

# Web Surveillance System

- [Introduction](#)
- [Project Description](#)
  - [Main Components](#)
  - [Key Functionalities](#)
- [Used Technologies](#)
  - [Backend](#)
    - [Node.js](#)
    - [Express.js](#)
    - [FFmpeg](#)
    - [SQLite \(better-sqlite3\)](#)
    - [fluent-ffmpeg](#)
  - [Frontend](#)
    - [HTML](#)
    - [CSS](#)
    - [JavaScript](#)
    - [HLS.js](#)
    - [Font Awesome](#)
  - [Other](#)
    - [Git](#)
    - [GitHub](#)
    - [Visual Studio Code](#)
    - [Google Chrome](#)
    - [npm \(Node Package Manager\)](#)
    - [IP Webcam](#)
- [Development Phases](#)
  - [Application Core](#)
  - [Streaming](#)
  - [Interface](#)
  - [User Handling](#)
  - [Stream Configuration](#)
  - [Recording](#)
- [Tasks Distribution](#)
  - [Estimated time allocation](#)
- [System Architecture](#)
  - [Project Files and Directory Organization](#)
  - [Core Server Setup](#)
  - [Routes](#)
  - [Database](#)
  - [FFmpeg](#)
  - [Utilities](#)
  - [Login and Registration](#)
  - [Streams Dashboard](#)
- [Problems and Solutions](#)
  - [Streams not playing smoothly](#)
  - [FFmpeg logs handling](#)
  - [Managing FFmpeg parameters](#)
  - [Avoiding duplicate FFmpeg processes](#)

- [Codec configuration challenges](#)
- [HLS manifest request timeout](#)
- [Recording only capturing last segments](#)
- [Installation and Execution](#)
  - [Prerequisites and Dependencies](#)
  - [Application Running](#)
  - [Example Usage](#)
- [Visual Demonstration](#)
- [Conclusion](#)

## Introduction

The **Web Surveillance System** project is a **web-based application** designed to streamline the management, processing, and playback of video streams for surveillance purposes. This system allows users to effortlessly **input video stream URLs**, configure their **playback settings**, and even **record and download** specific segments of the streams.

The primary motivation behind the project is to offer a **user-friendly**, **efficient**, and **configurable** solution for video surveillance needs, leveraging modern technologies such as [FFmpeg](#) for video processing and [HLS.js](#) for stream playback. By integrating these tools with a **robust backend** and **intuitive frontend** interface, the system aims to provide a comprehensive solution for both casual and professional users.

This document provides a **detailed overview of the system**, including its functionalities, technical implementation, development process, challenges encountered, and instructions for usage, as well as visual demonstrations to ensure a complete understanding of the project's scope and capabilities.

---

## Project Description

### Main Components

- **Frontend:** a user-friendly interface for stream input, configuration, playback, and recording. It provides options to customize playback settings like resolution, codec, and bitrate.
- **Backend:** handles stream processing using FFmpeg, manages user authentication, stores stream data in a database, and serves processed video streams to the client in HLS format.
- **Database:** uses SQLite to store user information, stream metadata, and configuration settings.

### Key Functionalities

- Inputting and managing video stream URLs.
- Configuring stream playback with customizable settings.
- Playback of live streams using the HLS protocol.
- Recording and downloading specific segments of video streams.
- Secure user authentication and management.

The application is built using modern technologies such as Node.js, Express.js, FFmpeg, SQLite, and HLS.js, ensuring efficiency and reliability in handling video streams. It is designed to be lightweight, adaptable, and easy to deploy, making it suitable for many use cases.



While the project's primary use case is video surveillance, the application is versatile and can be adapted for any scenario requiring video stream management.

---

## Used Technologies

### Backend

#### Node.js

A high-performance JavaScript runtime that enables scalable and efficient server-side application development.

Used to build the core server logic and manage asynchronous tasks.

#### Express.js

A minimalist and flexible framework for building web applications with Node.js.

Used for routing, middleware integration and handling API requests.

#### FFmpeg

A versatile multimedia processing tool for encoding, decoding, and streaming video content.

Responsible for converting video streams to HLS format, recording streams, and handling custom configurations like resolution, codec and bitrate.

#### SQLite (better-sqlite3)

A lightweight, embedded SQL database system optimized for high-performance read/write operations.

Used to store user credentials, stream metadata, and configuration settings.

#### fluent-ffmpeg

A Node.js library that provides a simple interface for interacting with the FFmpeg tool.

Used to manage FFmpeg commands for processing video streams, including encoding, transcoding to HLS, and recording functionalities.

### Frontend

#### HTML

The standard markup language used to structure content on web pages.

Used to define the layout and elements of the web interface, such as the login form, the streams dashboard, and configuration modals.

