ecovis connect documentation

docker-compose.yml (Database Configuration)

The docker-compose.yml file is used to define and configure services, networks, and volumes for your multi-container Docker applications. In the context of a database, this file likely sets up and configures the database container(s) and any necessary dependencies.

What it does:

- **Defines the database service**: The file will specify the image to be used (e.g., mysql, postgres, etc.), along with environment variables that configure the database settings (like the database name, username, and password).
- **Port mapping**: It ensures the database can be accessed from your local machine or other services by mapping the internal container port (e.g., 3306 for MySQL) to an external port on the host.
- **Volumes**: It may define a volume to persist the database data so that it remains intact even if the container is stopped or removed.
- **Networks**: If there are other services (like a web app or API) in the same Docker Compose setup, the database container will be connected to them through a shared Docker network, enabling communication between the containers.

Example Explanation:

The docker-compose.yml file includes the configuration for a MySQL database with the following:

1. **Service**: The database container is defined under the services section (e.g., db), where we specify the database image to use (e.g., mysql:5.7).
2. **Environment Variables**: Variables like MYSQL_ROOT_PASSWORD, MYSQL_DATABASE, and MYSQL_USER are provided to set up the database on container initialization.
3. **Volumes**: A volume is used to store the database data persistently so that the data survives container restarts.
4. **Port Binding**: Port 3306 (default MySQL port) is exposed to allow external connections.
5. **Networks**: If applicable, it connects the database to other containers within the same network, ensuring they can communicate internally.

vite.config.js (Vite Configuration)

The vite.config.js file is used to configure and customize the build and development settings for your project using Vite, a fast build tool and development server. This file contains project-specific settings, including how Vite compiles and serves your code.

What it does:

- **Development Server Configuration**: It defines how the Vite development server behaves (e.g., port number, proxy settings, etc.).
- **Build Configuration**: Configures options for bundling the application for production, such as output directory, optimization settings, etc.
- **Plugins**: You can include and configure various Vite plugins (e.g., for handling specific types of files or optimizing performance).
- **Customization**: It may specify custom settings for CSS preprocessors, environment variables, or other parts of the build process.

yarn.lock (Yarn Dependency Lock File)

The yarn.lock file is automatically generated when you install dependencies using Yarn. It locks the versions of all installed packages, ensuring consistency across all environments.

What it does:

- **Locks Dependency Versions**: Ensures that all contributors and environments use the exact same versions of dependencies and their sub-dependencies. This helps avoid version mismatches that could lead to bugs or inconsistencies.
- **Optimizes Installations**: Speeds up installation processes by ensuring the exact same dependency tree is used across different setups.

Important Note: This file **must not be manually modified**. Yarn will update it automatically when dependencies are added or updated. Altering it manually could lead to inconsistencies or errors.

package.json (Project Dependencies and Scripts)

The package.json file contains metadata about the project, including a list of dependencies, scripts, and other configurations like project name, version, and repository details.

What it does:

- **Dependencies**: Lists the dependencies and devDependencies (e.g., libraries and packages your project requires), along with their version ranges.
- **Scripts**: Defines useful command-line scripts (e.g., yarn start, yarn build) that automate common tasks like starting the server, building the project, running tests, etc.
- **Project Metadata**: Contains basic information about the project like the name, version, description, and author details.

Important Note: **This file should not be manually altered** for most configuration-related tasks. Any changes to dependencies should be done via Yarn commands (yarn add <package>, etc.) to ensure consistency with the yarn.lock file. The configuration in package.json is tightly coupled with the versions listed in the yarn.lock file, and altering one without the other could break things.

General Instruction:

Important: The Vite configuration, yarn.lock, and package.json should not be altered unless necessary.

- Changes to dependencies must be made using Yarn commands (yarn add, yarn remove).
- Configuration adjustments should be done within the appropriate files (e.g., vite.config.js for Vite-specific settings) but should avoid direct edits to yarn.lock and package.json outside of dependency updates.

Frontend Overview

The frontend of the project uses a **standard React setup** combined with **ViteJS** and **Tailwind CSS** for efficient development and styling.

main.jsx (React Setup & AuthContext)

The main.jsx file is the entry point of the React application, where key configurations are made:

- **React StrictMode**: This is enabled to help identify potential problems in the app during development. It intentionally renders components twice to surface any unsafe lifecycle methods or side effects, providing valuable warnings to improve code quality.
- **AuthContext**: This is a custom context that handles user authentication across the app. By using this context, the app can manage the user's login state (e.g., whether the user is authenticated, their profile, tokens, etc.) in one place and make it easily accessible to any component throughout the app.

