Univerzitet u Sarajevu

Elektrotehnički fakultet u Sarajevu

Odsjek za računarstvo i informatiku

Secure Remote Control

Softverski inženjering

2025

Grupa 1 - Client side app (Android)
Grupa 2 - Web-based admin panel
Grupa 3 - Communication layer

1. Target Environment

Web-based admin panel:

The system is designed to run in a Linux environment, with recommended distributions such as Ubuntu Server 20.04 LTS or later due to their stability, support, and compatibility with Node.js, Docker, and other modern development tools. While local development can be done on Windows or macOS, production deployment should be on Linux for better performance and security.

Minimum hardware specifications for running the backend, frontend, and communication layer are:

- **CPU:** Quad-core (x86_64 or ARM64 if supported)
- **RAM:** 8 GB (to support 1000+ concurrent sessions)
- **Storage:** SSD with at least 20 GB of free space (for logs, database, session cache)
- **GPU:** Not required (the system does not perform graphics-intensive tasks)

The system is cloud-ready. Current production deployment uses:

- **Render** for frontend hosting (React, served as static files)
- **Render** for backend services (Node.js API + WebSocket server)

Client side (Android):

Operating System:

- Minimum: Android 8 (API 26) – required for WebSocket and AccessibilityService
- Recommended: Android 12+ (API 31+) – better support for screen capture and WebRTC stability

Hardware Requirements:

- Processor: Quad-core recommended
- RAM: 4GB+
- Storage: 50MB

- Screen: Responsive to all screen sizes

The application itself is not a cloud-based application, as it runs locally on the Android device. However, it communicates with a cloud-hosted communication layer via secure WebSocket (WSS) connections to exchange data and control commands.

Server side (Communication Layer):

Operating System:

- **Linux (Ubuntu server) -** Node.js and MongoDB support; widely supported by cloud providers

Hardware Requirements:

- Processor: **2-4 CPU cores** recommended
- RAM: **4+ GB RAM**
- Storage: **20 GB SSD** minimum, **50+ GB SSD** recommended (OS, application files and dependencies, logs, temporary files and caches)
- Network requirements: **10 Mbps** Upload/Download Bandwidth

Cloud Deployment (cloud providers e.g. Railway) ; CI/CD pipeline (Railway + Git)

2. Software Dependencies

Web-based admin panel:

Runtime Dependencies:

- **Node.js v18.x or newer** (for backend)
- **npm v8.x or newer**
- **Docker** (for optional containerized deployment and CI/CD)

Frontend – Important Dependencies:

- `react / react-dom` – Core React libraries for UI rendering.
- `react-router-dom` – Navigation between pages (routing).

- `axios` – HTTP client for communicating with the backend API.
- `jwt-decode` – Decoding JWT tokens for authentication/authorization.
- `tailwindcss / @tailwindcss/vite` – Utility-first CSS framework and integration with Vite.
- `lucide-react / react-icons` – Icon libraries for user interface elements.

Backend – Important Dependencies:

- `express` – Main web framework for creating API routes.
- `cors` – Enables cross-origin requests (important for frontend-backend communication).
- `dotenv` – Loads configuration from `.env` files.
- `jsonwebtoken` – For generating and verifying JWT tokens used in authentication.
- `bcrypt` – Hashing passwords for secure storage of user credentials.
- `mongodb` – Direct communication with the MongoDB database.
- `socket.io-client / ws` – WebSocket communication for real-time features.

Libraries, Packages, Frameworks:

- **Backend:** Express.js, ws (WebSocket library for real-time communication), JWT (authentication), dotenv (enviroment variables), bcrypt (password hashing), cors (CORS middleware)
- **Frontend:** React.js (TypeScript), Vite, Axios, TailwindCSS
- **Database:** MongoDB (or a compatible NoSQL solution)

Containerization:

Docker support is available for both frontend and backend.

- Backend: Use node:18-alpine
- Frontend: Use node:18-slim for build phase
  Versioned Docker images should be defined in appropriate **Dockerfiles**.

