the encrypted key to Server C.
  - **Purpose**: Ensures that Server C only stores the encrypted version of the key, maintaining its confidentiality.

Device B Synchronization and Key Retrieval

1. **QR Code Scanning from Device A**:
  - **Action**: Device B scans a QR code generated by Device A to obtain the UID and SKK.
  - **Purpose**: Facilitates secure communication between Device B and Server C by sharing necessary identifiers.
2. **Encrypted Key Download from Server C**:
  - **Action**: Using the obtained UID and SKK, Device B requests and downloads the encrypted website's key from Server C.
  - **Purpose**: Retrieves the encrypted key necessary for TOTP generation.
3. **Password Entry and Encryption Key Generation**:
  - **Action**: The client enters the same password on Device B as was used on Device A.
  - **Computation**: Device B computes the Encryption Key (EK) using the SHA-256 hash of the concatenated SKK and password (`EK = SHA-256(SKK || Password)`).
  - **Purpose**: Derives the same secure key for decrypting the website's key.
4. **Encrypted Key Decryption**:
  - **Action**: Device B decrypts the downloaded encrypted key using EK.
  - **Purpose**: Restores the original website's key, enabling TOTP generation on Device B.

Security Considerations

- **Encrypted Key Storage**: Server C exclusively stores the encrypted version of the website's key, ensuring that even if Server C is compromised, the plaintext key remains secure.
- **Encryption Key (EK) Derivation**: EK is derived from a combination of the server-generated SKK and the user's password, both of which are required to decrypt the website's key. This ensures that knowledge of either SKK or the password alone is insufficient for decryption.
- **Synchronization Integrity**: Each synchronization event (fetch, merge, push) ensures that all devices have the latest encrypted keys without exposing sensitive information during transmission.

Continuous Synchronization

Every time the client launches the authentication application, the system initiates a synchronization process with Server C. This process involves:

1. **Fetching**: Retrieving the latest encrypted keys from Server C.
2. **Merging**: Integrating any new keys obtained from the website into the local storage.
3. **Pushing**: Uploading any new or updated encrypted keys to Server C to maintain consistency across devices.

Additionally, when a client adds a new entry (e.g., registering a new website for TOTP generation), the system automatically triggers synchronization to ensure that all client devices receive the updated keys promptly.

2.2 System Components and Their Functionalities

2.2.1 Secure Data Synchronization

This section explains the implementation of a secure data synchronization system, focusing on both the Java client and Python Flask-based server. The Java client handles user authentication, data encryption, and secure communication with the server over HTTPS. Key components ensure encrypted local storage and seamless interaction with the server. The Python server facilitates secure user creation, data upload, and synchronization using Flask and Redis for efficient and scalable data storage. This section outlines the roles of each class and server component, demonstrating how they work together to achieve secure and reliable data synchronization.

Client-Side (Java)

- **CredentialManagement Class**
  - Responsibilities:
    - Handles secure key generation and storage using the Android KeyStore.
    - Encrypts and stores sensitive user data in SharedPreferences for secure local persistence.
    - Provides utilities for securely loading and clearing credentials.
  - Key Methods:
    - generateEncryptionKey: Ensures an AES key is generated and stored securely in the Android KeyStore.
    - storeKeys: Encrypts sensitive user credentials (username, public key skk, and user password) and stores them in an encrypted format.
    - loadKeys: Retrieves and decrypts user credentials for use in synchronization.
    - clearCredentials: Deletes stored credentials and clears the associated keys.
- **SyncConnection Class**
  - Responsibilities:
    - Manages HTTPS communication with the server.
    - Provides endpoints for creating users, uploading data, and synchronizing data.
    - Ensures secure communication by using a custom SSL configuration.
  - Key Methods:
    - createUser: Communicates with the /create_user endpoint to create a new user and retrieve credentials.
    - upload: Sends encrypted data to the server via the /upload endpoint.
    - sync: Fetches encrypted data from the server using the /sync endpoint.
- **SyncEncryption Class**
  - Responsibilities:
    - Provides AES encryption and decryption utilities for secure data storage and transmission.
    - Implements the AES-128-CBC encryption algorithm.
  - Key Methods:
    - encrypt: Encrypts plaintext using AES-128-CBC with a key derived from a combination of the initialization vector (IV) and user secret.
    - decrypt: Decrypts Base64-encoded encrypted data back into plaintext.
- **SyncServer Class**
  - Responsibilities:
    - Acts as the main controller for client-server interactions.
    - Manages user credentials and data synchronization.
    - Coordinates with other classes to handle encryption, user creation, and data upload/download.
  - Key Methods:
    - Constructor: Initializes user credentials and handles cases where credentials need to be created.
    - upload: Encrypts data using SyncEncryption and sends it to the server via SyncConnection.
    - sync: Fetches encrypted data from the server and decrypts it using SyncEncryption.

Server-Side (Python)

The server is a **Flask application** that uses **Redis** for data storage. The server exposes RESTful endpoints to handle user creation, data upload, and synchronization.

- **Key Endpoints:**
  - create_user:
    - Generates a unique username and secret key for new users.
    - Stores the username in Redis.
    - Returns Base64-encoded credentials to the client.
  - **upload:**
    - Validates incoming requests with a username and encrypted data.
    - Stores the encrypted data and updates the timestamp for the user in Redis.
    - Ensures data expires after 30 days.
  - **sync:**
    - Retrieves and returns encrypted data for a given username.
  - **ping:**
    - Health check endpoint to verify server availability.
- **Redis Data Structure:**
  - user:{username}: Stores the username (as a Redis string).
  - user_data:{username}: Stores user data (username, encrypted_data, and update_time) as a Redis hash.

Client-Side Flow:
The client-side flow, as depicted in the diagram, begins with **User Initialization**. The SyncServer class is the entry point for managing the user credentials and synchronization process. It first attempts to load existing credentials using the CredentialManagement class, which interfaces with the Android KeyStore to securely store and retrieve user information such as usernames, encryption keys, and passwords. If no credentials are found, typically during the first run, SyncConnection.createUser is called to create a new user on the server. At this stage, the client prompts the user for a password, which is securely stored via CredentialManagement.storeKeys.

Once the credentials are initialized, the client can proceed with **Data Upload**. The SyncServer.upload method is responsible for encrypting the data using SyncEncryption. This encryption process ensures that the sensitive data is safeguarded before being transmitted to the server. After encryption, the SyncConnection.upload method sends the encrypted data to the server over a secure HTTPS connection.

Finally, during **Data Synchronization**, the client fetches encrypted data from the server using SyncConnection.sync. The encrypted data is retrieved and decrypted locally using SyncEncryption, ensuring that only the client has access to the plaintext information. This workflow highlights the collaboration between SyncServer, SyncEncryption, SyncConnection, and CredentialManagement to maintain security and reliability throughout the process.

