ChatGPT

Overhauling the Utility Employee Daily Reporting System

Project Context: We have a daily reporting system for utility company employees (built with **React + Vite + Tailwind CSS** on the front-end and **Express.js** on the back-end). The current implementation is incomplete and buggy, with integration issues between front-end and back-end (e.g. CRUD operations not persisting to the database). The initial database was MySQL (via phpMyAdmin), but we plan to migrate to **PostgreSQL** for production. Key features include user authentication (with *superuser* vs *regular user* roles), a dashboard, daily reports (laporan), employee data management (for superusers), daily/weekly/yearly checklists (uploaded by superuser, read-only for regular users), and an option to export data to PDF.

In this comprehensive guide, we will address the issues and outline how to **fix, integrate, and complete** the project, often by restructuring or rewriting parts of it for clarity and reliability. The solution will cover:

- Migrating the database from MySQL to PostgreSQL.
- Fixing back-end integration and CRUD functionality (ensuring data actually saves to the DB).
- Implementing proper authentication and session handling.
- Setting up role-based access control (superuser vs normal user).
- Front-end integration: connecting React to the Express API (with correct config).
- Handling file uploads for the checklists feature.
- Implementing PDF export functionality.
- Basic project setup and best practices for development and production.

1. Migrating to PostgreSQL for Robust Data Storage

Why PostgreSQL: PostgreSQL is an enterprise-grade open-source RDBMS with strong reliability and features (ACID compliance, richer JSON support, etc.). Migrating from MySQL involves updating the ORM configuration and possibly adjusting data types or queries that were MySQL-specific.

Sequelize Configuration: In the current back-end code, the Sequelize initialization is pointed at MySQL. We need to change this to use PostgreSQL. Steps to do this:

- **Install the Postgres driver**: Sequelize (v6) works with PostgreSQL via the `pg` library (and `pg-hstore` for certain data types). Install these in the backend project: `npm install pg pg-hstore` [1]. This provides the necessary low-level drivers.
- **Update Sequelize dialect and connection settings**: In `config/Database.js` (or wherever the Sequelize instance is created), set the dialect to `'postgres'` (instead of `'mysql'`). Also provide the correct database name, username, password, host, and port for your Postgres instance. For example:

```
import { Sequelize } from "sequelize";
const db = new Sequelize('report_app', 'postgres_user',
'postgres_password', {
  host: 'localhost',
  dialect: 'postgres'
});
export default db;
```

This uses separate params. Alternatively, you can use a connection URI:

```
const db = new Sequelize('postgres://user:pass@localhost:5432/
report_app');
```

Sequelize supports both formats [2] [3]. Ensure the credentials in the code (or in environment variables) match your Postgres setup.

- **Use environment variables** for sensitive config: Instead of hardcoding values, load them from a `.env` file with `dotenv`. e.g., `DB_NAME=report_app, DB_USER=..., DB_PASS=...`, then use `new Sequelize(process.env.DB_NAME, process.env.DB_USER, process.env.DB_PASS, { dialect: 'postgres', host: process.env.DB_HOST, ... })`. This makes it easy to change config without touching code.
- **Check data types and queries**: Most Sequelize code will work unchanged on Postgres, but double-check any raw SQL queries or model definitions. For instance, MySQL `DECIMAL(10,2)` or `ENUM` types are also supported in Postgres, so those can remain. The code's models (e.g., `amount` as DECIMAL, `type` as ENUM) should be fine on Postgres. Enums will be created as Postgres enum types by Sequelize.
- **Sync or migrate schema**: Since the code uses `db.sync()` on startup, running the application after changing the config will attempt to create the tables in the new Postgres database. Watch the server logs for any errors during `sync()`. If the MySQL database had existing data that needs migrating, you'll have to export/import it (through CSV or a migration script). If starting fresh, the sync will create new empty tables. In development, you can use `force: true` with sync (temporarily) to drop and recreate tables to match models – but **do not** use `force: true` in production as it will drop data.
- **Test the connection**: Use `db.authenticate()` to verify that the app can connect to Postgres, and catch any authentication or network errors [4]. Example:

```
try {
  await db.authenticate();
  console.log('PostgreSQL connection established successfully.');
} catch (error) {
  console.error('Unable to connect to the database:', error);
}
```

- **Remove MySQL-specific code**: If there are remnants like MySQL pooling or raw queries (e.g., using `mysql2` directly), these should be removed or replaced with Sequelize or `pg` equivalents. The provided project seems to primarily use Sequelize, so this might be minimal.

By making these changes, the backend will be fully using PostgreSQL. This addresses the database integration issues and sets the stage for reliable data persistence. (PostgreSQL will handle the CRUD operations as long as our code sends the queries correctly.)

2. Fixing Backend Integration and CRUD Issues

With the database in place, we need to ensure the **Express backend** correctly handles create/read/update/delete operations and that data flows from the client to the DB. The current bugs (like "CRUD not saved to database") suggest some logic errors or missing pieces in the backend. Let's tackle a few critical areas:

a. Model Relationships & Missing Fields:
Examine the Sequelize models and their associations. In the project code, for example, we have `Users`, `Karyawans`, `Laporans`, `Checklists`, etc. We must ensure that relationships are defined so that foreign keys exist in the tables:

- It appears **Users** and **Karyawans** should have a one-to-many relationship (each user might have multiple "karyawan" entries). However, in the provided `KaryawanModel.js`, there was no `userId` field defined, yet the controller was querying `where: { userId: req.userId }`. This is a bug – the model definition lacks the foreign key. To fix this, add a `userId` field in the Karyawan model definition (as a foreign key reference to Users), or simpler, define the association in Sequelize so it auto-adds it:

  ```
  Users.hasMany(Karyawans, { foreignKey: 'userId' });
  Karyawans.belongsTo(Users, { foreignKey: 'userId' });
  ```