#### CSS

A stylesheet language used for describing the presentation of a document written in HTML.

Used to style and layout the web interface, ensuring a responsive and visually appealing design.

#### JavaScript

A high-level programming language commonly used to create dynamic and interactive elements on web pages.

Used to power the dynamic functionalities of the application, such as form validation, real-time updates to the dashboard, session management, and seamless communication with the backend API.

#### HLS.js

A JavaScript library for streaming video using the HTTP Live Streaming (HLS) protocol.

Ensures seamless video playback on browsers that do not natively support HLS.

#### Font Awesome

Provides an extensive set of icons for enhancing the user interface.

Used on features like logout, adding streams, configuring settings, and deleting streams.

### Other

#### Git

A distributed version control system for tracking changes in code and collaborating with team members.

Used for version control to manage code changes, and maintain a clean project history.

#### GitHub

A web-based hosting service for Git repositories.

Used to host the [project repository](#), facilitate collaboration, and track progress through commits.

#### Visual Studio Code

A very powerful and widely used code editor, with endless capabilities thanks to its extensions system.

Used as the primary development environment, for writing code, testing, running the application and much more.

#### Google Chrome

A web browser with robust developer tools for inspecting and debugging web applications.

Used for testing and debugging the frontend application, including stream playback and UI responsiveness.

#### npm (Node Package Manager)

A package manager for Node.js that simplifies dependency management and script execution.

Used to install project dependencies and manage scripts for running the application.

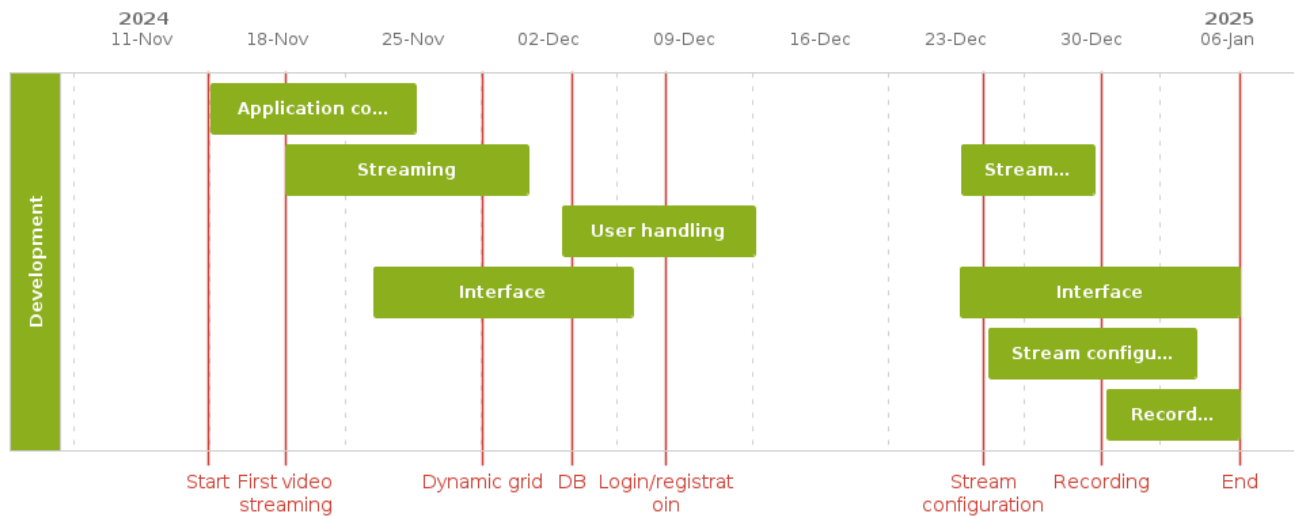
#### IP Webcam

An Android mobile app that streams the camera feed of a device as a video stream over the network.

Used to simulate and test live video streams during development.

As an additional tool, this documentation was created and organized using [Confluence](#).

## Development Phases



**⚠** This chart provides an approximation of the development timeline. The exact dates may vary slightly. For more precise information, including detailed change logs and updates, refer to the [project's version history](#) available in the repository.

### Application Core

The foundation of the project was built during this phase, establishing the basic structure of the backend and frontend. Key tasks included setting up the Node.js server, configuring Express.js routes, and creating a modular project structure to ensure scalability and maintainability.

### Streaming

This phase focused on integrating FFmpeg for video stream processing. Core functionality included converting streams into HLS format, managing real-time video playback, and ensuring compatibility with HLS.js on the frontend.

### Interface

A user-friendly interface was developed to provide a seamless experience. This included building HTML and CSS components for stream management and playback, as well as dynamic JavaScript functionalities like grid layouts and modal dialogs for stream configuration.