Client side (Android)

Runtime Dependencies:

- Android Runtime (ART)
- WebRTC Native Libraries (bundled)
- OkHttp WebSocket client (bundled)

Development Dependencies:

- Kotlin Coroutines
- Jetpack Compose
- Dagger Hilt
- Retrofit

Permissions Required:

- INTERNET
- FOREGROUND_SERVICE
- POST_NOTIFICATIONS
- ACCESSIBILITY_SERVICE
- MEDIA_PROJECTION

Server side (Communication Layer):

Runtime Dependencies:

- Node.js

Libraries, packages and frameworks:

- **Express** - Web server framework (HTTP requests, routing, middleware)
- **ws** - WebSocket library (real-time, full-duplex communication between clients and server)
- **jsonwebtoken** - Library (creating and verifying JWT tokens; authentication)
- **cors** — Middleware (Cross-Origin Resource Sharing)
- **Dotenv** - package (configures app without hardcoding secret)

Containerization:

- **Docker** is only used for local development and testing. Deployment is done on a cloud platform directly from code, without Docker containers in production.

3. Installation and Configuration

Client side (Android)

3.1.0 How is the software installed?

Clone the main or develop branch from the GitHub repository, then build the project using Android Studio and Gradle, with ProGuard optimization and release signing configuration enabled. The result will be a signed .apk file, ready for distribution.

3.2.0 Steps to Initialize the System

After installation, the app must be configured manually on the Android device. The user needs to enable notifications and the Accessibility Service, grant screen capture and remote control permission, and enter the registration code provided by the Web Administrator.

3.3.0 Environment variables or config files required:

- WebSocket URL: wss://remote-control-gateway-production.up.railway.app/ - predefined in code
- Device ID: Settings.Secure.ANDROID_ID
- Preferences stored in SharedPreferences or DataStore

3.4.0 Default users and passwords:

There are no default users in the client-side application. Registration is performed by entering a unique registration code provided by the Web Administrator.

Server side (Communication Layer):

3.1.1 How is the Software Installed?

- Manual installation via git clone and execution of npm scripts (npm install / npm start)

- Alternatively: Deployed through **Docker containers** or **CI/CD pipelines**

3.2.1 Environment Variables or Config Files

Project **requires a .env file** with environment variables for configuration. This includes:

- **PORT**: Server listening port
- **DB_URI**: MongoDB Atlas connection URI
- **DB_URI_LOCAL**: Local MongoDB URI (for Docker/local testing)
- **USE_LOCAL_DB**: If *true*, use local DB; else, use Atlas
- **JWT_SECRET**: Secret key for signing/verifying JWTs

3.3.1 Steps to Initialize the System

1. Clone the repository
2. Install dependencies (npm install)
3. Create a .env file and configure environment variables (DB_URI, PORT, etc.)
4. Ensure MongoDB Atlas is accessible (no need to set up schema manually-collections will be created on first use)

3.4.1 Default Users and Passwords

Database Access (MongoDB Atlas)
Default access credentials (*can be changed if needed):*

- Username: root
- Password: root

These are default settings that can be configured according to the security requirements of the production environment.

Web-based admin panel

3.1.2 How is the Software Installed?

- The Secure Remote Control Web App consists of a Node.js backend and a React (TypeScript) frontend.

- Installation involves cloning the repository and installing dependencies for both backend and frontend using standard Node.js package management (`npm install`).

- The backend is started with `npm start`, and the frontend is started with `npm run dev`.

3.2.2 Environment Variables or Config Files

- Configuration is managed through environment variables defined in `.env` files or set directly in the deployment environment.

- The backend requires environment variables such as:
  - `PORT` (server listening port)
  - `DB_URI` (MongoDB connection string)
  - `DB_URI_LOCAL` (Local MongoDB connection string for Docker/local testing)
  - `USE_LOCAL_DB` (If true using local DB else not)
  - `SECRET_KEY` (secret key for signing authentication tokens)

- The frontend requires:
  - `VITE_BASE_URL` (URL of the backend API to correctly route requests)
  - `VITE_WS_URL` (URL of the backend WS for real-time communication)
  - `VITE_API_UPLOAD_URL` (URL of API of the communication layer`s API for file uploads)

3.3.2 Steps to Initialize the System

- Ensure a MongoDB database is set up and accessible, commonly via MongoDB Atlas.

- Set the `DB_URI` environment variable with the database connection string.

- Configure a secure `SECRET_KEY` key for user authentication.
- Install dependencies for both backend and frontend (`npm install`).
- Start the backend server first, allowing it to connect to the database and prepare necessary collections.
- Start the frontend server, configured to communicate with the backend API.
- Additional API keys or external service credentials should also be configured as environment variables before startup.

3.4.2 Default Users and Passwords

- The application comes with default admin credentials for initial access:
    - Username: `administrator`
    - Password: `12345678`
- These are default credentials and this is the main account. Other admins can be created as needed just from this account.