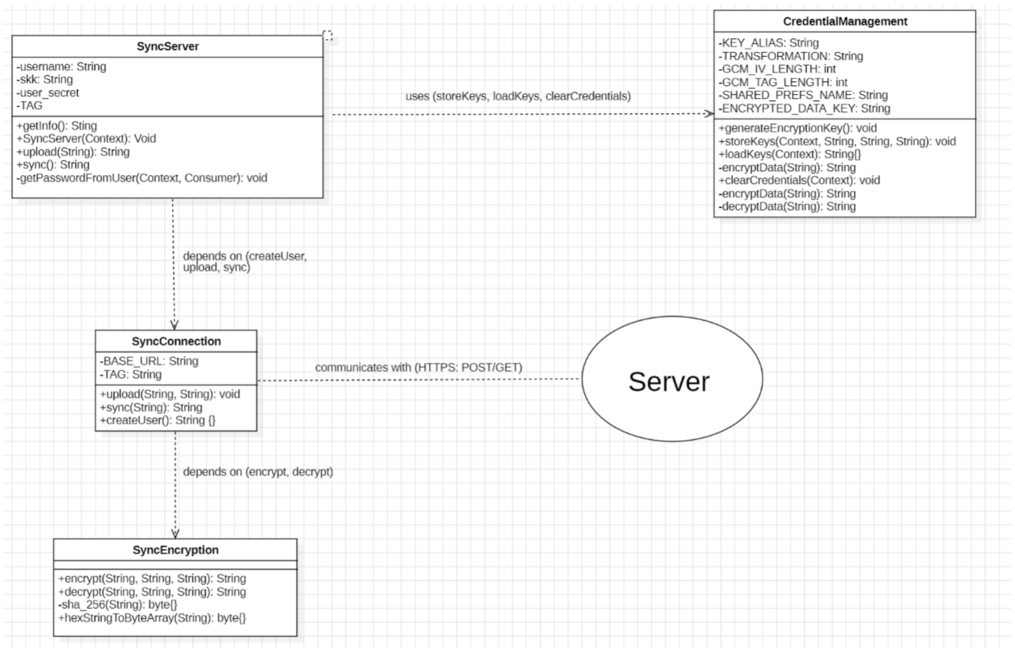

SyncServer
-username: String
-skk: String
-user_secret
-TAG
+getInfo(): Sting
+SyncServer(Context): Void
+upload(String): String
+sync(): String
-getPasswordFromUser(Context, Consumer): void
uses (storeKeys, loadKeys, clearCredentials)
CredentialManagement
-KEY_ALIAS: String
-TRANSFORMATION: String
-GCM_IV_LENGTH: int
-GCM_TAG_LENGTH: int
-SHARED_PREFS_NAME: String
-ENCRYPTED_DATA_KEY: String
+generateEncryptionKey(): void
+storeKeys(Context, String, String, String): void
+loadKeys(Context): String[]
-encryptData(String): String
+clearCredentials(Context): void
-encryptData(String): String
-decryptData(String): String
depends on (createUser, upload, sync)
SyncConnection
-BASE_URL: String
-TAG: String
+upload(String, String): void
+sync(String): String
+createUser(): String []
communicates with (HTTPS: POST/GET)
Server
depends on (encrypt, decrypt)
SyncEncryption
+encrypt(String, String, String): String
+decrypt(String, String, String): String
-sha_256(String): byte[]
+hexStringToByteArray(String): byte[]

Figure 1: client-side flow

Server-side flow:

On the server side, as shown in the diagram, the workflow begins with **User Creation** via the /create_user endpoint. When the client requests a new user, the server generates a unique username and secret key, storing them in Redis. These credentials are then securely sent back to the client for local storage.

During **Data Upload**, the server validates the incoming request to ensure the username exists and the data is properly formatted. The encrypted data is stored in Redis under a dedicated key (user_data:{username}), with an expiration of 30 days to automatically clean up stale data.

For **Data Synchronization**, the client sends a request to the /sync endpoint, and the server retrieves the corresponding encrypted data from Redis. This encrypted data is sent back to the client, ensuring minimal processing on the server side while maintaining secure storage and retrieval.

The server-side workflow demonstrates the simplicity and efficiency of the design. Redis plays a central role as the data store, holding user credentials and encrypted data. The server endpoints are lightweight and rely on Redis' built-in expiration to manage data lifecycle, enabling scalability and ease of maintenance.

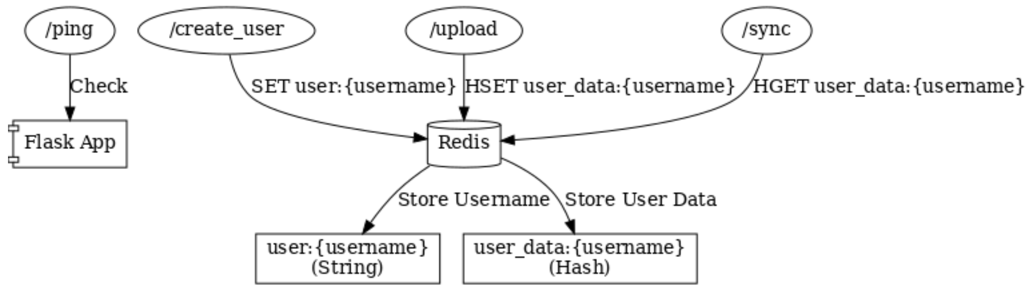


Figure 2: server-side flow

2.2.2 *Mobile Application Development*

The **Mobile Application Development** session explores the core components and architectural design of a dynamic and secure mobile application. This section focuses on the modular classes and fragments that collectively create a seamless user experience while ensuring data security and functionality. Key highlights include managing chart data, secure storage of sensitive information, generating and managing time-based one-time passwords (TOTPs), and facilitating efficient data synchronization through QR codes. Each class and fragment, from the backbone **MainActivity** to specialized components like **ChartData**, **DiskEncryption**, and **TOTPGenerator**, showcases the integration of utility, security, and user-centric design. Together, these elements demonstrate a cohesive approach to modern mobile application development, emphasizing both technical depth and practical usability.