### User Handling

Secure user authentication and management were implemented during this phase. Password hashing using Node.js crypto module and the creation of login and registration endpoints ensured user data security. The frontend included responsive forms for account management.

## Stream Configuration

Customizability was added by enabling users to modify stream settings such as resolution, codec, and bitrate. This required updates to both the backend (to process parameters via FFmpeg) and the frontend (to allow intuitive configuration via modals).

## Recording


The final phase introduced recording functionality. Users could toggle recording for streams and download captured segments as MP4 files. This involved managing FFmpeg recording commands, temporary file storage, and frontend interaction for saving recordings.

## Tasks Distribution

The entire project was developed by Aar3n Riveiro Vilar, including the presentation scheduled for December 10, 2024, and the creation of this documentation.

### Estimated time allocation

- Application core: 5-10 hours
- Streaming: 15-20 hours
- Interface: 15-20 hours
- User handling: 10-15 hours
- Stream configuration: 10-15 hours
- Recording: 5-10 hours

 The hours listed are approximate and based on milestones achieved during development process.

Smaller tasks, though not explicitly listed, played an essential role in contributing to the overall effort.

## System Architecture

### Project Files and Directory Organization

▼ Root directory

▼ src

- `server.js` : the main entry point of the application. Sets up the Express.js server, defines middleware, and initializes routes.
- `controllers`
  - `ffmpegController.js` : manages FFmpeg processes, including starting, stopping, and recording streams.
- `routes`
  - `routes.js` : defines API endpoints for user authentication, stream management, and data retrieval.
- `utils`
  - `utils.js` : contains utility functions for tasks like directory creation, password hashing, and validation.
  - `db.js` : handles database initialization and schema creation for user and stream tables.

▼ public

- `login.html` : the login and registration page for user authentication.
- `streams.html` : the main dashboard for managing and viewing streams.
- `style.css` : defines the visual style of the application.
- `login.js` : handles client-side logic for login and registration forms.

- `streams.js` : manages client-side interactions for stream management, configuration, and playback.

▼ config

- `default.js` : contains configuration settings for paths (e.g., output folders, logs...), server port, and FFmpeg parameters.
- `output` : a dynamically created directory used to store processed video files.
- `logs` : a dynamically created directory for storing log files generated by FFmpeg processes, useful for debugging and monitoring.
- `data` : a dynamically created directory used for storing the SQLite database.
- `package.json` : contains metadata about the project, including dependencies (e.g., express, fluent-ffmpeg, better-sqlite3) and scripts for running the application.

## Core Server Setup

The core server acts as the backbone of the application, responsible for initializing essential configurations, serving static files, and routing requests to the appropriate modules.

1. **Entry point:** the `server.js` file is the entry point for the application. It sets up the Node.js server using the Express framework.
2. **Middleware configuration:** the Express framework is configured to allow the server to parse incoming JSON requests, ensuring compatibility with API calls from the frontend, and the static files from the `public` directory are served, enabling the frontend to load resources seamlessly.
3. **Directory setup:** the `output` and `logs` directories are created, which are configurable in the `default.js` file.
4. **Routes integration:** the routes defined in the `routes.js` file are used, which handle public pages like the login and stream dashboards, and other endpoints for user authentication, stream management, and FFmpeg process control.
5. **Server initialization:** the server is launched and configured to listen on a specific port defined in the `default.js` file.

## Routes

The routing system defines how requests from the client are handled and processed by the server. This is achieved using Express.js, with routes organized in the `routes.js` file.


- **Public routes:**
  - `GET /`
    - Serves the login page (`login.html`).
    - Used as the landing page for the application.
  - `GET /streams`
    - Serves the streams dashboard (`streams.html`).
    - Only accessible after successful login.
- **API routes:**
  - User management:
    - `POST /api/login`
      - Validates user credentials against the database.
      - Returns the user ID if authentication is successful.
    - `POST /api/register`
      - Registers a new user by adding their credentials (with hashed passwords) to the database.
      - Ensures that usernames are unique.
  - Stream management:
    - `POST /api/streams`
      - Adds a new video stream to the database for the authenticated user.
      - Returns a unique stream name.

- `PUT /api/streams/:name`
  - Updates the configuration (e.g., codec, resolution) of an existing stream.
- `PUT /api/streams/:name/record`
  - Toggles the recording state (enabled/disabled) of a stream.
- `DELETE /api/streams/:name`
  - Deletes a specific stream from the database.
- `GET /api/streams/:userID`
  - Retrieves all streams associated with a specific user.
- FFmpeg control:
  - `POST /api/start-ffmpeg`
    - Starts an FFmpeg process for a given stream URL and configuration.
- HLS streaming:
  - `GET /hls/:id`
    - Dynamically serves HLS files (e.g., `.m3u8`, `.ts`) for a specific stream.
    - Ensures that only existing streams are accessible.