4. Continuous Integration / Continuous Deployment (CI/CD)

CI/CD tools used (Communication layer):

- Railway + GitHub integration
- CI/CD Platform: Railway

CI/CD tools used (Web-based admin panel):

- Render + GitHub integration
- CI/CD Platform: Render (it is automated)

Client side (Android)

CI/CD not automated. Build and deployment are manual via Android Studio

What Triggers Deployment?

- **Push to the develop branch** triggers automatic deployment for both frontend and backends

- **Render and Railway** use a **webhook** to trigger redeployment upon each commit

5. Network and Security

Client side (Android)

Ports That Need to Be Open

- 443 (WSS)
- Dynamic ports for WebRTC (UDP/TCP)

SSL/TLS Requirements:

- WSS encryption for signaling
- DTLS-SRTP for WebRTC
- Registration key validation
- Session token authentication

Authentication Mechanisms

The application does not perform any kind of authorization itself; device registration and deregistration are handled exclusively through the admin panel over a secure connection.

Registration Flow:

- When the app is launched for the first time, it generates a unique device ID.
- The device registers with the backend server over a secure WebSocket connection (wss://).
- The registration status and device ID are stored locally using SharedPreferences.
- On subsequent launches, the app uses the stored device ID to authenticate with the backend.

- No username or password is required for authentication; device identity is used instead.

Server side (Communication Layer):

Ports That Need to Be Open

- **Port 8080** - HTTP (Express) and WebSocket connections

SSL/TLS Requirements

The deployment platform (e.g., Railway) provides built-in SSL/TLS support, automatically handling **HTTPS and secure WebSocket (WSS) connections**. This includes certificate issuance, renewal, and HTTP-to-HTTPS redirection. As a result, the application does not need to implement SSL/TLS handling directly in its code—Railway's infrastructure manages it transparently.

Authentication Mechanisms

JWT (JSON Web Token) is used as the primary authentication mechanism.

- jsonwebtoken package (^9.0.2) issues and verifies tokens.

The server validates the token on each request to ensure that the user is legitimate.

Web-based admin panel:

Ports That Need to Be Open

- **Frontend:** 5173 (local), 443 (HTTPS)
- **Backend API:** 5000 (local), 443 (HTTPS)
- **WebSocket:** Shares the same port as backend, ensure wss:// is enabled

SSL/TLS Requirements

- All communication must use **TLS 1.2 or higher**
- TLS is mandatory for WebSocket connections (wss://)
- The used cloud deployment platforms redirect traffic to HTTPS out of the box
- On-premise: **Let's Encrypt** with **Nginx** are used as a reverse proxy

Authentication Mechanisms

- **Username & password with JWT** for admin panel access
- **JWT** for session control and API access
- **End-to-end encrypted (E2EE)** communication between Android client and backend

6. Database Deployment

Web-based admin panel

DBMS Required

Production data is stored in Mongo cluster on MongoDB Atlas free tier.

- **MongoDB**, either locally or in the cloud (MongoDB Atlas recommended for scalability)

Initialization Scripts or Migrations

- Collections are created dynamically on first use
- Optionally, you can provide seed scripts to preload configuration data

Hosting Supported

- **Local hosting** via Docker or MongoDB installation
- **Cloud hosting** via MongoDB Atlas (preferred)

Client side (Android)

DBMS Required

- There is no database used on the Android client side. However, local storage mechanisms are used to persist essential app state and session information.

Local Storage:

- SharedPreferences

    - Used for storing lightweight persistent data such as the device registration code, registration status, and user preferences. This is a key-value based storage provided by Android, ideal for simple configuration data.
- WebRTC memory buffers
    - Media streams (video/audio) captured from the screen are buffered temporarily in RAM during live sessions. These buffers are volatile and discarded once the session ends.

Server side (Communication Layer):

DBMS Required

- This project uses MongoDB as the database system.

Initialization scripts or migrations

- Initialization scripts or migrations exist in the migrations folder and are used for local development or testing (with Docker-based MongoDB) and production database.

Hosting supported

- Cloud-hosted MongoDB via MongoDB Atlas

7. Rollback and Recovery

Backup procedures

- Since the database is hosted on **MongoDB Atlas (managed cloud service)**, backups are managed by MongoDB Atlas automatically. Atlas provides **continuous backups** and **point-in-time recovery** features, ensuring data can be restored to any point within the backup retention window. Backup schedules and retention policies are configurable via the MongoDB Atlas dashboard.