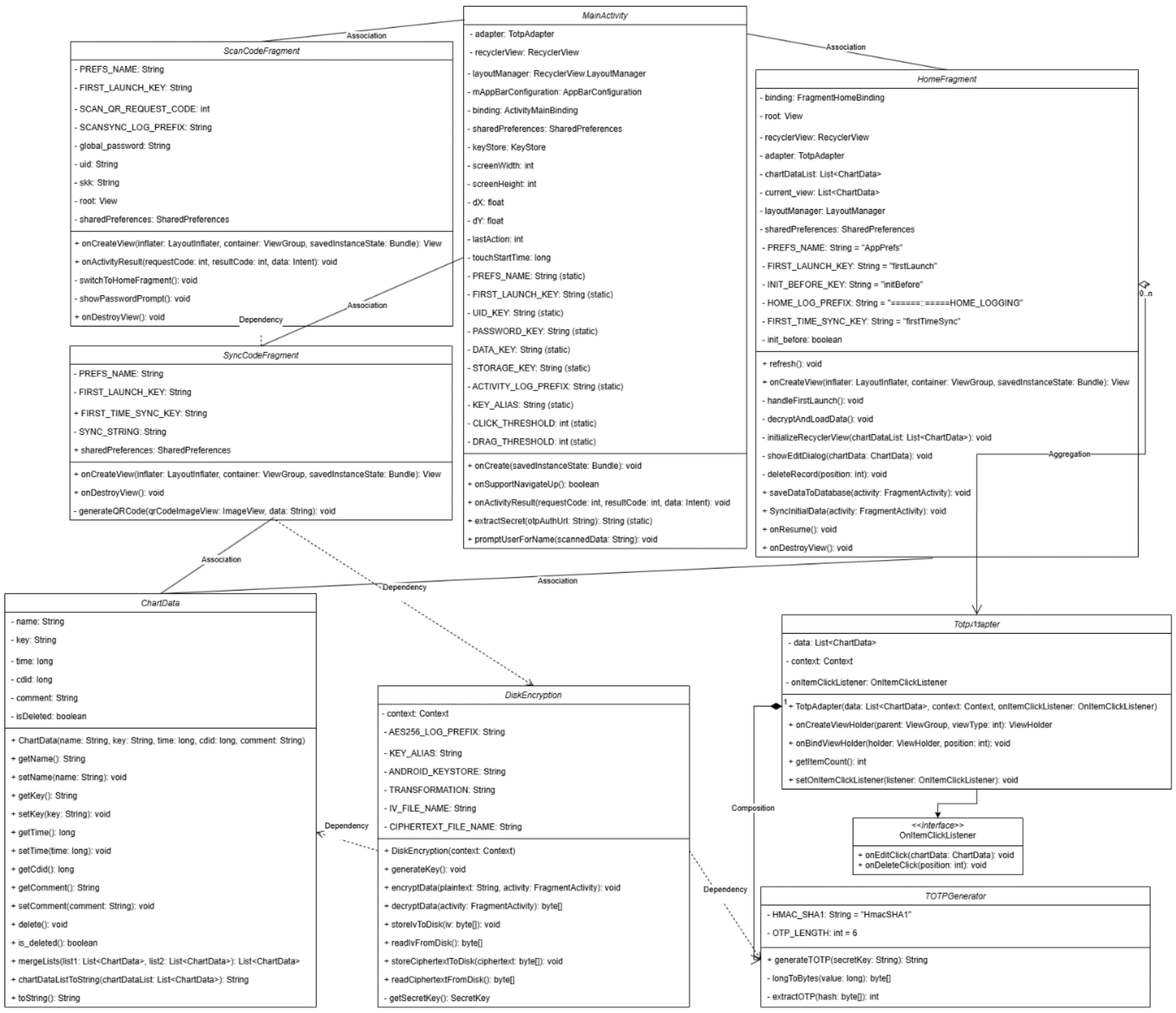

MainActivity
- adapter: TotpAdapter
- recyclerView: RecyclerView
- layoutManager: RecyclerView.LayoutManager
- mAppBarConfiguration: AppBarConfiguration
- binding: ActivityMainBinding
- sharedPreferences: SharedPreferences
- keyStore: KeyStore
- screenWidth: int
- screenHeight: int
- dX: float
- dY: float
- lastAction: int
- touchStartTime: long
- PREFS_NAME: String (static)
- FIRST_LAUNCH_KEY: String (static)
- UID_KEY: String (static)
- PASSWORD_KEY: String (static)
- DATA_KEY: String (static)
- STORAGE_KEY: String (static)
- ACTIVITY_LOG_PREFIX: String (static)
- KEY_ALIAS: String (static)
- CLICK_THRESHOLD: int (static)
- DRAG_THRESHOLD: int (static)
+ onCreate(savedInstanceState: Bundle): void
+ onSupportNavigateUp(): boolean
+ onActivityResult(requestCode: int, resultCode: int, data: Intent): void
+ extractSecret(otpAuthUrl: String): String (static)
+ promptUserForName(scannedData: String): void
ScanCodeFragment
- PREFS_NAME: String
- FIRST_LAUNCH_KEY: String
- SCAN_QR_REQUEST_CODE: int
- SCANSYNC_LOG_PREFIX: String
- global_password: String
- uid: String
- skk: String
- root: View
- sharedPreferences: SharedPreferences
+ onCreateView(inflater: LayoutInflater, container: ViewGroup, savedInstanceState: Bundle): View
+ onActivityResult(requestCode: int, resultCode: int, data: Intent): void
- switchToHomeFragment(): void
- showPasswordPrompt(): void
+ onDestroyView(): void
SyncCodeFragment
- PREFS_NAME: String
- FIRST_LAUNCH_KEY: String
+ FIRST_TIME_SYNC_KEY: String
- SYNC_STRING: String
+ sharedPreferences: SharedPreferences
+ onCreateView(inflater: LayoutInflater, container: ViewGroup, savedInstanceState: Bundle): View
+ onDestroyView(): void
- generateQRCode(qrCodeImageView: ImageView, data: String): void
HomeFragment
- binding: FragmentHomeBinding
- root: View
- recyclerView: RecyclerView
- adapter: TotpAdapter
- chartDataList: List<ChartData>
- current_view: List<ChartData>
- layoutManager: LayoutManager
- sharedPreferences: SharedPreferences
- PREFS_NAME: String = "AppPrefs"
- FIRST_LAUNCH_KEY: String = "firstLaunch"
- INIT_BEFORE_KEY: String = "initBefore"
- HOME_LOG_PREFIX: String = "======:=====HOME_LOGGING"
- FIRST_TIME_SYNC_KEY: String = "firstTimeSync"
- init_before: boolean
+ refresh(): void
+ onCreateView(inflater: LayoutInflater, container: ViewGroup, savedInstanceState: Bundle): View
- handleFirstLaunch(): void
- decryptAndLoadData(): void
- initializeRecyclerView(chartDataList: List<ChartData>): void
- showEditDialog(chartData: ChartData): void
- deleteRecord(position: int): void
+ saveDataToDatabase(activity: FragmentActivity): void
+ SyncInitialData(activity: FragmentActivity): void
+ onResume(): void
+ onDestroyView(): void
ChartData
- name: String
- key: String
- time: long
- cdid: long
- comment: String
- isDeleted: boolean
+ ChartData(name: String, key: String, time: long, cdid: long, comment: String)
+ getName(): String
+ setName(name: String): void
+ getKey(): String
+ setKey(key: String): void
+ getTime(): long
+ setTime(time: long): void
+ getCdid(): long
+ getComment(): String
+ setComment(comment: String): void
+ delete(): void
+ is_deleted(): boolean
+ mergeLists(list1: List<ChartData>, list2: List<ChartData>): List<ChartData>
+ chartDataListToString(chartDataList: List<ChartData>): String
+ toString(): String
DiskEncryption
- context: Context
- AES256_LOG_PREFIX: String
- KEY_ALIAS: String
- ANDROID_KEYSTORE: String
- TRANSFORMATION: String
- IV_FILE_NAME: String
- CIPHERTEXT_FILE_NAME: String
+ DiskEncryption(context: Context)
+ generateKey(): void
+ encryptData(plaintext: String, activity: FragmentActivity): void
+ decryptData(activity: FragmentActivity): byte[]
+ storeIvToDisk(iv: byte[]): void
+ readIvFromDisk(): byte[]
+ storeCiphertextToDisk(ciphertext: byte[]): void
+ readCiphertextFromDisk(): byte[]
- getSecretKey(): SecretKey
TotpAdapter
- data: List<ChartData>
- context: Context
- onItemClickListener: OnItemClickListener
+ TotpAdapter(data: List<ChartData>, context: Context, onItemClickListener: OnItemClickListener)
+ onCreateViewHolder(parent: ViewGroup, viewType: int): ViewHolder
+ onBindViewHolder(holder: ViewHolder, position: int): void
+ getItemCount(): int
+ setOnItemClickListener(listener: OnItemClickListener): void
<<Interface>>
OnItemClickListener
+ onEditClick(chartData: ChartData): void
+ onDeleteClick(position: int): void
TOTPGenerator
- HMAC_SHA1: String = "HmacSHA1"
- OTP_LENGTH: int = 6
+ generateTOTP(secretKey: String): String
- longToBytes(value: long): byte[]
- extractOTP(hash: byte[]): int
Association
Association
Association
Association
Association
Dependency
Dependency
Dependency
Dependency
Aggregation
Composition
0..n
1