The AuthContext makes use of React's Context API to provide authentication data and functionality (such as login and logout) to all components that need it, without the need to pass this data through props at every level of the component tree. This means that any component can simply access the current authentication state or trigger authentication-related actions.

Tailwind CSS (Styling)

Tailwind CSS is a utility-first CSS framework used to style the app. It allows developers to apply styles directly within the JSX using utility classes (e.g., bg-blue-500, p-4, text-center). This approach leads to a faster and more efficient styling process since it eliminates the need to write custom CSS for common styles, allowing you to compose your UI directly using predefined classes.

With Tailwind, you can easily customize the layout, spacing, colors, and other aspects of the design while keeping the styles modular and responsive. This method makes it easy to maintain a consistent design system across the app.

In summary:

- **ViteJS** provides fast bundling and efficient development.

- **React** is used for the core UI, with **React StrictMode** enabling development-time checks.
- **AuthContext** ensures centralized management of user authentication state.
- **Tailwind CSS** streamlines styling by applying utility classes directly to components.

Navbar.jsx (Navigation Bar)

The Navbar.jsx component is responsible for rendering the navigation bar of the application. It typically displays links or buttons for navigating between different sections of the app. Since the app involves authentication, the Navbar also conditionally renders different items based on whether the user is logged in or not, allowing access to authorized areas like user management or profile upload, and hiding them if the user is not authenticated.

AuthContext.jsx (Authentication Context)

The AuthContext.jsx file contains the **AuthContext** and the related logic for managing authentication and authorization. It centralizes authentication data and functions like logging in, logging out, and checking if the user is authenticated. By using **React Context**, it makes authentication state available to all components across the app.

- It tracks information such as whether a user is logged in and stores their data (like the user profile or authentication token).
- It helps in conditionally rendering different views or routes based on the authentication state (for example, preventing non-authenticated users from accessing certain pages).

Login.jsx (Login Component)

The Login.jsx component is responsible for handling user login functionality. It contains the logic for:

- **Login form**: Collects user credentials (typically username/email and password).
- **Authentication**: It triggers the login process via an API call to authenticate the user.
- **Error handling**: Displays messages if the authentication fails (e.g., incorrect credentials).

Once authenticated, the component typically redirects the user to the home page or a protected route.

App.jsx (Routing)

The App.jsx file handles the **routing** of the application. It uses a routing library (like React Router) to define the different routes of the app, such as the login page, the user management panel, and other components. It determines which component should be rendered based on the URL path.

- **Protected Routes**: App.jsx will check if the user is authenticated (using AuthContext) and render either the requested page or redirect them to a login page if they don't have proper authorization.

Components Overview

1. **Google.jsx (rss Integration)**
   This component fetches and displays RSS feeds from various sources (e.g., news websites or blogs). It handles the parsing of RSS feed data and renders it in a readable format within the app.
2. **UserManagement.jsx (Superadmin Panel for User Management)**
   This component is available for **superadmin** users and allows them to manage other users in the system. It typically includes features such as:
   - Viewing user details.
   - Editing user roles and permissions.
   - Deleting or disabling user accounts.
   - This panel is restricted to superadmin users only.
3. **List.jsx (User Contact List)**
   The List.jsx component displays a list of user contacts. It likely fetches user data from a backend and shows contact details (such as name, email, and phone numbers). It may allow users to interact with the contact list by searching or filtering the data.
4. **Next.jsx (File Management and Data Upload)**
   This component handles the uploading and management of files. Users can upload files and it may allow them to view, delete, or manage the uploaded files. This feature may be linked to user-related data or application-specific needs (like CSV uploads for user data).
5. **ProfileUpload.jsx (Admin Profile Upload)**
   The ProfileUpload.jsx component allows an **admin** (not a superadmin) to upload or update user profile data. This could involve uploading images, documents, or any relevant files tied to the user's profile, and may provide an interface for the admin to manage individual user information.
6. **Search.jsx (Search Component)**
   The Search.jsx component provides a search functionality for users to search for data (such as contacts, posts, or other user-related information). It likely takes input from the user, queries the backend or a data store, and displays matching results dynamically

Backend Overview:

The backend is responsible for processing CSV uploads, checking and cleaning the data, running scrapers to gather LinkedIn profile information, and finally storing the cleaned data into the database. The main logic is managed in **server.js**, and several utility files handle specific tasks to ensure the data is processed, cleaned, and formatted correctly before insertion into the database.