Client side (Android)

- There is no built-in rollback mechanism on the Android app itself. However, recovery and reinitialization are simple and effective due to the stateless nature of the app on the client side.

Recovery procedures:

- **App can be uninstalled/reinstalled**
  If the app behaves unexpectedly, a full reinstall ensures a clean start and resets all locally stored data.

- **Clear app data to reset state**
  Users can also go to system settings → App info → Storage and clear app data to reset the app to its initial state without reinstalling.

- **Deregister device from server via API**
  If needed, the device can be deregistered through an API call allowing for a fresh registration.

Version control & rollback:

- **GitHub repository retains stable versions**
  Only stable, tested versions are pushed to the main branch or marked as releases. If an issue arises after deployment, reverting to a previous stable version is as simple as rebuilding an older commit from GitHub.

Web-based admin panel:

Deployment rollback

- Render has a built-in rollback button, which makes reverting to a previous deployment straightforward. To roll back, go to the Render Dashboard, select your web service, navigate to the **Deploys** section, and click the **Rollback** button next to a previously stable deployment. This will instantly redeploy that version. Alternatively, you can perform a rollback via Git by using git revert <bad_commit_hash> or resetting to a known good commit with git reset --hard <good_commit_hash>, followed by a push to the tracked branch develop. Render will then automatically deploy the updated version. These methods allow for efficient recovery in case the latest deployment causes issues.

Server side (Communication Layer)

Deployment rollback

- Railway automatically deploys the latest commit on every push to the connected repository branch.To rollback, you can redeploy a previous stable commit by selecting it

in Railway's deployment history and triggering a redeploy. To perform a rollback, click the three dots at the end of a previous deployment, you will then be asked to confirm your rollback. This allows quick rollback to an earlier version if the latest deployment has issues.

8. Monitoring and Logging

Server side (Communication layer & Web-based admin panel):

Tools Used:

- **Railway** and **Render** provide built-in basic logging and monitoring
- Can integrate with external tools like **Prometheus**, **Grafana**, **Loggly**, or **ELK Stack**

Logs Generated and Their Location:

- **Backend:** Console output (console.log, console.error, console.warn)
- **Frontend:** Browser console logs
- Logs can be redirected to files or external logging services

Client side (Android)

- Logging is handled using Android's built-in Logcat system. Logs for WebSocket connections, WebRTC session status, and AccessibilityService activity are output using Logcat. These logs can only be viewed when the app is launched and monitored through Android Studio during development or testing.

9. User Access and Roles

9.1 Who Can Access the Deployed System?

- **IT support agents** and **administrators** can access the admin web interface

- **End-users (Android clients)** connect automatically when support is requested

9.2 How Are Users Provisioned and Managed?

- **OAuth2 (e.g., Google Workspace, Azure AD)** is used for user provisioning
- Admins have access to a dashboard for session monitoring and audits

10. Testing in Deployment Environment

Client side (Android):

Smoke testing and post-deployment checks

Smoke Testing (Basic Sanity Check)

Smoke testing serves as a quick verification to determine whether the app's critical functions work correctly. It is performed immediately after deployment and before any deeper testing is conducted.

The smoke test includes:

- **App Launch:** Verify that the application installs successfully and opens without crashing.
- **Permissions:** Confirm that the required permissions are granted (INTERNET, ACCESSIBILITY_SERVICE, MEDIA_PROJECTION, etc.).
- **WebSocket Connection:** Ensure the app establishes a secure WSS connection to the backend.
- **Screen Capture:** Check that screen sharing starts correctly after granting MediaProjection permission.
- **Accessibility Service:** Enable and verify that it functions as expected.

If any of these steps fail, the build is considered unstable and not ready for further use or testing. Once the smoke test passes, perform the following additional checks to validate deeper functionality and integration:

- **WebSocket Connection Stability**
  Monitor whether the WSS connection remains active over extended sessions (e.g., 15–30 minutes). The client should automatically attempt reconnection if interrupted.

- **Screen Sharing Consistency**
  Confirm that the screen stream remains uninterrupted and synchronized, especially when navigating between apps, rotating the screen, or minimizing the app.

- **Remote Control Responsiveness**
  Verify that remote input actions (e.g., typing, swiping) are consistently received and executed on the target device without delays or failures.

- **Permission Retention**
  Check that the granted permissions (MediaProjection, AccessibilityService) remain active across device restarts, sleep/wake cycles, and do not require re-authorization unless explicitly revoked by the user.