  According to Sequelize docs, calling `A.hasMany(B)` will automatically add the foreign key (e.g., `AId`) to model B if it doesn't exist [5]. So in this case, `Users.hasMany(Karyawans)` adds `userId` to Karyawans (assuming default naming). Make sure this association call happens *before* `db.sync()` so that the column is created. With this, queries like `findAll({ where: { userId: req.userId } })` will work as expected.
- Similarly, check other relationships: The code shows `Karyawans.hasMany(Laporans)` and `Users.hasMany(Laporans)` which implies `Laporans` should have `userId` and `karyawanId` foreign keys. The `LaporanModel.js` did call `Laporans.belongsTo(Users)` and probably got `userId` added (since we saw `Users.hasMany(Laporans)` in the code). Just ensure all associations are consistent:
- If each report (`Laporan`) belongs to one user and one category/employee (`Karyawan`), define both: `Laporans.belongsTo(Users, { foreignKey: 'userId' })` and `Laporans.belongsTo(Karyawans, { foreignKey: 'karyawanId' })`, along with the inverse `Users.hasMany(Laporans)` and `Karyawans.hasMany(Laporans)`. This matches what the code intends.
- For **Checklists**, if they are also tied to a category (daily/weekly/yearly might be modeled as entries in Karyawans or a separate entity), ensure foreign keys are set accordingly (the code snippet showed `Karyawans.hasMany(Checklists)` which would add `karyawanId` to Checklists).
- After adjusting models, run the migration (sync) to update the schema. In development, you might use `force: true` once to recreate tables with new fields. In production, use a proper migration script instead of force syncing to avoid data loss.

b. CRUD Controller Logic:
Review the controller functions for creating and updating records to ensure they correctly save to DB:

- **Create operations**: For example, in `createLaporan` (report creation), the code was doing `await Laporans.create({...})` and then `res.status(201).json({ msg: "success" })`. That looks correct. If data wasn't saving, possible reasons:
- The transaction might have failed (e.g., validation error, foreign key constraint fail) and they are catching the error. Check if `error.message` reveals something. It might be due to an invalid `karyawanId` or missing association (as discussed above). Fixing the model relation could resolve this.
- Another possibility is that the **client isn't sending the request properly** (e.g., missing `karyawanId` in the request body). Ensure the front-end includes all required fields. We'll handle front-end in the next section, but it's worth validating on the server: if a required field is missing, send a 400 with a clear message (as the code does for missing `karyawanId`).
- **Update operations**: Ensure that update controllers use the correct method (`.update` or fetching the instance and modifying). The code likely uses `Laporans.update({...}, { where: { id } })` or similar. This should work, but verify that `id` is the primary key (Sequelize will use the `id` field by default as PK if defined). If using UUIDs, ensure the lookup is by the right field (maybe they use `uuid` field for public facing IDs). For consistency, you might want to use `uuid` (which is generated) as an identifier in URLs instead of numeric `id` to avoid exposing internal IDs. The code did have a `uuid` field in each model. If so, you can change routes to use `:uuid` and query by `uuid` instead of numeric id. This is an enhancement for security/UX but not strictly necessary.
- **Delete operations**: Similar check — likely fine if `.destroy({ where: { id }})` is used. One integration bug could be that the front-end is not sending the correct identifier or method (e.g., sending a GET or POST instead of DELETE due to how fetch/axios call is written). We'll ensure the front-end calls the correct HTTP verb and URL.
- **User registration**: The `createUser` controller (for registering a new user) was adding default "karyawan" entries (which in original code looked like categories). After fixing the association, that code should do something like:

  ```
  const newUser = await Users.create({ username, email, password:
  hashedPassword });
  const defaultKaryawans = [ ... ];  // array of default categories
  // Attach userId to each
  defaultKaryawans.forEach(k => k.userId = newUser.id);
  await Karyawans.bulkCreate(defaultKaryawans);
  ```

  Ensure this runs without error. If `userId` wasn't a column, it would error – but we have added it. Also, check that `newUser.id` is the correct reference (Sequelize will have `newUser.id` as the auto-increment primary key). If we prefer to use `uuid` as reference, could use `newUser.uuid` as well, but since userId (numeric) is now in Karyawan, stick to numeric IDs internally.
- After changes, **test each API** with a tool like Postman or using the provided `.rest` file or Swagger if any. Create a user, log in, create a report, etc., to verify that data is indeed saved in the Postgres database.

c. Session Management & Authentication Fixes:
The app uses session-based auth (the code uses `express-session` and stores

`req.session.userId`). Integration issues often occur here if the front-end is not handling cookies right:

- The server CORS is configured with `credentials: true` for `http://localhost:5173`, which is good. This allows the browser to accept cookies from the server.
- **Front-end must send credentials**: Axios/fetch by default will **not** send cookies on cross-origin requests. We **must enable** this. In Axios, set `axios.defaults.withCredentials = true` or pass `{ withCredentials: true }` in each request that needs the session cookie [6] [7]. This is crucial – otherwise, the login may set a cookie, but subsequent requests (e.g., fetching `/laporans`) won't include it, resulting in the server thinking the user is not logged in (and thus not saving the data under a user context or outright rejecting with 401).
- For example, after a successful login, do:

  ```
  axios.defaults.withCredentials = true;
  ```

  so that all following requests automatically include the session cookie [7]. Or, on each call like:

  ```
  axios.get('http://localhost:5000/laporans', { withCredentials: true });
  ```