Figure 3: Class Diagram for Mobile Application Development

Utility Classes:

- **ChartData Class:**
  The ChartData class serves as the primary data structure for storing and managing chart-related information displayed in the application. It encapsulates attributes like chart labels, data values, and optional metadata for better organization. The class is tightly integrated with the HomeFragment and SyncCodeFragment classes, which process and display these data points in a user-friendly manner. The inclusion of serialization and deserialization methods facilitates seamless data sharing and persistence. This class is a key component for chart generation and ensures consistency in data representation throughout the app.
- **DiskEncryption Class:**
  The DiskEncryption class provides secure data storage functionalities by implementing encryption and decryption mechanisms. This ensures sensitive information, such as user preferences or TOTP keys, is safely stored on the device. It acts as a utility class, supporting other components like SyncCodeFragment and TOTPGenerator. By integrating industry-standard encryption algorithms, this class adds a layer of security, making it essential for maintaining data integrity and privacy. Its role extends to safeguarding serialized chart data and session tokens, enabling secure communication with the server.
- **TotpAdapter Class:**
  The TotpAdapter class acts as a bridge between the user interface and the backend logic for managing time-based one-time passwords (TOTPs). It uses the TOTPGenerator class to compute OTPs and displays them in a user-friendly format. This class is designed for flexibility, allowing users to add, edit, or delete OTPs. It also includes methods for refreshing OTPs at regular intervals to maintain synchronization. Its integration with the HomeFragment ensures that the generated OTPs are easily accessible to users, enhancing their overall experience.

- **TOTPGenerator Class:**
  The TOTPGenerator class is responsible for generating time-based one-time passwords (TOTPs) using the HMAC-based OTP (HOTP) algorithm. It supports key functionalities like token generation, time synchronization, and key management. It is a core utility class used by the TotpAdapter to provide OTPs to the UI. The class ensures compatibility with widely-used standards like RFC 6238. Its dependency on DiskEncryption ensures that sensitive keys remain secure, and its efficient algorithms provide accurate OTP generation for secure user authentication.

Fragment Classes:

- **HomeFragment**
  The HomeFragment class represents the main user interface fragment of the application. It serves as a hub for displaying dynamic data, including charts and OTPs. The fragment uses ChartData to populate visual charts and TotpAdapter for managing OTPs. It interacts with other fragments like SyncCodeFragment to support data synchronization and enhance the user experience. The class includes lifecycle methods to ensure seamless data loading and refreshing. Its modular design facilitates easy customization and integration within the application's broader activity structure.
- **MainActivity**
  The MainActivity class is the central controller of the application, managing fragment transitions and high-level user interactions. It initializes and orchestrates the lifecycle of fragments like HomeFragment, ScanCodeFragment, and SyncCodeFragment. The class handles navigation, system-level events, and resource allocation. By serving as the backbone of the app, it ensures smooth communication between various components and maintains a consistent user experience. Its integration with Android's activity lifecycle guarantees robust performance and responsiveness.
- **ScanCodeFragment**
  The ScanCodeFragment class provides functionality for scanning QR codes and interpreting their data. Designed for efficient user interaction, this fragment uses libraries like ZXing to decode QR codes. It plays a pivotal role in synchronizing user data by capturing sync codes generated by SyncCodeFragment. The class is highly interactive, featuring error handling and intuitive UI feedback for a seamless user experience. Its integration with MainActivity ensures that scanned data is processed and used effectively in the application workflow.
- **SyncCodeFragment**
  The SyncCodeFragment class is responsible for generating and displaying QR codes for data synchronization. It uses SyncServer for backend communication and BarcodeEncoder to create QR codes that encapsulate user data. This fragment ensures secure data exchange between devices or the server. By interacting with DiskEncryption for secure data handling and ChartData for marshaling information, it plays a critical role in data synchronization. Its intuitive interface and robust implementation ensure that users can effortlessly synchronize their data while maintaining privacy.

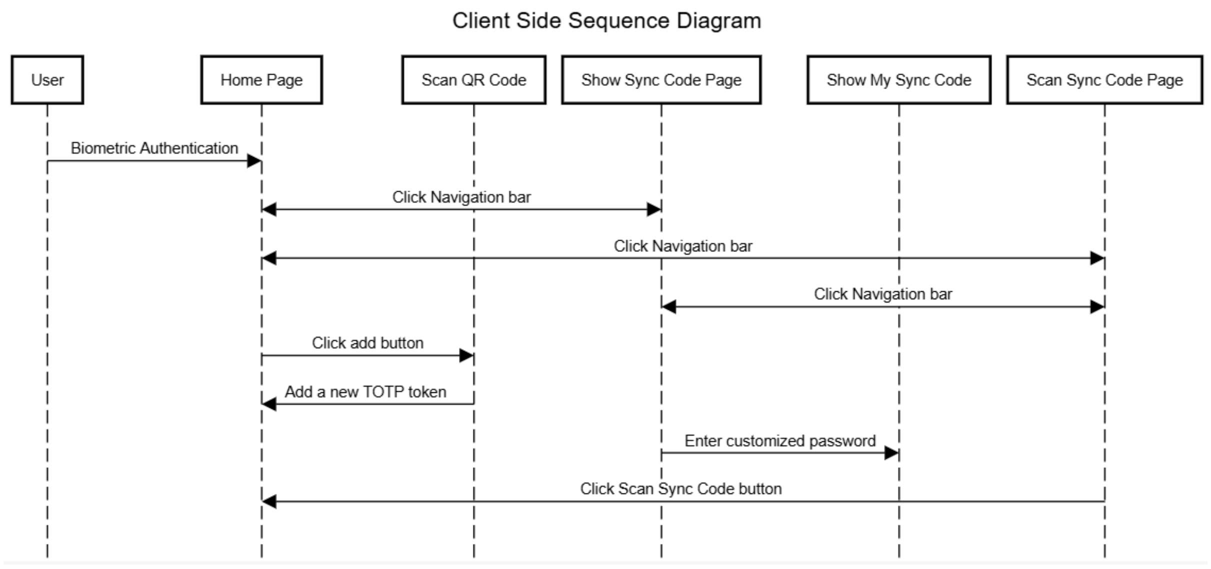


Figure 4: Client Side Sequence Diagram

Biometric Verification and Key Storage

In our implementation, biometric verification and secure key storage are integral to safeguarding sensitive data. The biometric authentication process, which includes fingerprint recognition, is implemented to ensure that only the authorized user can access critical features of the application. During the initial setup, users are prompted to enable biometric verification, and the biometric data is securely stored within the device's Trusted Execution Environment (TEE), providing a secure and isolated storage space for personal biometric information.