Server side (Communication Layer):

Post-Deployment Smoke Test

1. **Server Startup**
   Confirm server starts on PORT=8080 without errors (check Railway logs).
2. **DB Connection**
   Ensure connection to MongoDB Atlas (DB_URI) is successful (look for "Connected to MongoDB").
3. **WebSocket Test**
   Connect via WebSocket, send/receive a test message.
4. **Device Registration**
   Simulate heartbeat from a test device; check it's added and marked active.
5. **JWT Auth Check**
   Use valid JWT to access a protected route and confirm it works.
6. **Web Admin Test** *(if available)*
   Open interface and test session approval/denial.
7. **Log Review**
   Monitor Railway logs for errors or unexpected behavior.

Web-based admin panel

Smoke Testing or Sanity Check Procedures

After deploying the Secure Remote Control Web App to the production environment, it is essential to conduct smoke testing or sanity checks to ensure that the core functionalities of the system are operational. This initial post-deployment testing verifies that the application has been deployed correctly and that there are no major issues preventing its basic use. The following procedures are recommended:

- Confirm that both the frontend and backend services are reachable via their public URLs.
- Access the frontend interface and verify that the application loads without any critical errors in the browser console or network tab.
- Attempt to authenticate using valid admin credentials and ensure that the login process successfully retrieves and stores a JWT token (if applicable).
- Navigate through core sections of the admin panel (e.g., device list, session logs) and confirm that data is loaded via API calls without failure. Initiate key backend operations such as fetching device data, establishing remote control session logs, or interacting with WebSocket features to validate server-side functionality.
- Additionally, inspect Render's deployment logs for each service to identify any startup errors, and monitor real-time logs to detect uncaught exceptions or warnings.

Smoke testing should always be performed after every major deployment or configuration change to catch regressions early and ensure the platform is in a stable, usable state before exposing it to end users or clients.

After deployment, perform the following:

- Verify frontend loads correctly (React UI)
- Start a test session with an Android device
- Inspect logs for errors
- Ensure successful WebSocket handshake (wss://)

11. Step-by-Step Deployment to Blank Environment

Web-based admin panel

The repository consists of two main directories: `backend` for the Node.js API server and `frontend` for the React-based admin dashboard.

Requirements:

- A Render account
- GitHub repository admin access
- Necessary environment variables such as the MongoDB connection URI and JWT secret

Deploying the Backend (Node.js) on Render

1. Log in to your Render dashboard and create a new Web Service.
2. Connect your GitHub repository and select the appropriate branch (`develop`).
3. Set the name of the service to something like `secure-remote-backend`.
4. Set the name of root directory to backend.
5. In the build command field, enter: `npm install`.
6. In the start command field, enter: `npm start`.
7. Enable auto-deploy on every commit or after CI checks pass (if you want)
8. Set the environment to Node.
9. Add the required environment variables under the "Environment" tab. Example variables include:
    - `PORT`
    - `DB_URI`
    - `SECRET_KEY`
10. Save and deploy the service.

Deploying the Frontend (React) on Render

1. Create another new Web static site in your Render dashboard.
2. Connect the same GitHub repository and select the same or appropriate branch.
3. Set the root directory to frontend.
4. Set the name of the service to something like `secure-remote-frontend`.
5. In the build command field, enter: `npm install && npm run build`.
6. Set the name of publish directory to: dist.
7. Enable auto-deploy if you want.

8. Add any required public environment variables, such as:
    - `VITE_BASE_URL`
    - `VITE_WS_URL`
    - `VITE_API_UPLOAD_URL`
9. Save and deploy the service.

Connecting the Frontend to the Backend

Ensure that the frontend is configured to communicate with the backend by setting the `VITE_BASE_URL` environment variable to the public URL of the backend service. This can be done either in an `.env` file locally or directly in the Render service's environment settings. After setting the correct API base URL, redeploy the frontend service to apply the changes.

Verifying the Deployment

Once both services are deployed, test the full application by visiting the public URL of the frontend service. Ensure that all features, including authentication and device communication, are functioning correctly. You can monitor logs for both services through the Render dashboard to diagnose any runtime issues. It is recommended to configure both services for automatic redeployment upon GitHub push. Avoid hardcoding sensitive data by using environment variables, and for production setups consider adding custom domains, enabling HTTPS, and reviewing access permissions.