## Database

The database is a critical component of the system, responsible for securely storing user information and stream metadata. The project uses SQLite, a lightweight, embedded SQL database, to manage these data requirements efficiently.

- **Database setup:** the database is initialized in the `db.js` file using the `better-sqlite3` library. The SQLite database file (`app.db`) is stored in a dynamically created directory (`data`), which can be configured in the `default.js` file. After initialization, the systems verifies the existence of requires tables and creates them automatically if they are missing.
- **Tables and their structure:**
  - Users table
    - `id`: unique identifier for each user (primary key).
    - `username`: unique username chosen by the user.
    - `password`: hashed password for secure authentication.
  - Streams table
    - `id`: unique identifier for each stream (primary key).
    - `user_id`: foreign key linking the stream to a user.
    - `name`: unique name for the stream.
    - `url`: the original stream URL provided by the user.
    - `codec`, `resolution`, `framerate`, `preset`, `bitrate`: configuration settings for the stream.
    - `recording`: boolean indicating if recording is enabled.
    - `recordingDuration`: duration of the recording in seconds.
- **Operations and interactions:** the database is accessed via prepared SQL statements in the `routes.js` file.

 After a new stream is added to the database, its configuration settings are set to the default value defined in the `default.js` file, and the recording is disabled.

## FFmpeg

The FFmpeg controller is a critical part of the backend, responsible for managing video stream processing, including transcoding, recording, and stream configuration. This functionality is implemented in the `ffmpegController.js` file, which uses the `fluent-ffmpeg` library to interact with FFmpeg.

- **Key responsibilities:**

- Stream initialization: launches FFmpeg processes to transcode incoming streams into HLS format.
- Stream management: dynamically adjusts stream settings like resolution, codec, bitrate, and framerate based on user preferences.
- Recording: manages the recording of live streams, converting the output into MP4 format for download.
- Error handling: monitors FFmpeg processes for errors and logs them for debugging purposes.

- **Functions:**

- `startFFmpeg(streamName, streamUrl, config)`
  - Purpose: initializes an FFmpeg process to transcode a video stream into HLS format.
  - Process:
    - Creates a directory for the stream's HLS output, and a log file for the FFmpeg process.
    - Uses `fluent-ffmpeg` to start the transcoding process with the specified settings in the `config` parameter.
    - Writes the HLS playlist (`output.m3u8`) and segment files (`.ts`) to the directory.
    - Writes the FFmpeg process output to its log file, and listens to any errors that could occur.
- `stopFFmpeg(streamName)`
  - Purpose: terminates an active FFmpeg process associated with a specific stream.
  - Process:
    - Identifies and kills the process based on the stream name.
    - Cleans up files and directories related to the stream.
- `recordStream(streamName)`
  - Purpose: initiates recording for a specific stream.
  - Process:
    - Starts an FFmpeg process using `fluent-ffmpeg` to record the stream in MP4 format.
    - Saves the recorded file in a designated directory, making it available for download.


- **Parameters:**

- Streaming:
  - `-tune zerolatency`: optimizes the stream for minimal latency, ideal for live video processing.
  - `-an`: disables audio in the output stream.
  - `-g 60`: sets the keyframe interval to every 60 frames.
  - `-c:v <codec>`: specifies the video codec.
  - `-vf scale=<resolution>`: rescales the video to the desired resolution.
  - `-r <framerate>`: defines the framerate of the output.
  - `-preset <preset>`: adjusts encoding speed vs. quality trade-off.
  - `-b:v <bitrate>`: specifies the target video bitrate.
  - `-maxrate 2*<bitrate>`: caps the maximum bitrate to twice the target bitrate.
  - `-bufsize 4*<bitrate>`: sets the buffer size to four times the target bitrate.
  - `-f hls`: outputs the stream in HLS format.
  - `-hls_list_size <n>`: keeps the playlist size to  $n$  segments (`.ts` files).
  - `-hls_time 1`: sets the duration of each segment to one second.
  - `-hls_flags delete_segments+independent_segments+append_list`: automatically deletes old segments, ensures each segment is decodable independently, and appends new segments to the playlist dynamically.
- Recording:
  - `-live_start_index 0`: starts recording from the first segment in the HLS playlist.
  - `-c copy`: specifies to copy the input directly to the output without the need for recoding.



The `-hls_list_size` parameter controls the recording duration by defining the number of segments retained in the playlist. It's calculation depends on the recording duration and framerate set by the user, the keyframe interval, and the segment duration.


For example, for a stream at 30fps, with a keyframe interval of 60, and a segment duration of 1 second, each segment will store 2 seconds of video. If the user wants to record the most recent 10 seconds, then 5 segments will be needed.