  Either approach works as long as the cookie is sent.
- **Check cookie options**: By default, `express-session` cookies are HttpOnly and SameSite (Lax by default). If your front-end is on a different origin (localhost:5173 vs 5000), `SameSite=Lax` might block the cookie in some cases (because the browser may consider it cross-site). Setting `cookie: { sameSite: 'none', secure: true }` would allow cross-site cookies, but then you need HTTPS for secure cookies. Since this is development, one approach is to keep both front and back on the same domain in prod (or at least use a proxy). For now, if things aren't working, consider temporarily `sameSite: 'strict'` vs `'lax'` vs `'none'`.
- Also, ensure the session store is working. The code uses `connect-session-sequelize` which stores sessions in DB. On first run, it should create a table (likely called `Sessions` by default). Check that and see if sessions are being saved.
- **Authentication endpoints**: Verify the `login` controller: it should find the user by email/username, verify password (the code uses argon2 to compare hash), then set `req.session.userId = user.uuid` (the code appears to use UUID in session). After login, the client should get a cookie named maybe `connect.sid`. Ensure the login response is handled on front-end (store user info in Redux state, perhaps).
- **The `/me` endpoint**: There is a route `router.get('/me', Me)` which likely returns current user info if session is valid. The front-end can call this (with credentials) on app load to check if the user is logged in (persisting across refresh). Make sure it returns the user details (excluding password).
- **Logout**: The `logout` route (DELETE `/logout`) should destroy the session (`req.session.destroy`). The front-end should call this and then clear local state (and maybe redirect to login).
- With sessions fixed and axios sending cookies, the integration issue of "data not saved" might be resolved, because now `req.userId` will be set in protected routes. For example, `createLaporan` uses `req.userId` (set in the `verifyUser` middleware after checking session) to associate the new report with the user. If `req.userId` was undefined (due to missing cookie), the code might have been creating records with `userId = undefined` or not at all. After fix, `req.userId` will be the logged-in user's ID and the record will save properly.

d. API Responses and Error Handling:
For better integration, make sure the backend returns clear responses that the front-end can use:

- Use JSON consistently for responses. E.g., on success of a create, return the created object or a message. On failure, return `{ error: "message" }` with appropriate status code.
- The code often uses messages in Indonesian (like "Laporan berhasil dibuat"). These are fine, but ensure the front-end knows where to display them or handle errors (perhaps show a toast on error). Alternatively, you can use English or standardized error codes for consistency.
- For debugging, you can also log the requests on server (or use a logger) to ensure the data coming in from front-end is as expected.

By solidifying the backend logic and ensuring the session/auth flows, we address the major cause of integration woes. The CRUD operations will persist data in the database reliably when invoked correctly.

3. Front-End Integration with the Express API

Now we focus on the **React front-end (Vite)** and how it communicates with the back-end API. A number of integration bugs stem from misconfigured requests or state management issues on the client side. We will ensure the front-end is properly set up to consume the API and reflect changes in the UI.

a. Base API URL and Environment Configuration:
Currently, the React code is likely using hardcoded URLs like `http://localhost:5000/...` in axios calls (as we saw in the auth slice). This works in development, but it's better to externalize this. We can use Vite environment variables for flexibility:

- Create a file `.env.development` in the frontend with `VITE_API_URL=http://localhost:5000` (and similarly `.env.production` with the prod API URL). In Vite, any variable starting with `VITE_` will be exposed via `import.meta.env`. For example, `import.meta.env.VITE_API_URL` will hold the base URL [8].
- In an axios instance or the Redux thunks, use this base URL. For instance:

```
axios.defaults.baseURL = import.meta.env.VITE_API_URL;
axios.defaults.withCredentials = true;  // as discussed, to include
cookies
```

  Do this once at app startup (maybe in `src/app/store.js` or `src/main.jsx` before rendering the app). This way, you can call relative endpoints: `axios.get('/laporans')` will actually go to `http://localhost:5000/laporans` as base, and include credentials.
- This approach allows changing the API host by just changing env files, and no need to litter the code with `'http://localhost:5000'` strings. It's cleaner and prevents mistakes.

b. State Management and Redux Toolkit:
The project uses Redux Toolkit (there's an `authSlice.js` in the code). We should leverage that for managing user auth state and possibly for other data (reports, employees, etc.).

- **Auth Flow**: When a user logs in, the `LoginUser` async thunk in `authSlice` calls the API and, on success, should store the user data in state and redirect to the dashboard. Verify that this is implemented:

- The `login` controller likely returns user info (or the front might call `/me` after login to get user info). Ensure the front-end sets `state.user` to that info and `state.isSuccess = true`.
- Also handle failure (wrong password, etc.) by showing `state.isError` and maybe an error message.
- On initial app load, if a session cookie exists, you can dispatch an action to call `/me` and populate the store (so the user stays logged in after refresh).
- **Protecting Routes**: Use React Router to protect the private pages. For example, if the user state is null (not logged in) redirect from `/dashboard` to `/login`. We can create a `<PrivateRoute>` component or use a pattern with `<Routes>`:

```
<Routes>
  <Route path="/login" element={<LoginPage />} />
  <Route
    path="/"
    element={user ? <MainLayout /> : <Navigate to="/login" />}
  >
    {/* inside MainLayout or as nested routes: */}
    <Route index element={<DashboardPage />} />
    <Route path="laporan" element={<LaporanPage />} />
    {/* ... other protected routes ... */}
  </Route>
</Routes>
```

  This ensures that if `user` is not in state (not authenticated), any attempt to hit a protected route is redirected. Protected routes are a common concept – they restrict access to authenticated users only [9].
- **Role-Based UI**: Based on `user.role` or a boolean like `user.isAdmin`, conditionally render admin features. For example, in the sidebar or menu, only show the "Employee Data" and "Upload Checklist" links if the current user is a superuser. In React, that might look like:

```
{user && user.role === 'admin' && (
  <NavLink to="/employees">Manage Employees</NavLink>
)}
```

  This way, regular users won't even see the option. Additionally, you might add another route guard for admin routes (similar idea: if user is not admin, redirect).
- **Fetching Data**: For each section (dashboard, reports, etc.), ensure the front-end fetches data from the API and handles loading states:
- **Dashboard**: call an endpoint (maybe `/dashboard` or compile data from `/laporans`) to get summary stats. Update a `dashboardSlice` or local state with this info and display it. Use effect hooks to load on component mount.
- **Reports page**: likely requires loading all reports for the logged-in user. The back-end `getLaporans` already filters by `req.userId`, so calling `GET /laporans` returns only the current user's reports (for normal user) or possibly all if superuser – the code doesn't show multi-user queries, so maybe superuser is also restricted to their own, unless extended. If superuser

should see all employees' reports, we could modify `getLaporans` to allow an admin to specify a userId or to ignore the filter.