Client side (Android)

1. Environment Setup

Before building the application, prepare your local development environment:

- Install Android Studio
  Download and install Android Studio (latest stable version recommended). Android Studio provides the necessary tools to build and test the app.
- Install Java JDK 17
  Ensure Java Development Kit (JDK) version 17 is installed. It is required for Gradle builds and Kotlin compilation.
- Install Android SDK
  Through Android Studio's SDK Manager, install the required SDK platforms and build tools (targeting API 31+ for optimal support).

- Set ANDROID_HOME environment variable
  Define the Android SDK path to let Gradle and command-line tools locate the SDK.

2. Project Setup

Clone the official project repository and build the app using Gradle:

- Clone the GitHub repository
  git clone https://github.com/SI-SecureRemoteControl/Client-Side-Android-app.git
- Build the project
  Use the Gradle wrapper to clean and build the project
- Generate .apk through Android Studio

3. Device Setup

Application can be installed on devices in multiple different ways e.g. installing app through WiFi debugging/USB debugging or the .apk file can be delivered through a Google Drive link or uploaded to platforms intended for app distribution and installation.

After installing the application (via APK transfer or other method), the user must follow these steps to properly set up the device for remote control:

1. **Enable Permissions**
   Upon first launch, the application will prompt the user to enable essential permissions:
   - Accessibility Service – Required for enabling remote control functionalities
   - Notification Access – Allows the app to access and forward notification content
     These permissions must be granted manually in system settings when prompted.
2. **Device Registration**
   After granting the required permissions, the user must register the device.
   - The app will request a registration code, which the user obtains from the web admin panel or through a separate communication channel (e.g., email or chat).
   - Once the code is entered, the device will be linked to the admin system and marked as active.
3. **Request Remote Session**
   Once registration is successful and all permissions are granted, the user can request a remote session through the app.

4. Remote Control Activation
   When the session request is approved by the web admin, the admin gains remote control access to the device.

Server side (Communication Layer):

Step-by-Step Deployment on Cloud (Railway)

1. Sign up or log in at railway.app.
2. Click New Project and select Deploy from GitHub repo.
3. Connect GitHub Repository; authorize Railway to access your GitHub account and select the repository containing your backend code.
4. In Railway's project dashboard, go to Settings > Environment Variables, and add your required variables like:
   - `PORT=8080`
   - `DB_URI`
   - `JWT_SECRET`
5. Configure Build and Start Commands (usually automatic)
   - Build command: `npm install`
   - Start command: `npm start`
6. Deploy- Railway automatically clones the repo, installs dependencies, builds, and starts your server on every push.
7. Railway provides a public URL where your backend API is accessible.

User guide for running the *Secure Remote Control* application

NOTE: For the instructions on running your local database, please see the accompanying document **at the end**.

We **highly recommend** reading through **the entire** document **carefully** before running the application.

For running the application locally, first start web admin with instructions under segment Group 2.

After that, start the Communication layer app with steps under Group 3.

After that is done, you can start the Android app with instructions under Group 1.

Keep in mind that if you want to start locally, both web admin and communication layer need to use the same database source (local or remote).

Also, when running projects locally, Web admin and communication layer support localhost resolving, but android app requires full ip address when connecting to communication layer.

You can find the local ip address of the machine where you start the web admin and communication layer by using the **ifconfig/ipconfig** command in your terminal.

Group 1 - Android

1. Install Android Studio Flamingo

Go to the official Android Studio website (https://developer.android.com/studio) and download the version called "Flamingo".
Follow the installation steps for your operating system (Windows, macOS, or Linux).

2. Clone the Client Side Android App Repository

Go to the following link:
https://github.com/SI-SecureRemoteControl
Open Client-Side-Android-app
Open Android Studio
Click on "Get from VCS" (Version Control System)
Paste the repository link and click "Clone"
After cloning, right-click on the project folder in the Project view and choose "Git → Pull" to make sure you have the latest code

3. Sync the Gradle

Gradle is the tool that builds your project.

When the project opens, a bar will appear at the top saying "Sync Now" – click it
If not, go to the menu and click "File → Sync Project with Gradle Files"
Wait for it to finish downloading everything it needs

4. Connect Your Phone or Build an APK

You can either connect your Android phone with a USB cable (make sure Developer Mode and USB Debugging are turned on), or
Go to "Build → Build Bundle(s) / APK(s) → Build APK(s)" to create a file you can install manually on your phone.

5. Run the Application

If your phone is connected, click the green "Play" button in Android Studio.
If you created an APK, install it on your phone and open the app manually.