 Active processes are tracked using a global variable. When a new process is requested, the systems checks if it is already running, in which case it will be automatically stopped and restarted with the new settings.

## Utilities

The utilities module is designed to encapsulate common functionality used across the application. This approach ensures modularity, reduces code duplication, and improves maintainability. The utility functions are implemented in the `utils.js` file.

- `createDirectory(directoryPath, source = 'Server')`
  - Purpose: creates a directory if it does not already exist.
  - Process:
    - Checks if the directory specified in the `directoryPath` parameter exists, and creates it if it doesn't.
    - Logs a message to the console, using the `source` parameter to specify who called the function.
- `removeDirectory(directoryPath, source = 'Server')`
  - Purpose: deletes a directory and its content.
  - Process:
    - Checks if the directory specifies in the `directoryPath` parameter exists, and removes it recursively.
    - Logs a message to the console, using the `source` parameter to specify who called the function.
- `hashPassword(password)`
  - Purpose: hashes a plain-text password securely for storage.
  - Process:
    - Generates a random salt.
    - Hashes the password using the `scrypt` algorithm.
    - Returns a string combining the salt and hash.
- `verifyPassword(password, storedHash)`
  - Purpose: verifies that a given password matches a stored hashed password.
  - Process:
    - Extracts the salt and hash from the `storedHash` parameter.
    - Hashes the input password using the extracted salt.
    - Compares the generated hash with the stored hash.

 The `crypto` module in Node.js includes the `scrypt` function for hashing passwords. However, since it follows a callback-based approach, it requires conversion to a promise-based function to use the modern `async / await` syntax. This is achieved by utilizing the `promisify` method from the Node.js `util` module.

## Login and Registration

The login and registration feature is implemented to manage user access to the application securely and efficiently. The frontend components provide a simple and intuitive interface for these operations.

- **Purpose:**
  - Login: authenticates users, granting access to the streams dashboard upon successful verification.
  - Registration: allows new users to create an account by securely submitting their credentials.

- **Components:**

- `login.html`

- Structure:

- Login form for entering the username and password.
      - Registration form for creating a new account, including fields for username, password, and confirmation of the password.
      - Includes links to switch between the forms dynamically.

- `login.js`

- Purpose: handles client-side logic both for login and registration processes.

- Key features:

- Dynamic form toggling between login and registration views.
      - Client-side validation for required fields.
      - Asynchronous requests to the backend API for login and registration.

- **Workflow:**

- Login process:

- Input validation: ensures the username and password fields are not empty, and displays error messages for invalid inputs.

- Backend interaction:

- Sends a `POST /api/login` request with the credentials.
      - On success, retrieves the user ID from the response, stores it in `sessionStorage` and redirects the user to the streams dashboard ( `streams.html` ).
      - On failure, displays an error message (e.g., "Wrong password" or "User does not exist").


- Registration process:

- Input validation: ensured that all fields are filled, and that the passwords match.

- Backend interaction:

- Sends a `POST /api/register` request with the username and password.
      - On success, retrieves the user ID from the response, stores it in `sessionStorage` and redirects the user to the streams dashboard ( `streams.html` ).
      - On failure, displays error messages (e.g., "User already exists" or "Passwords do not match").

 The login form is displayed by default when the page loads.

 The `sessionStorage` API stores data making it only available for the current session and automatically clearing it once the browser tab is closed.

## Streams Dashboard

The streams dashboard system is the primary interface for managing, viewing, and configuring video streams. It is implemented in the `streams.html` file, with dynamic functionality provided by the `streams.js` file.

- **Purpose:**

- Add new streams by providing their URLs.
  - View live streams in an adaptive grid layout.
  - Configure stream settings such as resolution, codec, and bitrate.
  - Record and save segments of streams.
  - Remove streams when they are no longer needed.

- **Components:**


- `streams.html`

- Header: displays the title of the application and a logout button.

