WSB MERITO UNIVERSITY TORUN

Semester Project

CityLink Live – Integration with External API
Technical Documentation

Student: Yisak Wodajo
Course: Web Communication Technologies
Date: December 07, 2025

1. Introduction

1.1 Project Context and Goals

CityLink Live is a full-stack web application built as a semester project for the course Integration with External API. The primary goal of the project is to demonstrate how a modern web application can combine user management, real-time communication, and external data sources in a secure and user-friendly way.

From a user's perspective, CityLink Live provides a city-centric experience: users can register, log in, and manage their own profiles; each user can choose a favorite city; for a selected city, the application retrieves live weather data and recent news from external APIs; and users can join a real-time chat room based on that city and exchange messages with others.

From a technical perspective, the project implements a backend API built with Node.js and Express, integration with external APIs (Weatherstack for weather, Newsdata.io for news), a SQLite database for persistent storage, a React and Vite-based frontend, WebSocket-based communication via Socket.io for real-time chat, and security features such as password hashing with bcrypt, JWT-based authentication, and environment-based secrets.

1.2 Learning Objectives

This project is meant to support the following learning outcomes:

• Understand how to design and implement a modern web application with separate frontend and backend layers.
• Gain practical experience with RESTful APIs and WebSocket communication.
• Learn how to integrate external services and handle their responses, errors, and rate limits.
• Learn how to store and manage user data securely using a database.
• Practice software engineering skills such as documentation, testing, and version control.

2. System Overview and Architecture

2.1 High-Level Description

CityLink Live follows a classic client–server architecture with an additional real-time communication layer. The frontend is a React single-page application running in the browser, while the backend is an Express server exposing REST endpoints on http://localhost:5001. The backend connects to a SQLite database via db.js, to external APIs (Weatherstack and Newsdata.io) via HTTP calls using Axios, and to Socket.io for chat.

The user interacts only with the frontend. The frontend, in turn, calls the backend's API and subscribes to Socket.io events to get real-time updates.

2.2 Main Components

The system can be logically divided into several components:

• Authentication and user management: handles registration, login, JWT token creation, and user profile management, with support for admin users.
• City dashboard: combines weather data, news articles, and real-time chat for a chosen city.
• Admin panel: allows administrators to view, edit, and delete user accounts and toggle admin status.
• Database layer: uses SQLite for persistent storage of users and messages.
• External API integration: Weatherstack and Newsdata.io for weather and news data.
• Real-time chat: uses Socket.io to manage city-based chat rooms and broadcast messages.

2.3 Architecture Overview

At a high level, the browser hosts the React frontend, which communicates with the Node.js and Express backend through HTTP and WebSocket connections. The backend uses Axios to call external APIs and uses SQLite to persist users and messages. Socket.io enables real-time chat between multiple clients.

3. Technologies Used

3.1 Frontend

The frontend is built with React and Vite. React is used to build a single-page application with reusable components and hooks such as useState and useEffect. Vite provides a fast development server and build tool. React Router manages 3navigation between pages, and Axios is used to send HTTP requests to the Backend.

3.2 Backend

The backend is implemented with Node.js and Express. Express provides routing and middleware support. Socket.io adds real-time communication on top of the HTTP server. Axios is used to call the external Weatherstack and Newsdata.io APIs. Authentication is handled with JSON Web Tokens using the jsonwebtoken library, and passwords are hashed securely with bcrypt. The cors middleware enables cross-origin requests from the frontend during development, and dotenv loads environment variables from the .env file.

3.3 Database and Persistence

The project uses SQLite as a lightweight relational database. The sqlite3 driver connects the Node.js backend to the database. The db.js module opens the database, creates tables if they do not exist, and seeds a default admin user when necessary. All user accounts and chat messages are stored persistently in the citylink.db file.

3.4 External APIs

CityLink Live integrates two external APIs. Weatherstack provides current weather data for a given city, including temperature, feels-like temperature, description, humidity, and wind speed. Newsdata.io provides recent news articles for a query term, which in this application is the city name. The backend normalizes the responses from both providers into simple JSON structures that the frontend can easily display.

4. Backend Design

4.1 Server Initialization

The backend entry point is index.js. It imports the required libraries and the db.js module, creates an Express application, and wraps it in an HTTP server. Socket.io is attached to this server to handle real-time events. Middleware such as cors and express.json are registered to enable cross-origin requests and JSON body parsing. The server defines routes for health checks, authentication, profile management, admin actions, external data, and message history, and then starts listening on the configured port

4.2 Authentication Logic

Authentication is based on email and password. During registration, the server checks whether a user with the given email exists. If not, it hashes the password with bcrypt and inserts a new row into the users table with email, password_hash, username, favorite_city, and is_admin. During login, the server retrieves the user by email, compares the provided password with the stored hash using bcrypt.compare, and if valid, signs a JWT containing the user id, email, username, and is_admin. The token and basic user data are returned to the client.