6. Enter Server Address in the App

When the app opens, you'll see a field asking for a server address
Type:

ws://your_computer_ip_address:8080

For example, if your computer's IP address is **192.168.1.5**, you would enter:
ws://192.168.1.5:8080

Group 2 - Web app

1. Clone the Web App Repository

Go to the following link:
https://github.com/SI-SecureRemoteControl/web-app

Open a terminal (or Git Bash), navigate to a folder where you want the project to be and run the following commands:

```
git clone https://github.com/SI-SecureRemoteControl/web-app.git
cd [project-folder]
git pull
```

2. Install Node.js

Download Node.js from https://nodejs.org and install it if you don't have it
Make sure it includes npm (Node Package Manager)

3. Add the .env Files

There are **two** `.env` files to create:

- A backend `.env` file (inside the `backend` folder)
- A frontend-specific `.env` file (inside the `frontend` folder)

Do the following:

For the **backend** .env, create a new .env file with no name, inside project (web-app folder).

In the file, paste the following:

PORT=9000
DB_URI='mongodb+srv://root:root@cluster.qciyr2x.mongodb.net/Cluster?retry Writes=true&w=majo rity&appName=Cluster'
DB_URI_LOCAL='mongodb://localhost:27017/'
USE_LOCAL_DB=true
SECRET_KEY='10429b28ab5cb5042e393b2f20d76bb1'

A secret key can be **anything** you want, this is just an **example**.

Do not be confused with two database URIs, the local URI is used by developers when testing and adding new features. If you want to use a local database you can run docker compose which will pull Mongo images and start up a Mongo instance in your local docker container.

Brief explanations for each field in the .env file for backend:

1. PORT: Specifies the port on which the server should listen.
2. DB_URI: This is the connection string for a cloud-hosted database, such as MongoDB.
3. DB_URI_LOCAL: Connection string for a **locally hosted NoSQL database instance** (e.g. MongoDB), such as one running in Docker or directly installed.
4. USE_LOCAL_DB: A flag that determines **which database URI** to use — the **local** one or the **cloud** one.
5. SECRET_KEY: Used to sign and verify JWT tokens for authentication

For the **frontend** .env, create a new .env file with no name, inside frontend folder of the repository

In the file, paste the following:

VITE_BASE_URL=http://localhost:9000
VITE_WS_URL=ws://localhost:9000
VITE_API_UPLOAD_URL=http://localhost:5000/api/upload
VITE_COMM_LAYER_API_URL=http://localhost:5000

Brief explanations for each field in .env:

1. **VITE_BASE_URL**: base URL for the backend API or server during local development
2. **VITE_WS_URL**: WebSocket URL of backend for real-time communication.
3. **VITE_API_UPLOAD_URL**: This is the full URL to the upload API endpoint, which points to the local instance of the communication layer.
4. **VITE_COMM_LAYER_API_URL**: This is the full URL to the communication layer

6. Start the Backend

Open a terminal or command prompt
Navigate to the backend folder (usually the root folder of the project)

Run the following commands:

npm install
npm start

7. Start the Frontend

Open a new terminal or command prompt
Navigate to the `frontend` folder inside the project

Run the following commands:

npm install
npm run dev

8. Access the App

Once both backend and frontend are running, open your browser and go to the address shown in the terminal

Group 3 - Communication Layer

The Secure Remote Control Gateway is a Node.js-based server that acts as a communication layer between IT admins and Android devices. It facilitates secure WebSocket-based real-time interactions, including device registration, WebRTC signaling, and remote control commands.

1. ***Ensure you have node.js installed on your machine:***

   node -v
   npm -v

2. ***Clone the secure-remote-control-gateway repository:***

   git clone
https://github.com/SI-SecureRemoteControl/secure-remote-control-gateway.git

   cd secure-remote-control-gateway
   git pull

3. ***Install dependencies:***

   npm install

4. *Create a .env file and add the following fields:*

```
PORT=8080
DB_URI='mongodb+srv://root:root@cluster.qciyr2x.mongodb.net/Cluster?retryWrites=true&w=majority&appName=Cluster'
# Lokalni MongoDB (Docker)
DB_URI_LOCAL='mongodb://localhost:27017/'
USE_LOCAL_DB=false
JWT_SECRET='23181ac6f726f8d199b4c6732de22cc8b1869102118b74eab4dd8fbb7d18fc492fee2016f3f483ce369a3c0bfa9228daa8d0926865d8505d2607de6253930596'
WEBSOCKET_URL='wss://backend-wf7e.onrender.com/ws/control/comm'
SERVICE_URL='https://remote-control-gateway-production.up.railway.app'
```

Field explanations:

1. **PORT=8080**
   Specifies the port number the server or application will listen on (8080 is a common default for web servers).