- Add stream section: a simple input field for entering the URL of a new stream, and a button to submit the URL.
- Videos container: a dynamically managed grid for displaying active streams.
- Configuration modal: a modal form to adjust stream settings like resolution, framerate, codec, bitrate, and recording duration.
- `streams.js`
  - Allows adding, removing, and configuring streams.
  - Adjusts the grid layout dynamically based on the number of streams.
  - Integrates HLS.js for stream playback.
- **Workflow:**
  - Adding a stream:
    - The user enters a stream URL and clicks the button to submit it.
    - The input is validated and a `POST /api/streams` request is sent to the backend.
    - The stream is added to the database, and a `POST /api/start-ffmpeg` request is sent to the backend in order to start the FFmpeg process for the stream.
    - An HLS video player element is created using HLS.js and added to the dashboard.
  - Viewing streams:
    - Each stream is displayed in the adaptive grid layout as a video element.
    - Controls are available for configuring, recording, removing or saving the stream.
  - Configuring a stream:
    - The user clicks the settings button of a stream and the configuration modal appears, allowing adjustments to parameters such as resolution, codec, and recording duration.
    - Upon submission, a `PUT /api/streams/:name` request updates the stream settings on the database, and the FFmpeg process is restarted with the new parameters.
  - Recording a stream:
    - The user toggles the recording switch of a stream, sending a `PUT /api/streams/:name/record` request to the backend.
    - The new state is stored in the database, and the FFmpeg process is restarted with the corresponding setting.
  - Saving a stream:
    - The user clicks the download button and a `GET /api/streams/:name/download` request is sent to the backend.
    - The recording is downloaded as an MP4 file.
  - Removing a stream:
    - The user clicks the remove button and a `DELETE /api/streams/:name` is sent to the backend.
    - The stream is removed from the database and the dashboard.

 The grid layout automatically adjusts whenever a stream is added or removed:

- **One stream**: occupies the entire screen.
- **Two streams**: each takes up half of the screen.
- **Three or four streams**: each occupies one-quarter of the screen.
- **More than four streams**: they are displayed across multiple rows, with up to five streams per row.

 When the streams dashboard loads, the application automatically retrieves the user ID from the `sessionStorage` to identify the active session. If it does exist:

1. A `GET /api/streams/:userID` request is sent to the backend to fetch all streams associated with the user.
2. For each stream:
  - An FFmpeg process is initiated on the server to start processing the stream.
  - The stream is dynamically displayed on the dashboard.

# Problems and Solutions

## Streams not playing smoothly

- **Problem:** streams were not playing properly because the video source used for testing provided keyframes at irregular intervals. This led to FFmpeg encoding packets without their necessary reference frames, resulting in playback issues.
- **Solution:** after analysing several segments using the `ffprobe` tool, the root cause was identified. The solution was to add the `-g` parameter to every FFmpeg process, which defines the GOP (Group of Pictures) size and ensures that keyframes are inserted at regular intervals, resolving the playback problems.

## FFmpeg logs handling

- **Problem:** FFmpeg outputs its regular logs to `stderr` and sends actual errors through the same stream or `error` events. This behaviour caused confusion when trying to differentiate between standard logs and critical errors, making it difficult to implement proper error handling and logging.
- **Solution:** the issue was addressed by setting up separate handlers for the `stderr` and `error` events.

## Managing FFmpeg parameters

- **Problem:** managing FFmpeg parameters posed a significant challenge due to their complexity and diversity.
  - Fixed parameters that were always required.
  - Customizable parameters by the user, but with default values needed.
  - Dependent parameters that need to be calculated dynamically based on other parameters.
  - Recording parameters which were skipped or not depending on whether the stream needs to be recorded, and for how much time.
- **Solution:** a modular and centralized system for parameter management was designed.
  - `default.js`: this file serves as a centralized storage for all parameters and their default values.
  - `ffmpegController.js`: before running an FFmpeg process, a complete line with the parameters is dynamically assembled, including whatever is needed for each specific stream.

## Avoiding duplicate FFmpeg processes

- **Problem:** when restarting an FFmpeg process, there was a risk of creating duplicate processes for the same stream, because the `kill()` signal for finishing a process is not guaranteed to be instant.
- **Solution:** a promise-based approach was implemented to ensure that the existing process was fully terminated before the new one was started. This approach used process tracking and clean-up mechanisms to avoid conflicts.

## Codec configuration challenges

- **Problem:** the goal was to support a wide range of video codecs to provide users with greater flexibility. However, this posed several challenges.
  - HLS compatibility: HLS is only compatible with certain codecs, primarily H.264 (AVC) and H.265 (HEVC). Many other codecs would require transcoding to HLS-compatible formats, requiring complex additional logic for each unsupported codec.
  - Browser support: the H.265 codec, while supported by HLS, is not natively supported by most browsers because it requires licensing fees.
- **Solution:** the codec configuration was streamlined to focus on H.264 and H.265, ensuring compatibility with HLS while minimizing complexity. H.264 was left as the default codec, while H.265 is left as optional.

✖ On every browser where the application was tested, H.265 was not supported. The only exception might be [Safari](#), but there wasn't any iOS/Mac device available for testing.

## HLS manifest request timeout

- **Problem:** when adding an HLS video element, the browser makes an initial request to the manifest with a default timeout of 20 seconds. Since the manifest was often not available immediately, this led to failed playback attempts, until the timeout finished and the request was repeated.
- **Solution:** the timeout for the manifest request was reduced to one second by configuring the HLS object before attaching the video source, ensuring that the browser retried quickly and avoided prolonged waits.