4.3 Auth and Admin Middleware

The authMiddleware function checks for an Authorization header, extracts the token, and verifies it using jwt.verify and the JWT_SECRET. If the token is valid, the decoded payload is attached to req.user. Otherwise, the request is rejected with a 401 or 403 status. The adminMiddleware function relies on authMiddleware and checks the is_admin flag on req.user, rejecting access if the user is not an Administrator.

4.4 REST Endpoints

The backend exposes REST endpoints for authentication, profile management, admin operations, external data, and message history. Public routes include POST /api/register, POST /api/login, and GET /api/health. Protected routes include GET /api/me, PUT /api/profile, DELETE /api/profile, and GET /api/rooms/:city/messages. Admin-only routes include GET /api/users, PUT /api/users/:id, and DELETE /api/users/:id. External data is provided by GET /api/cities/:city/weather and GET /api/cities/:city/news.

4.5 Real-Time Chat with Socket.io

Socket.io is configured on the same HTTP server as Express. When a client connects, the server listens for joinRoom and chatMessage events. The joinRoom event causes the socket to join a city-specific room, based on the city name. When chatMessage is received, the server validates the payload, inserts a new row into the messages table, and broadcasts the message to all clients in the same city room using io.to(city).emit. This design ensures that chat messages are both persisted and delivered in real time.

5. External API Integration

5.1 Weatherstack Integration

5The route GET /api/cities/:city/weather constructs a request to the Weatherstack current weather endpoint using the WEATHERSTACK_API_KEY and the requested city. Axios is used to send the request with a timeout. The server checks for errors in the response payload and maps successful responses into a simplified object containing the city, country, temperature, feels_like, description, humidity, and wind_speed fields. This abstraction hides the original API format from the frontend and makes it easier to switch providers in the future.

5.2 Newsdata.io Integration

The route GET /api/cities/:city/news calls the Newsdata.io news endpoint using the NEWS_API_KEY and a query equal to the city name. The server checks that the response status is success and then maps the results array into a list of articles with title, description, url, source, and publishedAt fields. The list is limited to a small number of articles for readability. As with the weather endpoint, this abstraction simplifies the frontend logic.

5.3 Error Handling and Timeouts

Both Weatherstack and Newsdata.io requests are wrapped in try and catch blocks with timeouts set to 5 seconds. If a request fails or returns an error payload, the server logs the details and responds with a generic error message such as Failed to fetch weather data or Failed to fetch news. The frontend displays a human-friendly message when such errors occur, ensuring that external API

failures do not crash the application.

6. Database Schema

6.1 Users Table

The users table stores account and profile information. It includes a primary key id, a unique email, a password_hash for secure credentials, a username for display, an optional favorite_city, an is_admin flag to distinguish administrators from normal users, and a created_at timestamp. The db.js module ensures that the table is created if it does not exist and inserts a default administrator user when necessary.

6.2 Messages Table

The messages table stores chat messages. Each message has a primary key id, a user_id foreign key referencing the users table, a city string indicating the chat room, the content of the message, and a created_at timestamp. Messages are inserted when the server handles chatMessage events from Socket.io clients. 6Message history for a city is retrieved via GET /api/rooms/:city/messages.

6.3 CRUD Operations

The application supports basic create, read, update, and delete operations on users and messages. Users are created via POST /api/register and updated via PUT /api/profile and PUT /api/users/:id. User accounts can be deleted by the user through DELETE /api/profile or by an admin through DELETE /api/users/:id. Messages are created when chatMessage events are received and read via the message history endpoint. When a user is deleted, their messages are also Removed.

7. Frontend Design

7.1 Routing and Navigation

The frontend uses React Router to define routes for the dashboard, login, register, profile, and admin pages. The top navigation bar includes links to Home, Login, Register, Profile, and Admin, depending on whether the user is authenticated and whether they have admin privileges. The Logout button clears the JWT token and user data from local storage and returns the user to the login page.

7.2 API Client and State Management

An Axios instance is configured in api.js with a base URL pointing to the backend API. An interceptor attaches the JWT token stored in local storage to the Authorization header for all outgoing requests. React hooks such as useState and useEffect are used throughout the components to manage local state and trigger side effects such as loading user data, weather, news, and chat history.