2. **DB_URI**
   Connection string for a **remote MongoDB Atlas** database. It includes credentials and cluster information.

3. **DB_URI_LOCAL**
   Connection string for a **local MongoDB instance** running on your machine.

4. **USE_LOCAL_DB=false**
   Boolean flag to switch between the remote DB (false) or local DB (true).

5. **JWT_SECRET**
   A long secret key used to **sign and verify JSON Web Tokens (JWTs)** for authentication and security.

6. **WEBSOCKET_URL**
   WebSocket server URL used for **real-time bidirectional communication** between clients and the server.

7. **SERVICE_URL**
   URL of a **remote service endpoint**

5. ***Start the server:***

npm run start

Running this will trigger database migrations in migrate.js file in the Communication layer project.

Those migrations **are not necessary** in the project, rather they are used for cleaner database handling and changes history in the database scheme.

Secure Remote Control:
Local database setup

For this project, MongoDB is used as the primary database, but any other NoSQL database can be used as well.

The database is deployed on MongoDB Atlas, while for local development and testing, a Docker image is used to run the database inside a container.

To manage the database structure, the tool "mongo-migrate" is used, which allows version control and migration management.

MongoDB was chosen due to its flexibility in handling unstructured data, and because it allows multiple services to access the database without conflicts.

Since the system uses two independent services that communicate with the database, it was important to avoid complications related to concurrent access and simultaneous migrations.

MongoDB with the mongo-migrate tool enables controlled migrations, preventing issues in multi-environment access.

INSTRUCTIONS

First, make sure Docker is installed on your computer:

https://www.docker.com/products/docker-desktop

You also need to install the Mongo shell. To install it, do the following:

For Windows:
Go to the following link:
https://www.mongodb.com/try/download/shell

Choose the **msi** option under packages
After installation, add **mongosh** to your PATH

For macOS:
Open the terminal and run the following command:
***brew install mongodb/brew/*mongosh**

For Linux:

Open the terminal and follow the official MongoDB documentation for your distribution.
For Ubuntu and Debian-based systems, you can run the following commands:

curl -fsSL https://pgp.mongodb.com/server-6.0.asc | sudo gpg -o /usr/share/keyrings/mongodb-server-6.0.gpg --dearmor

echo "deb [signed-by=/usr/share/keyrings/mongodb-server-6.0.gpg] https://repo.mongodb.org/apt/ubuntu focal/mongodb-org/6.0 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-6.0.list

sudo apt update
sudo apt install -y mongodb-mongosh

This will install the MongoDB shell (`mongosh`).
After installation, you can run **mongosh** in the terminal to enter the Mongo shell.

Setup:

Clone the repository of the **communication layer**

Open a terminal (or Git Bash), navigate to a folder where you want the project to be and run the following commands:

> **git clone https://github.com/SI-SecureRemoteControl/remote-control-gateway**
> **cd [project-folder]**
> **git pull**

After cloning or pulling the repository and accessing the project code, you need the .env file we created:

PORT=8080
DB_URI='mongodb+srv://root:root@cluster.qciyr2x.mongodb.net/Cluster?retryWrites=true&w=majority&appName=Cluster'
DB_URI_LOCAL='mongodb://localhost:27017/'
USE_LOCAL_DB=true

These are the two URIs used to connect to the deployed and local databases, respectively.

Make sure that when **testing**, you are using your local database by setting **USE_LOCAL_DB=true**, so that you avoid **interfering** with the production database.

In the terminal, run the following command to install all required dependencies:

npm install

Then, to start the application:

npm start

In the project folder, there is a file called **docker-compose.yml**

Run the following command:

docker-compose up -d

This will start the Docker container.

Check if the container is running by typing the following:

docker ps

You should see a list of containers.

Look for the one named **local-mongo**

Then run the following command

docker exec -it local-mongo mongosh

This will open the interactive Mongo shell inside the locally running MongoDB container

Once inside, you will see the prompt
test>

From here, you can do all sorts of manipulations in the database - inserting, querying, updating, and more. For full reference, visit the **official MongoDB documentation**.

Some useful commands include:

To list all databases,
show dbs

To enter a database,
use <database_name> (our is SecureRemoteControl)

To view all collections within the selected database,
show collections