## Recording only capturing last segments

- **Problem:** by default, FFmpeg's recording only captured the last three segments of the HLS playlist, resulting in incomplete recordings.
- **Solution:** the issue was fixed by adding the `-live_start_index 0` parameter to FFmpeg for recording. This ensured that recording started from the first available segment in the playlist, capturing the entire stream from the beginning.

✓ In its current state, the application operates reliably without any known issues. While there is always room for small improvements and optimizations, these are enhancements rather than problems and do not affect the overall functionality or user experience.

## Installation and Execution

### Prerequisites and Dependencies

Before installing and running the application, the following requirements must be met:

- **Node.js**

- Download and install the [latest version](#) of Node.js and npm.
- Verify the installation by running:

```
1 node --version
2 npm -version
```

- **FFmpeg**

- Download the [latest version](#) of FFmpeg and add it to your system PATH.
- Verify its accessibility by running:

```
1 ffmpeg -version
```

- **Project dependencies**

- Navigate to the project's root directory and install project dependencies by running:

```
1 npm install
```

- **Web browser**

- Use any modern web browser to access the web interface.


✓ The application was tested and worked flawlessly on the following browsers:

- [Google Chrome](#)

- [Mozilla Firefox](#)
- [Microsoft Edge](#)
- [Opera](#)
- [Samsung Internet](#)

- **Camera simulation (optional)**

- [IP Webcam](#): converts your Android smartphone into an IP camera on the local network.
- [IP Camera Lite](#): converts your iPhone into an IP camera on the local network.
- [VLC + WebCam](#): use VLC Media Player to simulate an IP camera using your laptop's webcam.

 The *IP Camera Lite* application could not be tested due to the lack of an iOS/Mac device, so no specific instruction are provided.

## Application Running


### 1. Start the server

- Navigate to the project's root directory and start the server by running:

```
1 npm start
```

### 2. Access the application

- Using your desired web browser, navigate to `http://localhost:3000`.

 To access the application from any device on the local network other than the one running the server, simply navigate to `http://<SERVER_IP>:3000`, where `SERVER_IP` corresponds to the IP of the server's hosting device.

## Example Usage

### 1. Register and log in

- On the initial screen, click on the "Sign up here" link to switch to the registering form.
- Register with a desired username and password.

### 2. Add a stream

- After being redirected to the streams dashboard, enter a video stream URL in the provided field, and click the add button.

### 3. Configure and manage the stream

- Click on the switch to enable recording.
- Click on the settings button, and adjust the stream configuration using the available options.

### 4. Live view


- Watch the live stream in the adaptative grid layout.
- Add or remove streams as required.

### 5. Downloading recordings

- For a stream with recording enabled, click the download button to start downloading an MP4 file corresponding to the configured duration in the settings.

### 6. Log out

- Click the log out button to finish the session and redirect to the login page.

 When adding a video stream for testing, there are two main options:

- **Use a publicly available URL:**

- Public IP cameras or video streams are available on the internet. For example, [this GitHub repository](#) provides a curated list of publicly accessible streams.
- **Generate your own video stream:**
  - IP Webcam:
    - Open the app and click on the “Start server” option located at the bottom.
    - The corresponding URL will be `http://<DEVICE_IP>:8080/video`, where `<DEVICE_IP>` corresponds to the device IP (e.g., `http://192.168.0.7:8080/video`).
  - VLC + WebCam:
    - Open VLC and press `Ctrl + S`. Navigate to the “Capture Device” tab, select your webcam on the “Video device name” selector, and click “Stream”.
    - Click “Next”, select `HTTP` on the “New destination” selector, click “Add”, and click “Next”.
    - Select `Video - H.264 + MP3 (MP4)` on the “Profile” selector, click “Next” and click “Stream”.
    - The corresponding URL will be `http://<DEVICE_IP>:8080`, where `<DEVICE_IP>` corresponds to the device IP (e.g., `http://192.168.0.5:8080`).

## Visual Demonstration

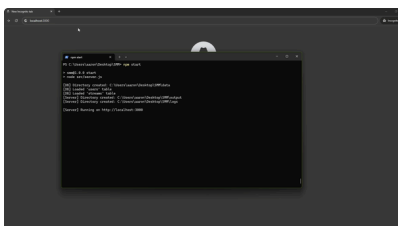
A video demonstration of the application can be found [here](#), showcasing its core functionalities and workflow:

- **Server start-up:** the server is initialized, setting up the backend and database.
- **User registration and login:** a new user is created and successfully logs into the application.
- **Adding a stream:** a video stream URL is added, and the stream is displayed in the dashboard.