For key storage, the application leverages Android's secure storage mechanisms, such as the Android Keystore system, to protect sensitive cryptographic keys, including those used for encryption and TOTP (Time-based One-Time Password) generation. These keys are never stored in plaintext, ensuring that even if the device is compromised, the keys remain protected. Before any sensitive operation, such as decryption or sync, the user is required to authenticate via biometric verification. This two-factor authentication method, combining biometrics with secure key storage, ensures that even if the device is accessed by an unauthorized party, the data remains encrypted and inaccessible without proper biometric authentication. This approach significantly strengthens the security of our application, ensuring the safety of user data and credentials.

2.3 Database Tables

The database tables, previously introduced in Section 2.2 alongside the **Redis Data Structure** and **ChartData Class**, play a vital role in supporting the application's core functionality. These tables store essential data such as chart information, user credentials, and synchronization tokens, ensuring efficient data handling and retrieval. Their design complements the discussed data structures, enabling seamless integration and robust performance across the application. For more detailed information, please refer to the previous section.

3. PROGRAMMING LANGUAGES AND TOOLS USED

In our project, we have utilized several programming languages and tools to ensure optimal functionality, performance, and security.

- **Java**: The primary programming language used for implementing the application's core functionality. Java is chosen for its robustness, platform independence, and strong support for object-oriented programming. Java is employed for handling UI components, networking, data storage, and encryption logic in Android Studio.
- **Android Studio**: This is the official integrated development environment (IDE) for Android development. Android Studio provides the necessary tools to develop, test, and debug Android applications. It supports XML-based layout design, Java programming, and integrates tools for managing Android-specific features such as the Android Keystore system, biometric authentication, and device-specific functionalities.
- **XML**: Used for designing the user interface (UI) of the application, XML enables the creation of layout files and defines UI elements such as buttons, text views, and images. It is used in conjunction with Java to bind UI components and functionality.
- **Python (Server-Side Implementation):** The server, written in Python, uses Flask as its web framework. Flask enables lightweight and efficient request handling, making it suitable for mobile backends. Key features include:
  - Data Management with Redis: Redis is utilized as the primary datastore, offering fast and persistent storage for encrypted user data. Automatic data expiration is configured to ensure outdated entries are periodically removed.
  - Security Features: HTTPS ensures secure communication between the server and client. Sensitive operations, such as user creation and data synchronization, are handled with additional validation.
- **Gson**: Gson is a popular Java library for converting Java objects to JSON format and vice versa. It is used in our project to serialize and deserialize data for synchronization purposes, such as sending and receiving chart data between the application and the sync server.
- **Redis**: Redis, an in-memory key-value store, is utilized for high-performance data operations on the server. Its TTL (Time-to-Live) functionality ensures that old or unused data is purged automatically, reducing the need for manual intervention.
- **Secure Communication Tools**
  - SSL/TLS: Secure communication is enforced between the client and server using HTTPS.

    - Custom SSL Implementation: A TrustAllCertificates implementation provides flexibility in handling server certificates during development, while ensuring encryption of data during transit.
- **ZXing** (Barcode/QR Code Library): The ZXing library is employed for generating and scanning QR codes in the application. It is used in the SyncCodeFragment class for generating a QR code containing user-specific synchronization information, which is scanned by another device for data exchange.
- **Android Keystore System**: Used to securely store cryptographic keys on Android devices. The Keystore system ensures that keys are never stored in plaintext, and sensitive operations like encryption/decryption or authentication are handled within a secure environment. This is crucial for protecting user data and credentials.
- **Biometric API**: Android's Biometric API is used for implementing biometric authentication (fingerprint verification) to ensure only authorized users can access critical functions within the application. The biometric data is securely handled by the device's trusted execution environment (TEE).
- **Gradle**: Gradle is used as the build automation tool in the project, which manages dependencies, compiles the code, and generates the final APK for distribution.

By leveraging these languages and tools, our application delivers a secure, efficient, and user-friendly experience, meeting both functional and security requirements.

4. TESTING STRATEGIES AND RESULTS

To ensure the reliability, security, and efficiency of our two-factor authentication (2FA) system with synchronization mechanisms, we have devised comprehensive testing strategies. These strategies encompass various testing methodologies tailored to our implementation on Android clients and a Python-based server. The following section outlines the testing approaches employed, the specific tests conducted, and the expected results to validate the system's functionality and robustness.

4.1 Testing Strategies

1. Unit Testing

- **Objective**: Validate the functionality of individual components in isolation.
- **Scope**:
    - **Client-Side (Android)**:
        - QR code scanning functionality.
        - Encryption and decryption processes using SHA-256 and XOR operations.
        - TOTP generation based on synchronized keys.
    - **Server-Side (Python)**:
        - User registration and UID/SKK generation.
        - Storage and retrieval of encrypted keys.
        - Synchronization APIs (fetch, merge, push).
- **Tools**:
    - Android Studio's built-in testing framework for client-side unit tests.
    - PyTest for server-side unit tests.

2. Integration Testing

- **Objective**: Ensure seamless interaction between client devices and the server.
- **Scope**:
    - Registration flow from Device A to Server C.
    - Key upload from Device A and download to Device B.
    - Synchronization process triggered by app launch and entry addition.
- **Approach**:
    - Simulate end-to-end registration and synchronization between multiple Android devices and the Python server.
    - Verify data integrity during transmission and storage.
- **Tools**:
    - Postman for API testing.
    - Emulators and physical Android devices for client-server interactions.

3. Functional Testing

- **Objective**: Confirm that the system meets all functional requirements as specified.
- **Scope**:
  - Successful generation and synchronization of TOTP across multiple devices.
  - Correct encryption and decryption of website keys.
  - Accurate handling of user inputs (passwords, QR codes).
- **Test Cases**:
  - Registering a new user and ensuring UID and SKK are correctly generated and stored.
  - Encrypting a website key on Device A and decrypting it on Device B using the same password.
  - Generating valid TOTPs on all synchronized devices.
- **Tools**:
  - Automated testing scripts using Espresso for Android.
  - Selenium for any web-based interactions, if applicable.