  - Implement listing of reports, with optional filter UI for date range and type. The current API supports query params `start_date`, `end_date`, `type`, etc. Build a form or controls for the user to filter (e.g., pick a month or a date range). When applied, call the API with appropriate query (Axios can send params).
  - Show the list in a table or cards. Include an "Edit" and "Delete" button if editing is allowed (maybe a user can edit their own reports? If not, skip editing for normal user; admin might have edit rights).
  - Connect create report form to the API: after user enters details and submits, call `POST /laporans`. On success, you might dispatch an action to add it to Redux state or simply refresh the list by refetching.
  - Provide feedback for actions (e.g., "Report added successfully" message or any validation error from server like the one if `karyawanId` is missing).
- **Employee Data page (for superuser)**: fetch all users/employees. If using the same Users table, you might have an endpoint `/users` that returns all users (or all regular users). Use that to populate a list of employees. Allow superuser to create a new employee (which could involve creating a new user account with a default password, or sending an invite email, depending on requirements). The code as given doesn't clearly differentiate employees vs users, so an admin might just create a new user account via the existing registration logic. You can repurpose the registration form for admin use, or make a simplified "add employee" that just asks for name and email, then generates a password and sends it via email (since email settings exist).
  - Implement edit/delete for employee records if needed (e.g., update email, or remove an employee who left).
  - **Important:** If an admin creates a user, ensure the new user gets the default set of checklist categories (if that concept remains) by reusing that logic in `createUser`.
- **Checklist pages (Daily/Weekly/Yearly)**: For a regular user, these pages likely need to fetch the checklist content uploaded by the admin.
  - We can have an API like `GET /checklists?type=daily` returning the latest daily checklist file info or data. The front-end then either shows a link or directly displays it.
  - If the checklist is, say, a PDF or image, one approach is to provide a URL to the file (if we serve it statically as mentioned earlier). The user could click "Download" to get it, or we could embed it. For embedding a PDF, an `<iframe src={fileUrl} />` or using the `<object>` tag works in browsers (or a specialized PDF viewer component).
  - Provide an **Export to PDF** button if the requirement is that the user can export the checklist itself to PDF. However, if the checklist is already a PDF, "download" is essentially the export. The export PDF feature might be more relevant to the reports (e.g., export a summary of reports).
  - For admin, on the checklist upload page, use a file input to select a file and send it via Axios. Remember to set the request to multipart form data. With Axios, you create a `FormData` object:

    ```
    const formData = new FormData();
    formData.append('file', selectedFile);
    formData.append('type', 'daily'); // or weekly/yearly
    axios.post('/checklists', formData, { withCredentials: true,
      headers: { 'Content-Type': 'multipart/form-data' }
    });
    ```

    The backend (with Multer) will handle storing it. After a successful upload, maybe refresh the list or show a success note.
    - If multiple checklists can exist (e.g., one for each day or each year), you might list them. But likely it's one current version of each, so maybe just store/overwrite the existing one. Define that logic clearly: e.g., `ChecklistModel` might have a `type` field (daily/weekly/yearly) so you keep one row per type, updating it when a new file is uploaded (possibly also keep an `updatedAt` timestamp).
- **UI/UX Improvements**: Keep the UI clean and responsive:
- Tailwind CSS is already set up, ensure you use it to style forms, buttons, tables consistently.
- Show loading spinners or messages when data is being fetched.
- Show confirmations for destructive actions (like deleting a report or an employee).
- Use modals or separate pages for forms as appropriate (e.g., maybe the "Add Report" form is a modal on the reports page, etc., depending on the design).
- Ensure that after certain actions (add/edit/delete), the app state updates without needing a full page refresh. This is where managing state with Redux can help (e.g., dispatch an action to remove a deleted item from the list in state).

c. Testing Front-End Integration:
After implementing the above, run the app and simulate typical usage:

- Log in as a normal user: verify you can see your dashboard, list your reports, create a new report (and that it appears in the list and in the database), try exporting PDF (does it download with correct content?), view the checklists (download a sample file).
- Log out, ensure you are redirected and cannot access protected pages.
- Log in as superuser: verify you can see admin-specific links. Go to Employee Data: add a user, see it appear. Go to Checklist Upload: upload a file for "weekly checklist". Maybe log in as a normal user in another browser to see that file available.
- Try edge cases: invalid login (show error), leaving required fields blank (backend should 400, front-end should display message), etc.

By systematically verifying each feature, you can catch any remaining integration issues (for example, if something is not updating without refresh, you might need to adjust how state is updated or use a React `useEffect` to refetch data after an action, etc.).

4. Role-Based Access Control (Superuser vs Regular User)

The system distinguishes between **superusers** (who can manage data and upload checklists) and **regular employees** (who can submit reports and view information). Enforcing this distinction is critical both on the UI and security (backend).

a. Defining Roles:
The simplest approach is to add a field in the Users model, e.g., `role` or `isAdmin`. For example:

```
role: {
  type: DataTypes.STRING,
  allowNull: false,
  defaultValue: 'user'  // 'admin' for superuser accounts
}
```

Alternatively, a boolean `isSuperuser`. Given the context, probably one or a few accounts are superuser (like a manager or admin staff), others default to normal user.

- If you add this field now, and you have an initial admin user, you might set it manually in the database or modify the registration so the first user or a user created with a certain code is admin. Or simply update the DB record after creating the first user.
- Ensure that `verifyUser` middleware not only checks session but could also attach the user's role (it currently finds the user by session and sets `req.userId`; we might set `req.userRole = user.role` too for convenience).

b. Backend Authorization Middleware:
Implement middleware or checks in routes to limit access:

- Example middleware:

```
export const verifyAdmin = async (req, res, next) => {
  const user = await Users.findOne({ where: { uuid:
req.session.userId } });
  if (!user) return res.status(404).json({ error: "User not found" });
  if (user.role !== 'admin') {
    return res.status(403).json({ error: "Forbidden: Admins only" });
  }
  next();
}
```

  Use this on routes like `app.use('/users', verifyUser, verifyAdmin, userRoutes);` or on individual endpoints (e.g., the checklist upload route). This ensures even if a non-admin tries via API, they get denied [10].
- In controllers, you can also put checks if needed (e.g., in `getLaporans`, if an admin wants to fetch someone else's reports, you could allow an override if `req.userRole === 'admin'` and an ID is provided in query).