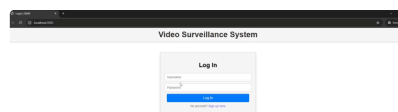
**i** The video feed in the demonstration is captured using an external camera recording the screen where the application is being tested. This setup ensures the capture of a real-time video feed for accurate demonstration.

- **Stream configuration:** stream settings are modified, and recording is enabled.
- **Downloading recording:** the recording is saved as an MP4 file.
- **Dynamic grid updates:** streams are added and removed, demonstrating the adaptive grid layout.

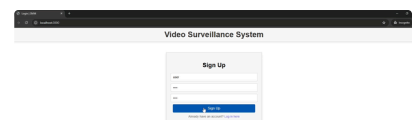
**✗** Whenever stream settings are updated, the stream may stop temporarily, as shown in the video. This happens because HLS tries to load the previous manifest, which becomes invalid when the new FFmpeg process starts.



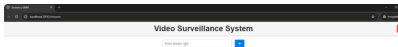
Server start-up



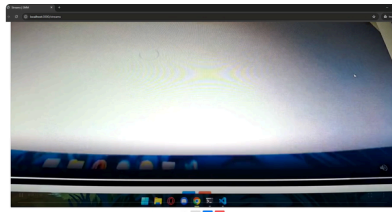
Login screen



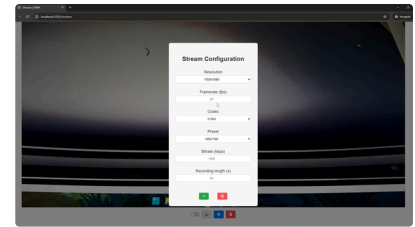
Registration screen



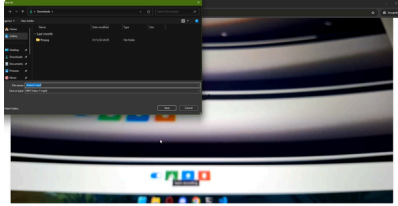
Streams dashboard



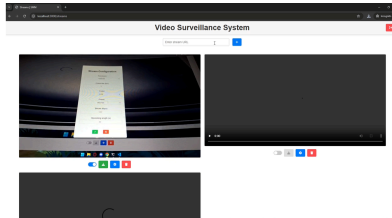
Stream added



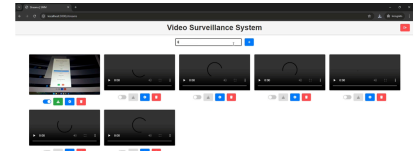
Configuration menu



Downloading recording




Grid with three streams



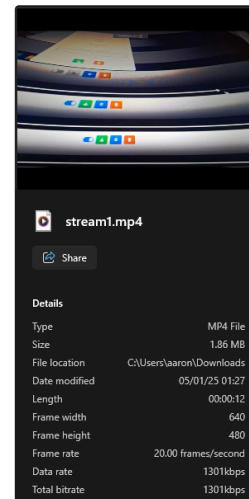
Grid with more than five streams

The properties of the downloaded recording demonstrate the accuracy of the application, as they match with the settings configured before downloading:

- Length: 12 seconds.
- Resolution: 640x480.
- Framerate: 20 frames per second.
- Bitrate: 1301 kbps.

 The recorded video length may not exactly match the configured duration, especially at low framerates. This is due to segments amount being round up.

The bitrate may slightly vary from the configured value. This behaviour is inherent to FFmpeg's processing and encoding methods.



Downloaded file properties

## Conclusion

The **Web Surveillance System** project demonstrates the integration of modern web and video processing technologies into a cohesive and **user-friendly** application. Through the use of Node.js, Express.js, FFmpeg, and a lightweight SQLite database, the backend effectively handles **video processing** and **user management**. Meanwhile, the frontend, powered by HTML, CSS, JavaScript, and HLS.js, provides an **intuitive interface** for users to manage, configure, and view live video streams.

Throughout the development, significant challenges were encountered, such as managing FFmpeg parameters, ensuring compatibility with HLS, and handling browser-specific playback limitations. These obstacles were resolved through systematic analysis and innovative solutions, highlighting the **robustness** and **adaptability** of the system.

While primarily designed for video surveillance, the **flexibility** of the system allows it to be applied to various scenarios requiring **video stream management**. Its modular design ensures **scalability**, **maintainability**, and the potential for **future enhancements**.

In its current state, the application fulfils its intended purpose, providing a **reliable** and **efficient** platform for video surveillance. Future iterations could explore additional features, such as **multi-user support**, **advanced analytics**, and extended **codec compatibility**, further



expanding its usability and appeal.

By completing this project, valuable experience was obtained in **full-stack development**, **multimedia processing**, and **problem-solving**, ensuring a strong foundation for tackling future technical challenges.