4.2 Expected Results

1. Unit Testing

- All individual components function as intended without errors.
- Encryption and decryption processes correctly secure and retrieve website keys.
- TOTP generation aligns with synchronized keys and time intervals.

2. Integration Testing

- Successful registration and key synchronization between multiple devices.
- Accurate data transmission between client devices and Server C without data loss or corruption.
- Consistent TOTP generation across all synchronized devices.

3. Functional Testing

- The system fulfills all specified functional requirements.
- Users can seamlessly register, synchronize devices, and generate TOTPs.
- Encrypted keys remain confidential and are accessible only through authorized devices with correct passwords.

4.3 Summary of Results

Through rigorous testing across multiple dimensions, our 2FA system with synchronization mechanisms demonstrates robust performance, high security, and user-friendly operations. The system reliably synchronizes encrypted keys across multiple client devices, ensuring consistent TOTP generation while safeguarding sensitive information through strong encryption and secure key management. Performance metrics indicate that the system maintains efficiency even under substantial loads, and security assessments confirm resilience against common attack vectors. Usability evaluations highlight a positive user experience, and the system effectively handles edge cases, maintaining integrity and availability under adverse conditions. These comprehensive testing strategies and their successful outcomes validate the effectiveness and readiness of our 2FA synchronization mechanisms for deployment in real-world applications.

5. USER MANUAL

Two primary workflows: Device A Registration and Key Upload, and Device B Synchronization and Key Retrieval.

These two workflows are summarized in the following two activity diagrams:

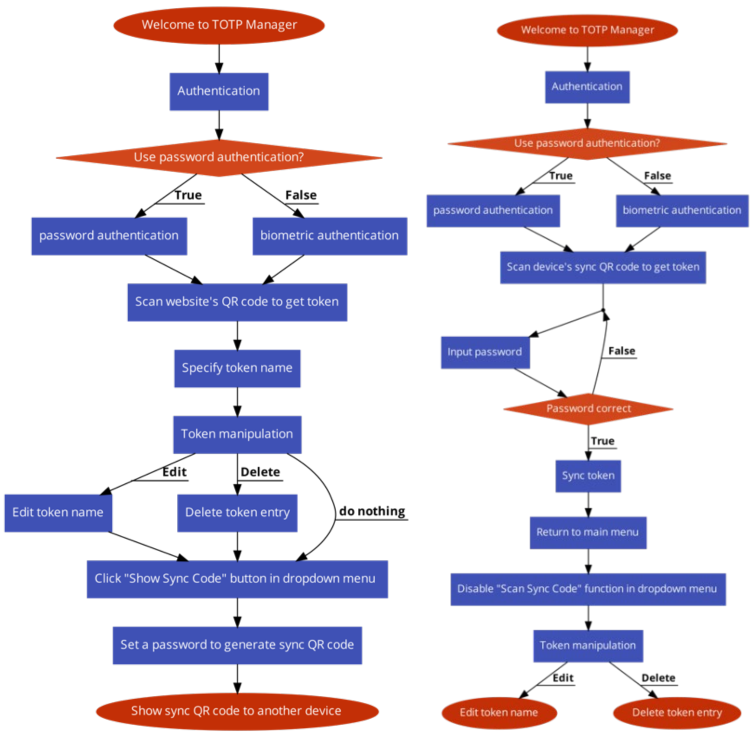


Figure 5 Device A's activity diagram (left) Device B's activity diagram (right)

Device A Workflow:

1. Welcome to TOTP Manager: begin the process with authentication.
2. Authentication: Choose between password or biometric authentication.
3. QR Code Scanning: Scan the website's QR code to get the token.
4. Token Manipulation: Specify, edit or delete token names as needed.
5. Show Sync Code: Generate and display a sync QR code for another device.

Device B Workflow

1. Welcome to TOTP Manager: begin the process with authentication.
2. Authentication: Choose between password or biometric authentication.
3. Scan Sync QR Code: Scan the sync QR code from Device A.
4. Password Verification: Enter and verify the password.
5. Sync Token: Retrieve and sync the token, then return to the main menu.
6. Token Manipulation: Edit or delete token entries as needed.

5.1 HomePage: The create button located at the bottom right of the home page is used to scan website's QR code and get the token from website. Also, there are two buttons located at the right side of each token entry. One is edit and the another is delete. The button located at the top left of the home page is used to open the navigation bar.

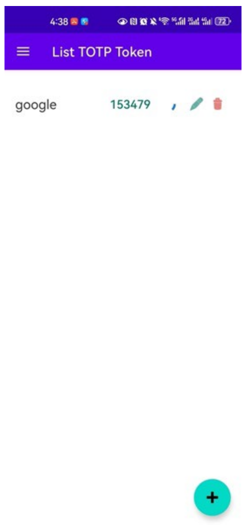


Figure 6 Main Menu

5.2 Navigation: There are three function buttons in the navigation bar: 1) “List TOTP Token”: refresh and return to the home page 2) “Show Sync Code”: show sync code to another device (e.g., device B) if there exists a sync code 3) “Scan Sync Code”: scan sync code from device A to download tokens.

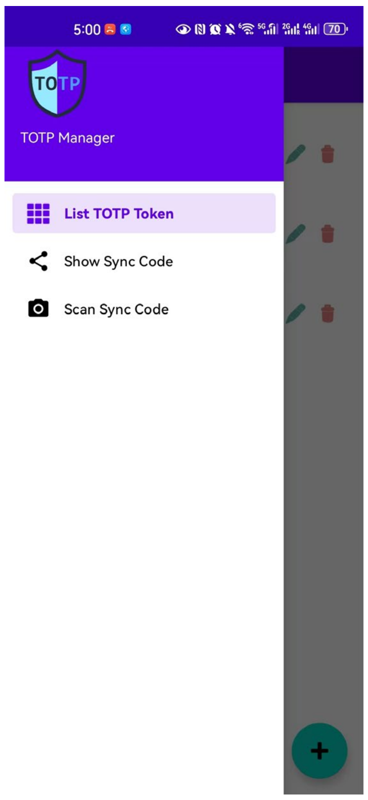
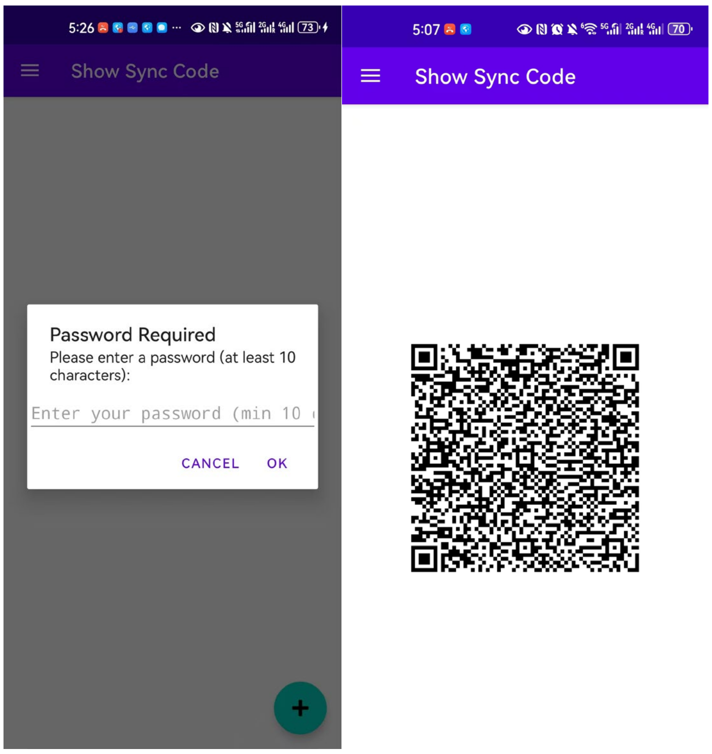