c. Front-End Enforcement:
As discussed, hide admin-only links and use route protection. On top of that, for an extra precaution, you might implement an **AdminRoute** similar to PrivateRoute:

```
<Route path="/admin" element={
  user && user.role === 'admin' ? <AdminPage/> : <Navigate to="/" />
} />
```

This way, if somehow a normal user navigates to an admin URL manually, they will be kicked out. This is in addition to the backend 403, so it's defense-in-depth.

d. Granular Permissions:
Consider what a *regular user* can do in detail: - Likely can **create/read their own reports**. Should they update or delete past reports? Possibly, but maybe not (depends on the use case – perhaps if a mistake is made, they should request admin to edit, or maybe they can edit same-day entries, etc.). Define this policy. The code currently allows update/delete of `laporans` but restricts by `userId` so a user can only affect their own data. That's fine. You could leave edit rights to users for now. - Cannot access other

users' data. The backend naturally prevents that by scoping queries to `req.userId`. If an admin needs to see all, you might adjust that when `req.userRole` is admin (e.g., ignore the userId filter or allow filtering by a query param). - Regular users **cannot create new employees or checklists** or view those management pages. We've hidden the UI, and if they call the APIs, `verifyAdmin` will block them.

Now, what can a *superuser* do: - Everything a user can, plus: - View and manage all reports (if required). If managers need to see all employees' reports for oversight, we should implement an endpoint or option for that. Perhaps on the dashboard for admin, show an overview of all reports, or allow filtering by employee in the reports page. This could be done by adding an API: e.g., `GET /laporans?user={userId}` only if admin. Or have separate admin routes. For now, focus on required ones: employee data and checklists. - CRUD on Employees (Users). - Upload/checklists maintenance.

Test the role separation: - Log in as normal user, try to navigate to an admin URL (should redirect or show nothing). Try calling an admin API via browser console fetch or so – should get 403. - Log in as admin, ensure you can do admin tasks and also do normal tasks if needed.

This role setup guarantees that superusers have the elevated privileges they need, and regular users are constrained to just their part of the system, aligning with the requirements.

5. Handling File Uploads for Daily/Weekly/Yearly Checklists

One unique feature is the **checklist uploads** by superuser and downloads by normal users. We need to implement file upload in the backend and integrate it in the front-end.

a. Backend File Upload Setup (Multer):
We will use **Multer** (Express middleware) for handling file uploads. Multer makes it easy to get files from an HTML form or FormData request and save them to disk or memory. Key steps:

- **Install Multer**: `npm install multer`. Require it in the relevant route file or controller.
- **Configure storage**: Decide where to store files. A simple approach is to use the local filesystem. For example, create an `uploads/` directory (inside the backend project). We can use `multer({ dest: 'uploads/' })` for automatic disk storage in that folder [11]. This will save files with random names. Alternatively, use `multer.diskStorage` to control filenames and destinations. For now, `'uploads/'` is fine (it will generate unique names).
- **Create upload route**: In `ChecklistRoute.js` (or a new route), add something like:

```
import multer from "multer";
const upload = multer({ dest: 'uploads/' });
router.post('/checklists', verifyUser, verifyAdmin,
upload.single('file'), async (req, res) => {
  // req.file is the uploaded file, req.body contains text fields like
type
  try {
    if (!req.file) return res.status(400).json({ error: "No file
uploaded" });
    const { type } = req.body; // e.g., "daily", "weekly", "yearly"
    // Save file info and type in database (ChecklistModel)
    const checklist = await Checklists.create({
      name: req.file.filename, // saved filename on server
      originalName: req.file.originalname,
      type: type,
      userId: req.userId  // perhaps store who uploaded
    });
    res.status(201).json({ msg: "Checklist uploaded", checklist });
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
});
```

This is a simplistic example. You might have the Checklist model with fields: `filename`, `originalName`, `type` (and maybe a `url` or something if stored remotely). Here we assume local storage, so the `filename` (which Multer generates or we can generate) is enough to retrieve the file later.

- **Serving the files**: To allow users to download or view the file, we must serve the static files. E.g., in `index.js`, add:

```
app.use('/uploads', express.static('uploads'));
```

This will serve any file in that folder via `http://server:5000/uploads/<filename>`. Make sure not to expose sensitive files – since only checklists are stored there, it should be okay. For more security, one could require auth to download, but serving static might bypass auth. If that's a concern, instead of static serving, create a route like:

```
router.get('/checklists/:id/download', verifyUser, async (req, res) => {
  // find checklist by id, ensure either admin or (if you had per-user
files) authorized
  const file = await Checklists.findByPk(req.params.id);
  if (!file) return res.status(404).send("Not found");
  res.download(path.join(__dirname, '../uploads', file.name),
file.originalName);
});
```

This uses `res.download` to let user download with original filename. Given our scenario, it might be fine to let all logged-in users access the files, since nothing highly sensitive, but it's your call. At minimum, you might restrict it to logged-in users by not exposing the URL publicly.

- **Multiple checklists**: If you want separate upload endpoints or handling for daily/weekly/yearly, you can either use the `type` field as above to distinguish, or even separate routes (`POST /checklists/daily`, etc., all using the same multer logic but possibly storing in different subfolders or tables). Using a single table with a type column is simpler. You might enforce one file per type: e.g., if a file of that type exists, overwrite it (update the DB entry and replace the file). To do that, you'd check `await Checklists.findOne({ where: { type } })` – if exists, delete the old file (fs.unlink) and update the record or create a new one replacing the old (and maybe delete the old DB entry).
- **Feedback**: After an upload, the controller returns a message or the new DB entry. The front-end can use that to confirm upload success.

b. Front-End File Upload Form:
For superusers:

- Create a page or component for uploading checklists. Perhaps a simple form with:
- A dropdown or radio to select the checklist type (Daily, Weekly, Yearly).
- A file input (`<input type="file" />`).
- A submit button.
- Use React state to handle form data (e.g., `selectedType` and `selectedFile`). On submit, construct FormData and do `axios.post('/checklists', formData)` as described earlier.
- Since we set `axios.defaults.baseURL` and `withCredentials`, the call will include the cookie (so `verifyUser` passes) and hit the correct URL.
- Handle the response: if success, maybe show a success alert like "Checklist uploaded successfully." If error, display error. Also, possibly reset the form.
- (Optional) Immediately update the list of checklists or some state if you show the current uploaded files on the UI.