The backend uses **execSync** for running synchronous operations, which means each task is handled one after the other.

1. empty.js (Check and Remove Empty Entries)

This file is responsible for **checking and removing empty entries** from the JSON data after it's uploaded. When an admin uploads a CSV file, it gets converted to JSON. If there are any empty JSON objects or fields, this file will **clean up the data** by removing those empty entries before further processing.

2. processcsv.js (CSV to JSON Conversion)

The processcsv.js file handles the task of **converting the uploaded CSV file into JSON format**. This step is crucial because once the data is in JSON format, it can be manipulated more easily for checking, cleaning, and further processing.

3. check.js (Duplicate Check for LinkedIn Profiles)

The check.js file is used to **check for duplicate entries** of LinkedIn profiles. It compares the LinkedIn profiles in the uploaded data against the database to identify if any profiles already exist. This helps to prevent the database from containing duplicate records, ensuring the integrity of the data.

4. main.js (LinkedIn Scraper)

The main.js file is the heart of the scraper process. It **scrapes LinkedIn profiles** based on the provided data (likely URLs or profile identifiers) and gathers all available data related to those profiles. After scraping the data, it generates a CSV file with all the collected information, which is then converted to JSON in a later step. This file likely runs a script that accesses LinkedIn's publicly available data (or potentially uses an API).

5. csv.js (CSV to JSON Conversion)

Once the LinkedIn profile data is scraped and generated as a CSV, the csv.js file is responsible for **converting this new CSV data back into JSON**. The conversion makes it easier to process the data in the next steps and ensure it aligns with the format needed for database insertion.

6. dead.js (Remove Failed Scraper Entries)

The dead.js file processes the **failed scraper entries**. If any scraper attempts fail (for example, if a LinkedIn profile cannot be fetched), these failed entries are moved to a not_success.json file. The file also **removes these failed entries** from the output JSON to ensure only successfully scraped profiles are kept for database insertion.

7. last.js, last2.js, last3.js (Add Reference and Class Info)

These files are responsible for **adding reference and class information** to the data. When the admin uploads a CSV, these files will **augment the JSON data** with extra reference and class information, which could be metadata such as the source of the data, categories, or classifications related to the profiles. These files help in adding more context to the scraped data.

8. refdown.js & refup.js (Make Database Changes)

The refdown.js and refup.js files are used for **making changes to the database**. These files likely handle operations that update or modify references to specific records, ensuring that the right relationships and references are made in the database before the data is stored.

- refdown.js might be responsible for **removing or updating references** in the database.
- refup.js might handle **adding or modifying references** as needed.

These files are crucial for maintaining the consistency of data in the database.

9. clean.js (Data Cleanup and Masking)

The clean.js file **cleans up the data further** before it gets uploaded to the database. This step involves additional data manipulation and **masking** of sensitive or non-compliant information. It ensures that the data conforms to the format and structure required by the database, making it ready for insertion.

10. cc.js (Remove Current Company Anomalies)

The cc.js file is focused on **removing anomalies** related to the "current company" data in the scraped LinkedIn profiles. If the scraper returns incorrect or incomplete company data, this file ensures that such anomalies are cleaned out to maintain data accuracy and consistency before it's uploaded to the database.

11. Final Step (Database Upload)

Once all the files have processed the data—checking for duplicates, scraping LinkedIn profiles, cleaning, and augmenting the data—the final step is to **upload the cleaned data to the database**. This happens after all necessary adjustments are made to ensure that the data is valid, complete, and properly formatted for storage. The database is updated with the newly processed and sanitized records.

General Flow:

1. **CSV Upload**: The admin uploads a CSV.
2. **CSV to JSON**: The file is converted to JSON using processcsv.js and csv.js.
3. **Empty Entries Removal**: empty.js removes any empty entries.
4. **Duplicate Check**: check.js checks for duplicate LinkedIn profiles.
5. **Scraping**: main.js scrapes LinkedIn profiles and generates new data.
6. **Failed Entries Removal**: dead.js removes failed scraper entries.
7. **Augment Data**: last.js, last2.js, and last3.js add reference and class info.
8. **Database Updates**: refdown.js and refup.js make necessary database changes.
9. **Further Cleanup**: clean.js and cc.js clean the data and remove anomalies.
10. **Database Upload**: The final data is uploaded to the database.