Figure 7 Dropdown Menu

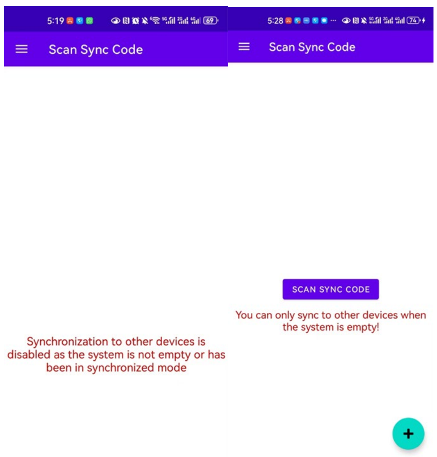
5.3 Show Sync Code: Other devices can download tokens by scanning this sync code shown by device A. The user of device A needs to input a password to generate the sync code.



Password Required
Please enter a password (at least 10 characters):
Enter your password (min 10

Figure 8 Show Sync Code Page

5.4 Scan Sync Code: "Scan Sync Code" is disabled when there already exists a sync code in the device. "Scan Sync Code" is enable when there does not exist a sync code in the device.



Synchronization to other devices is disabled as the system is not empty or has been in synchronized mode

You can only sync to other devices when the system is empty!

Figure 9 Scan Sync Code Page

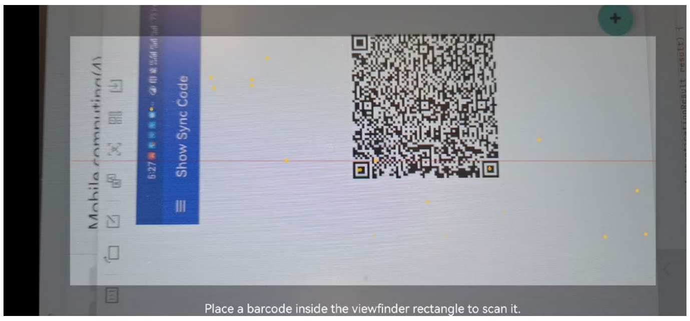


Figure 10 Scanning Sync Code

5.5 Scan Website's QR (Create Token Entry): Users can scan website's QR code to obtain a new token.

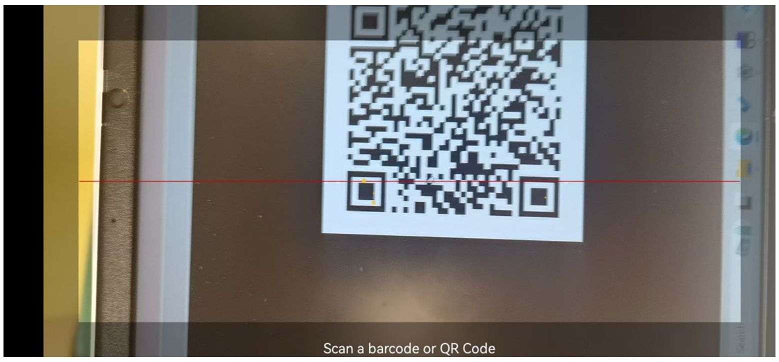


Figure 11 Scanning Websites' QR Code

5.6 Edit Token Name: After getting tokens from website, users can specify the token name.

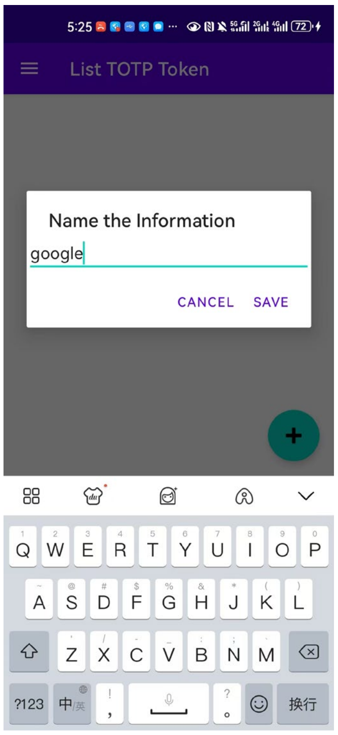
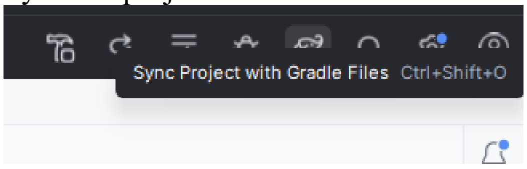


Figure 12 Edit Token Name

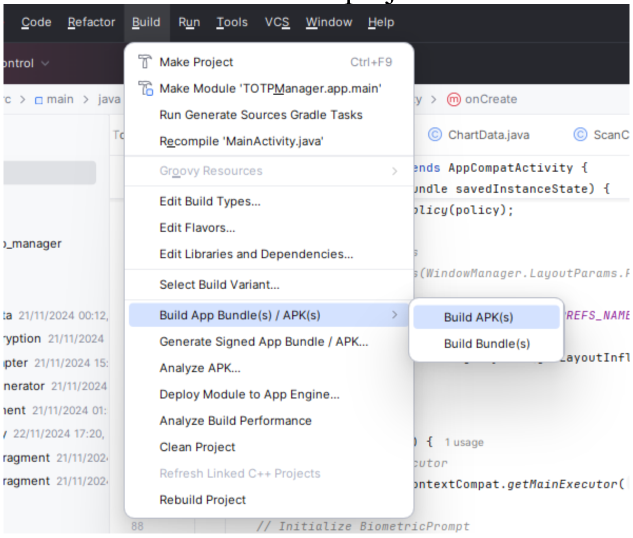
6. INSTALLATION GUIDANCE

Android side

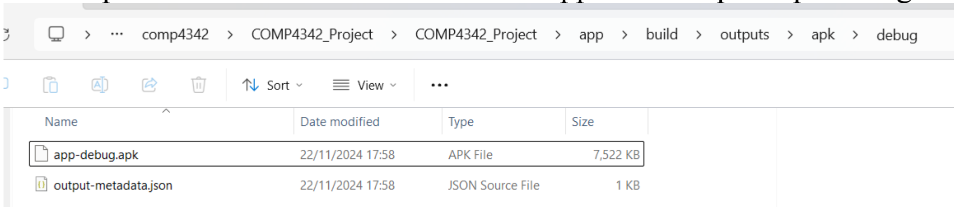
- Open the source code in Android Studio.
- Sync the project with Gradle files.