For regular users:

- Create pages to view each type of checklist. The project structure shows `src/pages/Checklist/Harian.jsx`, `Mingguan.jsx`, `Bulanan.jsx` (bulanan likely means monthly, which is interesting since the user said yearly, but maybe they intended monthly and yearly similarly). Possibly the wording changed; let's assume *Bulanan* is monthly and *Tahunan* (not listed but likely intended) is yearly.
- In each of those pages, you would fetch the corresponding checklist from the backend. Could be a generic `GET /checklists?type=daily` that returns the file info (or even the file data encoded, but better to get just info or URL). If you have served the uploads statically, the response might contain the filename or full URL. For example, the backend could respond with `{ url: "http://localhost:5000/uploads/abc123.pdf", originalName: "DailyChecklist.pdf" }`.
- Then the front-end can either provide a download link:

  ```
  <a href={checklist.url} target="_blank" rel="noopener
  noreferrer">Download {checklist.originalName}</a>
  ```

  This would open it in new tab (for viewing) or allow download (you can add `download` attribute to force download).
- Alternatively, embed the PDF directly if that’s a requirement:

  ```
  <iframe src={checklist.url} width="100%" height="600px" title="Daily
  Checklist"></iframe>
  ```

  This will show the PDF in-browser (provided the browser PDF viewer is available).
- Provide an export to PDF if needed. But since the checklist itself is a PDF (in our assumption), the "export" might simply be the download action. If the checklist were some HTML content, and they wanted to export it, you could use jsPDF as well – but storing as PDF in the first place is easier for fidelity (especially if these checklists are forms or documents created by the admin in Word/PDF format).

c. Testing File Handling:
- Upload a sample file via the admin UI. Check the server `uploads/` folder to see if it saved. Also check

the database `Checklists` table for a new entry. - As a regular user, go to the daily/weekly page and see if the file is accessible (download or view). - Try edge cases: upload without selecting file (should error); upload a file that's too large or wrong format (maybe you restrict to PDF, you can set Multer fileFilter to only accept `.pdf` by checking `req.file.mimetype`). If needed, add validation:

```
const upload = multer({
  dest: 'uploads/',
  fileFilter: (req, file, cb) => {
    if (file.mimetype !== 'application/pdf') {
      return cb(new Error('Only PDF files are allowed'));
    }
    cb(null, true);
  }
});
```

Then handle that error in the route. - If multiple uploads of same type are allowed, see how the UI deals (maybe list all with dates). - Clean up: ensure that if a checklist is updated, old file is removed to save space (not strictly necessary for functionality, but for completeness).

By implementing file upload carefully, we ensure the checklist feature works smoothly: superusers can provide up-to-date forms or documents for employees, and employees can always get the latest version.

6. PDF Export Feature for Reports

The requirement mentions an **export to PDF** feature, likely so users (or admins) can get a PDF version of certain data (like a summary report, or a filled checklist, or perhaps an entire report log).

The project code includes a `/pdf` route and some front-end code (like `Pdf/Export.jsx` and `pdfGenerator.jsx`), hinting that this was in progress. We have two broad approaches to implement PDF generation:

Approach 1: Client-Side PDF Generation (using jsPDF or similar)
This means using a library in the browser to generate the PDF from data or HTML. Advantages: no need for server processing, quick feedback, and keeps data on client side. Since each user likely only exports their own data, this is a good approach.

- **jsPDF** is a popular library for this. It allows you to programmatically add text, images, etc., to a PDF. For example, you can create a PDF and add content like:

  ```
  import { jsPDF } from "jspdf";
  const doc = new jsPDF();
  doc.text("Daily Report Summary", 20, 20);  // add text at x=20, y=20
  // add more text or maybe loop through report entries adding lines
  doc.save("report.pdf");  // triggers download
  ```

  This would produce a simple PDF with given text [12] [13].

- **html2canvas + jsPDF**: If you want to export an existing React component or HTML table to PDF, one trick is to use `html2canvas` to capture it as an image and then put that image into jsPDF. Another is to use jsPDF's html plugin or a higher-level library like **html2pdf.js** that does both steps. For example, jsPDF has a `fromHTML` (in older versions) or you can use `doc.html(element, options)` in newer versions to convert HTML to PDF. The Stack Overflow discussion noted a plugin approach for older jsPDF to handle HTML content [14].
- **autoTable for tables**: If exporting tabular data (like a list of reports), the jsPDF-AutoTable plugin is very handy. It can take an array of data or even an HTML `<table>` and create a nicely formatted table in the PDF.
- **Front-end UI**: Provide a button "Export to PDF" on the relevant page (e.g., on the Reports page or Dashboard). When clicked, gather the data:
- If the data is already displayed (say a list of reports in a table), you can either re-generate it in the PDF or simply capture the HTML.
- A straightforward way: use jsPDF to create a text-based summary. For example, for each report entry, do `doc.text(` ${date} - ${description} - ${status}`, 10, y)` incrementing `y` for each line.
- Alternatively, if styling and exact format is important, you can design a hidden PDF layout (like an HTML template for the report) and then pass that to a generator.
- The code in `exportController.js` on the backend seems to prepare some data based on a period (month/year). Possibly the idea was: user picks a month, client posts to `/pdf` with that, server responds with data or a generated PDF. They might have planned to generate PDF server-side. But we can simplify by doing it in front-end.
- **Example**: A user selects "May 2025" and hits Export. The front-end could either:
- Ask the server for all reports in May 2025 (e.g., GET `/laporans?start_date=2025-05-01&end_date=2025-05-31&grouped=true` if such exists) then build PDF.
- Or if the data is already loaded/shown (maybe the UI has filtered to May 2025 already), just use that data from state.
- Use jsPDF to create a PDF with a title "Laporan Mei 2025", then for each entry add lines or table rows.
- Prompt download. This all happens in the browser instantly.
- **Testing**: Ensure the PDF content is correct (dates formatted, etc.). Also ensure multi-page handling (if too many items, jsPDF can add new pages).
- **When to consider server-side**: If you need to generate a very complex PDF or include high-quality images, or if the data size is huge, server might be better. But for typical reports (text and maybe small tables), client-side is fine.