7.3 Key Pages

The LoginPage and RegisterPage components implement the authentication UI, with forms that submit to the backend authentication routes. The Dashboard component is the main view, combining weather, news, and chat for a selected city. ProfilePage shows and updates the current user's profile and allows the user to delete their account. AdminPanel exposes a table of all users with controls for editing and deleting accounts and managing admin status.

7.4 Personal API (external Intergration)

Generate & View : This demonstrates that the project not only consumes third-party APIs, but also exposes its own API for integration.

- Page: Profile
- Endpoints (JWT):
    - GET /api/personal-api-token
        - Returns { has_token, masked_token }
    - POST /api/personal-api-token
        - Generates a new token for the logged-in user
        - Returns { api_token: "FULL_TOKEN_ONCE" }

The full token is shown only once. After that, only a masked version (e.g. abcd...wxyz) is visible.

Authentication with API Key

External tools send: x-api-key: YOUR_PERSONAL_API_TOKEN

External API Key Endpoints

- GET /api/external/me

  Returns basic user info for that key
- GET /api/external/cities/:city/weather

  Same weather data as the dashboard
- GET /api/external/cities/:city/news

  Same news data as the dashboard

Example Usage (curl)

```
curl "http://localhost:5001/api/external/cities/Warsaw/weather" \
 -H "x-api-key: API_TOKEN"

curl "http://localhost:5001/api/external/me" \
```

```
-H "x-api-key: API_TOKEN"
```

8. Security Considerations

8.1 Password Security

Passwords are never stored in plain text. Instead, bcrypt is used to hash passwords with a salt before storing them in the database. During login, the provided password is hashed and compared to the stored hash using bcrypt.compare, which protects against direct password disclosure even if the database were compromised.

8.2 Authentication and Authorization

The application uses JSON Web Tokens for authentication. Tokens are issued upon successful login and must be included in the Authorization header for protected routes. The authMiddleware verifies tokens and rejects requests without valid tokens. Admin-only routes are additionally protected by adminMiddleware, which checks the is_admin flag.

8.3 API Key Management

API keys for Weatherstack and Newsdata.io are stored in environment variables in the backend .env file. The .env file is excluded from version control using .gitignore to prevent accidental exposure of secrets in the repository. Only .env.example is committed, which documents the required environment variables without containing real keys.

9. Testing

9.1 API and Backend Testing

API routes were tested using tools such as Postman or Thunder Client, as well as through direct browser requests for simple GET endpoints. Tests ensured that correct status codes and response formats were returned for successful operations and for error cases such as invalid credentials, missing tokens, and unauthorized access to admin routes.

9.2 Frontend and User Flow Testing

The frontend was tested by walking through typical scenarios: registering new users, logging in, updating profiles, changing favorite cities, viewing weather and news for different cities, participating in chat from multiple browser windows, and performing admin actions such as editing and deleting users. Error messages and edge cases, such as missing external API keys, were also exercised.

10. Deployment and Running Instructions

10.1 Local Development Setup

To run the project locally, the repository is cloned and both backend and frontend dependencies are installed using npm. The backend requires a .env file based on .env.example, with the port, JWT secret, and external API keys. The backend server is started with npm start in the backend directory, and the frontend is started with npm run dev in the frontend directory. The application is then accessible from the browser at http://localhost

10.2 Potential Production Deployment

For a real deployment, the backend could be hosted on a Node-compatible platform and connected to a managed database. Environment variables would be set through the hosting provider. The frontend could be built into static assets with npm run build and deployed to a static hosting service such as Netlify or Vercel. The frontend's Axios base URL would need to point to the public backend URL.

11. Challenges and Future Work

11.1 Challenges Faced

Several challenges were encountered during the project. Initial attempts to integrate other weather and news providers resulted in authentication errors and rate-limit issues, which helped reinforce the importance of correct API key configuration and careful reading of provider documentation. Implementing role-based admin access also required changes to the schema, middleware, and frontend navigation.

11.2 Future Improvements

Future enhancements could include more advanced form validation, better error messages, user avatars, richer chat features, and more robust security measures such as security headers, rate limiting, and comprehensive input sanitization. Automated testing with a framework such as Jest could also be added to increase confidence in changes over time.

12. Conclusion

CityLink Live is a complete example of a modern web application that integrates user management, external APIs, and real-time communication. It demonstrates how a React frontend can work together with a Node.js and Express backend, backed by a SQLite database and enhanced by external weather and news services. It also shows how JWT-based authentication, role-based authorization, and environment-based configuration contribute to secure and flexible design. From a learning perspective, the project consolidates key concepts of web engineering, including API design, persistence, error handling, security, and documentation. The resulting application meets the course requirements for the Integration with External API project and forms a solid foundation for further extensions and improvements.