- Build the APK file for the project.

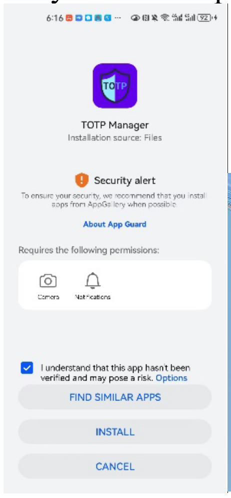
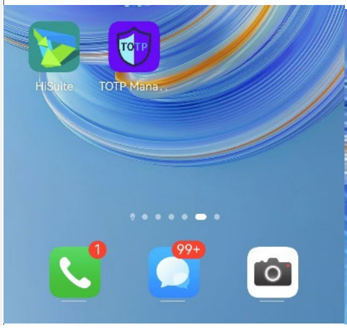

Code
Refactor
Build
Run
Tools
VCS
Window
Help
Make Project Ctrl+F9
Make Module 'TOTPManager.app.main'
Run Generate Sources Gradle Tasks
Recompile 'MainActivity.java'
Groovy Resources
Edit Build Types...
Edit Flavors...
Edit Libraries and Dependencies...
Select Build Variant...
Build App Bundle(s) / APK(s)
Generate Signed App Bundle / APK...
Analyze APK...
Deploy Module to App Engine...
Analyze Build Performance
Clean Project
Refresh Linked C++ Projects
Rebuild Project
Build APK(s)
Build Bundle(s)
onCreate
ChartData.java

- The output APK file should be located at \app\build\outputs\apk\debug.


comp4342 > COMP4342_Project > COMP4342_Project > app > build > outputs > apk > debug
Sort
View

Name	Date modified	Type	Size
app-debug.apk	22/11/2024 17:58	APK File	7,522 KB
output-metadata.json	22/11/2024 17:58	JSON Source File	1 KB

- Transfer the APK file to your mobile phone via wired or wireless data transfer.
- On your mobile phone, install the APK file.

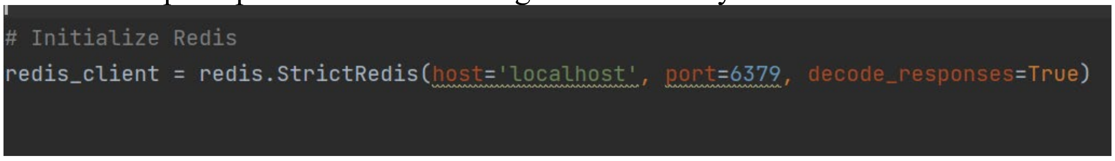

TOTP Manager
Installation source: Files
Security alert
To ensure your security, we recommend that you install apps from AppGallery when possible
About App Guard
Requires the following permissions:
Camera
Notifications
I understand that this app hasn't been verified and may pose a risk. Options
FIND SIMILAR APPS
INSTALL
CANCEL


- Launch the software by clicking the icon on your screen.

Server side

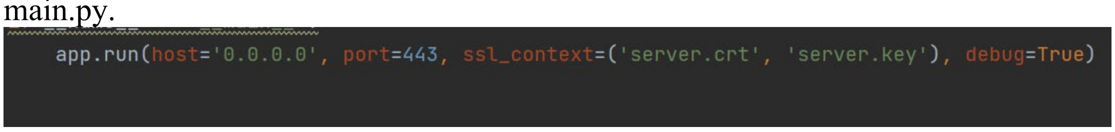
- Install dependencies: Install dependencies:
  pip install flask
  pip install redis

- Please ensure Redis is prepared, then start Redis to ensure that the port used by Redis matches the port specified for initializing Redis in the Python server.



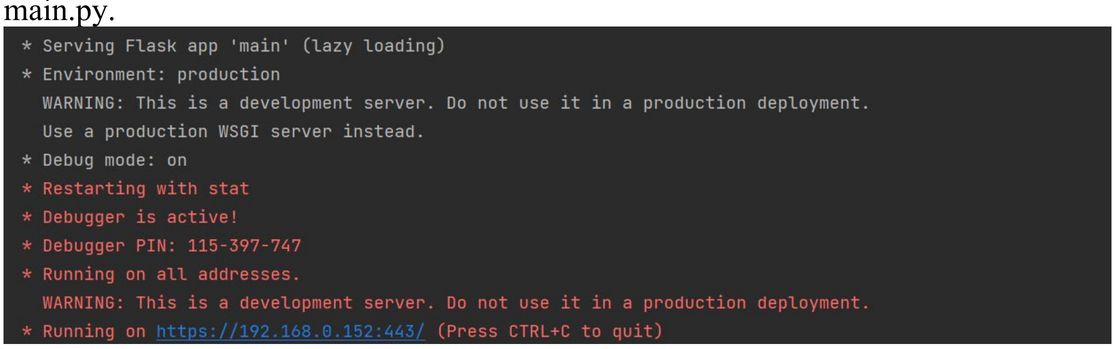
```
# Initialize Redis
redis_client = redis.StrictRedis(host='localhost', port=6379, decode_responses=True)
```

- Please create the certificate and key yourself, naming them server.crt and server.key. Ensure that the SAN value of the certificate matches the host where the Python application is running. The certificate and key should be in the same directory as main.py.



```
    app.run(host='0.0.0.0', port=443, ssl_context=('server.crt', 'server.key'), debug=True)
```

- After completing the above steps, you can start the Python server by running python main.py.



```
 * Serving Flask app 'main' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 115-397-747
 * Running on all addresses.
   WARNING: This is a development server. Do not use it in a production deployment.
 * Running on https://192.168.0.152:443/ (Press CTRL+C to quit)
```

7. REFERENCES

[1]T. Hagos, *Android Studio IDE quick reference : a pocket guide to Android Studio development.* California: Apress, 2019.

[2]M. Khatib and N. Salman, *Mobile Computing*. IntechOpen, 2018.

[3]V. A. Cunha, D. Corujo, J. P. Barraca, and R. L. Aguiar, “TOTP Moving Target Defense for sensitive network services,” *Pervasive and mobile computing*, vol. 74, pp. 101412-, 2021, doi: 10.1016/j.pmcj.2021.101412

[4]S. Furnell, *Mobile security*, 1st edition. Ely, U.K: IT Governance Pub., 2009.

8. PEER REVIEW

Chenxi Liu: 2.2.2 Mobile App Development, 6 Installation Guidance
Haolin Zhang: Design the Overall System, Project Manager, Report Audit, Quality Control
Tianji Huang: 2.2.1 Secure Data Synchronization, 3 Programming Languages and Tools Used
Le Xu: 2.1 Synchronization Mechanisms, 5 User Manual

	Chenxi Liu	Haolin Zhang	Tianji Huang	Xu Le
Percentage(%)	25	25	25	25