Approach 2: Server-Side PDF Generation (using an API)
This approach would have the client call an API (like the `/pdf` route) and the server returns a PDF file (setting `Content-Type: application/pdf`). The advantage is you can use powerful Node libraries or even headless Chrome (via Puppeteer) to generate pixel-perfect PDFs. However, it's more resource-intensive on the server and adds complexity.

Given the code skeleton: - The `exportReport` controller likely intended to gather user's data for a given month (`period`) and maybe compile totals (the code maps month names to numbers, probably to parse "Mei 2025" into date range). It then prepares a `response` object with presumably some summary (maybe total income/expense if it were finance). - Possibly they planned to use a PDF library like **pdfmake** or **pdfkit** to actually generate a PDF buffer and send it. That part is not seen in the snippet. - Implementing this fully would involve writing PDF layout code on the server. For example, using **pdfkit**:

```
import PDFDocument from 'pdfkit';
// ... inside exportReport:
const doc = new PDFDocument();
// set response headers for PDF
res.setHeader('Content-disposition', 'attachment; filename="report.pdf"');
res.setHeader('Content-type', 'application/pdf');
doc.pipe(res);
// add content to doc
doc.text(`Report for ${period}`, { align: 'center' });
data.forEach(item => {
  doc.text(`${item.date} - ${item.category}: ${item.amount}`);
});
doc.end();
```

This streams the PDF to the response. The front-end would initiate a download if it sees those headers. - This is quite involved, and unless specifically needed, the client-side method might suffice. Also, server-side requires ensuring fonts, etc., are handled (if including non-latin text or special formatting). - **Hybrid**: The server could also just send back aggregated data (like totals) and the client uses that to generate PDF. In that case, the PDF route is basically duplicating what the client could do by calling existing endpoints. So it might not be necessary.

Recommendation: Use **client-side PDF generation** for simplicity. The code base already includes a React component `PdfGenerator.jsx` which likely was intended to do this. We can utilize that or write our own using jsPDF or the library of choice. This keeps the back-end simpler (we can even remove the `/pdf` endpoint if unused).

Implementing client-side PDF with jsPDF:

- Install jsPDF: `npm install jspdf`.
- In the `Export.jsx` or wherever the Export PDF UI is, import and use it.
- Example for reports:

```
import { jsPDF } from "jspdf";
import autoTable from 'jspdf-autotable';  // if you want to format
tables easily

function exportReports(reports, period) {
  const doc = new jsPDF();
  doc.setFontSize(16);
  doc.text(`Utility Reports - ${period}`, 14, 20);
  // Prepare table data
  const tableColumn = ["Date", "Category", "Description", "Status"];
  const tableRows = reports.map(r => [
    new Date(r.createdAt).toLocaleDateString(),
    r.karyawan ? r.karyawan.name : r.type,
    r.amount || "-",   // or any relevant fields
    r.status
  ]);
  autoTable(doc, { head: [tableColumn], body: tableRows, startY: 30 });
  doc.save(`Reports-${period}.pdf`);
}
```

This would create a PDF with a title and a table of reports. The `autoTable` plugin nicely handles table layout (you'll need to import it from `jspdf-autotable`). If you want to include multi-language text or specific fonts, you might need to set those up (jsPDF by default supports basic Latin fonts).

- For checklists, if needed, you could allow exporting the checklist *content* or maybe just rely on the file download. If, say, the checklist was an HTML form the user fills in the browser and then wants a PDF of their answers, that could be a scenario for jsPDF (taking their input and making a document). But since it's not explicitly stated, we focus on exporting reports.

Testing PDF export:

- Try exporting with no data vs with data.
- Check the downloaded PDF file: does it open and show the expected info? Are all columns present? If the text is too long, see if it wraps (autoTable can handle wrapping).
- If there are a lot of records, ensure it spans multiple pages. autoTable will handle page breaks automatically. If doing manual text, you'd need to check page height and call `doc.addPage()` when needed.
- Test on different browsers to ensure the download triggers properly (some might open in PDF viewer tab, which is fine).

By implementing the PDF export, we give users the ability to take the data offline or print it, fulfilling that requirement. This feature especially adds value for summary reports (monthly/yearly summaries for management, or personal record-keeping for employees).

7. Basic Project Setup and Best Practices

Finally, let's cover the **setup steps and remaining tasks** to ensure everything runs smoothly in development and production:

a. Setting Up the Development Environment:

1. **Install Dependencies:** As mentioned, run `npm install` in both `back-end` and `front-end` directories. Ensure all required packages are in package.json (e.g., express, sequelize, pg, cors, axios, react, etc., as well as dev dependencies like nodemon or vite). If any errors, install missing ones.
2. **Configure Environment Variables:** Create a `.env` file in the backend root:

```
APP_PORT=5000
DB_HOST=localhost
DB_PORT=5432
DB_NAME=report_app
DB_USER=postgres
DB_PASS=yourpassword
SESSION_SECRET=someRandomSecretValue
```

Also add any email creds if email features are to be used:

```
GMAIL_USER=yourgmail@gmail.com
GMAIL_PASS=yourapppassword  (avoid plain password if possible, use app
password for Gmail)
```

and in production, set `NODE_ENV=production` accordingly. In the front-end, create `.env.development` with `VITE_API_URL=http://localhost:5000`.

3. **Initialize Database:** Make sure PostgreSQL is running. Create the database `report_app` (or whichever name you set). You can do this with `createdb report_app` on the command line (if PostgreSQL is in PATH and configured), or via a GUI. No tables need to be created manually because Sequelize will handle it on sync.
4. **Start the Back-end server:** In development, you might have a script like `"dev": "nodemon index.js"` in package.json. Use `npm run dev` to start with auto-reload on code changes. Otherwise, `node index.js`. You should see a console log like "Server is running on port 5000". If it crashes or errors, fix those (commonly, issues in DB connection or missing module).
5. **Start the Front-end dev server:** Navigate to `front-end` directory, run `npm run dev`. Vite will output a local URL (usually `http://localhost:5173`). Open that in your browser. You should get the login page of the app (if routing is set such that unauthenticated sees login or landing page).
6. **Verify Basic Navigation:** Try to register a new user (if that path is available on UI). If not, you might create a user manually:
7. Temporarily, you could disable `verifyAdmin` on user creation or use a pre-made admin in the seed. Alternatively, directly insert into the database a user. For example, using psql or a DB admin tool, insert a row into `users` table (username, email, password hash from argon2, role). Argon2 hash can be generated using a small Node script or using the Forgot Password flow if implemented. For initial ease, maybe create an admin by modifying the code to allow a default admin creation if none exist (just an idea).
8. Once you have credentials, log in.
9. **Test End-to-End:** As described earlier, go through the app functionality. If any errors show up in console or network requests fail (check browser devtools Network tab), debug those:
10. 401 Unauthorized on API calls? Likely missing credentials or session issue.
11. 404 Not Found? Possibly wrong URL (check if the request URL matches what backend expects).
12. CORS error? Means either origin is not allowed (if you opened from a different host) or you forgot withCredentials (CORS would block response). Adjust CORS config or front-end config accordingly.
13. UI errors (undefined variables, etc.) – fix the component logic or state as needed.

b. Production Build and Deployment:

- To build the front-end for production: `npm run build` in front-end will produce a `dist` folder with static assets.
- You can serve these with a static server or integrate into Express. One way: in Express (back-end), after all API routes, add:

```
import path from 'path';
app.use(express.static(path.join(__dirname, '../front-end/dist')));
app.get('*', (req, res) => {
  res.sendFile(path.join(__dirname, '../front-end/dist/index.html'));
});
```

This will serve the React app on all unknown routes (so client-side routing works). Make sure to adjust paths if needed. Alternatively, host the front-end on a separate static hosting (Netlify, Vercel, etc.) and just ensure it calls the correct API.

- Update CORS in production if front-end is on a different domain (or allow the domain). If served from same domain (e.g., your Express serves the files), you can remove the origin restriction and just do `app.use(cors({ credentials:true, origin: sameDomain }))` or even disable CORS if same domain (since requests will be same-origin, the cookie will be sent automatically).
- Use environment vars for production API URL as well if separate.
- **Security**: In production, ensure `SESSION_SECRET` is long and random. Possibly use HTTPS such that secure cookies can be used (`secure: true, sameSite: 'none'` for cross-site if needed).
- **Emails/OTP**: The code had features for email verification and OTP (likely for password reset). If these are needed:
- Set up a real email service or Gmail SMTP with app password.
- Test the registration flow that it sends an email with OTP or link.
- This might be beyond the core ask, but since the code is there: ensure that after a user registers, an email is sent with a verification link (`EmailVerification.js`) or code. The `OTP.js` and `verifyOTP` middleware suggest maybe a two-factor or email confirmation. Integrate those pages on the front-end (maybe the Register component should trigger sending OTP and then verify).
- If time doesn't permit full implementation, you might stub these out or leave them for a later iteration, focusing on primary functionality first.

c. Clean up and Maintenance:

- Remove any leftover console.log debugging or commented code that is no longer needed (makes code cleaner).
- Write **README documentation** for the project: how to set up, how to run, what environment variables to define, etc., so that any new developer (or yourself in future) can easily start the project.
- Consider using **Postman** or similar to document the API endpoints (could list them in README too).
- Future improvements can be listed (like switching to JWT auth if needed, adding more analytics in dashboard, etc.).

Conclusion

By addressing each of the above points, we effectively **fix the integration issues and complete the project**:

- The move to PostgreSQL is completed with Sequelize's configuration adjusted [3], ensuring better alignment with production needs.
- Backend CRUD operations are now reliable – missing relations have been added (e.g., userId in Karyawan) so data persistence works correctly [5], and we've enforced session usage in API calls by configuring Axios to send cookies [6].
- The authentication system is robust, with proper session handling and role-based access control both server-side and client-side. Unauthorized access is prevented (returning 403 or redirecting as appropriate) [9].
- The React front-end is fully integrated: using environment variables for config, utilizing Redux for state, and properly fetching data from the Express API. We ensure that after each action (create/update/delete), the UI updates to reflect the current state.

- The superuser features (employee management and checklist uploads) are implemented. Superusers can upload files via an easy form, thanks to Multer handling file storage on the backend [11]. Regular users can retrieve these files and even export reports or data to PDF for convenience.
- We also went through the essential setup and deployment steps, so the team knows how to run the app and what to configure.

With these changes, the Daily Reporting System for Utility Employees should be fully functional and much easier to maintain. All daily reports will be saved in the database, the admin can monitor and provide resources (checklists), and users can interact with the system without facing the previous bugs. The code is better structured and integrated, following best practices for a full-stack JavaScript application.

---

[1] [2] [3] [4] Getting Started | Sequelize
https://sequelize.org/docs/v6/getting-started/

[5] Associations | Sequelize
https://sequelize.org/docs/v6/core-concepts/assocs/

[6] [7] Understanding Cookie Management with Axios Interception
https://www.dhiwise.com/post/managing-secure-cookies-via-axios-interceptors

[8] javascript - loading env variables in react app using vite - Stack Overflow
https://stackoverflow.com/questions/70883903/loading-env-variables-in-react-app-using-vite

[9] [10] Authentication with React Router v6: A complete guide - LogRocket Blog
https://blog.logrocket.com/authentication-react-router-v6/

[11] Express multer middleware
https://expressjs.com/en/resources/middleware/multer.html

[12] [13] How to generate PDFs in the browser with Javascript (no server needed) - Joyfill
https://joyfill.io/blog/how-to-generate-pdfs-in-the-browser-with-javascript-no-server-needed

[14] jspdf - Generate pdf from HTML in div using Javascript - Stack Overflow
https://stackoverflow.com/questions/18191893/generate-pdf-from-html-in-div-using-javascript