Awesome. I'll create a complete step-by-step plan for building your PhishNet project using GoPhish's code base for functionality, with React + Vite (frontend), Node.js (backend), and PostgreSQL (database). I'll include detailed prompts that you can use for AI models or coding assistance for each module.

Give me a moment to prepare everything for you! 🚀

**PhishNet Development Plan**

**Overview:** Build **PhishNet** by replicating GoPhish features with a clean, modular React+Vite frontend and Node.js (Express) backend using PostgreSQL. Incorporate multi-tenancy (each organization’s data isolated by a tenant/org ID), reuse GoPhish’s schema (Campaigns, Groups, Targets, Templates, Pages, SMTP, etc.) adapted for Postgres, and add HTML import/export for email templates. Follow feature-based modules for scalability ([Best Way to Structure an Express.js App: Modular vs Layered Approach Explained | Medium](https://medium.com/@branimir.ilic93/express-js-best-practices-modular-vs-layered-approach-for-medium-and-large-appsintroduction-626e61cc908d#:~:text=What%20is%20Modular%20Architecture%3F)).

**Step 1: Project Initialization & Architecture**

* **What to Build:** Set up the project scaffolding and choose tech stack. Create separate backend/ and frontend/ folders. Configure Express and React (Vite). Implement a **multi-tenant foundation**: add an organizations (or tenants) table to support multiple accounts. Use a “tenant discriminator” approach – include an organization\_id foreign key in each main table so each org’s data is separated ([Designing Your Postgres Database for Multi-tenancy | Crunchy Data Blog](https://www.crunchydata.com/blog/designing-your-postgres-database-for-multi-tenancy#:~:text=With%20this%20approach%20you%27re%20using,you%27re%20joining%20on%20that%20key)).
* **Technologies/Libraries:** Node.js, Express, Sequelize ORM (or TypeORM/Prisma) with PostgreSQL driver. React + Vite for frontend, React Router for routing, Tailwind CSS (with a UI library like Radix/UI-Components) for styling (matching PhishNet’s dark theme). JWT for auth, bcrypt for passwords.
* **Folder/File Structure:**
* /phishnet-project
* │
* ├─ backend/
* │ ├─ src/
* │ │ ├─ config/ (DB config, email config, etc.)
* │ │ ├─ models/ (Sequelize models per module)
* │ │ ├─ routes/ (Express routers per module)
* │ │ ├─ controllers/ (controller logic per module)
* │ │ ├─ services/ (business logic, email sending, etc.)
* │ │ └─ middleware/ (auth, error handling, multi-tenant checks)
* │ ├─ app.js (Express app)
* │ └─ package.json
* │
* ├─ frontend/
* │ ├─ src/
* │ │ ├─ components/ (shared UI components, e.g. Sidebar, Header, Buttons)
* │ │ ├─ pages/ (React pages: Dashboard, Login, Campaigns, etc.)
* │ │ ├─ hooks/ (custom hooks, e.g. useAuth)
* │ │ ├─ api/ (axios API service)
* │ │ └─ assets/ (images, logo, etc.)
* │ ├─ vite.config.js
* │ ├─ package.json
* │ └─ tailwind.config.js
* │
* └─ docker-compose.yml (optional for Postgres, Node)
* **APIs/Endpoints:** At this stage, bootstrap basic endpoints:
  + GET /api/status or root to check server health.
  + GET /api/organizations (list organizations – for testing multi-tenancy).
  + Later: setup auth routes (/api/auth/login, etc.).
* **Database Tables/Models:** Define organizations table with fields: id (PK), name, created\_at, updated\_at. Ensure other tables will have organization\_id referencing organizations(id).
* **Frontend Components:** None specific yet; just prepare the Vite React app with routing. Implement a global layout component (Sidebar, Header) to match PhishNet’s style. Use provided components (e.g. Header.jsx, Sidebar.jsx) as references.
* **Example Prompt:**
  + *Backend:* “Generate an Express project with Sequelize and PostgreSQL. Create a model for Organization with fields id (auto-increment), name, createdAt, updatedAt. Initialize an Express router for organizations.”
  + *Frontend:* “Generate a Vite React app structure. Set up React Router with a login route and a protected dashboard route. Include Tailwind CSS configuration.”

**Step 2: Database Schema Design & Multi-Tenancy**

* **What to Build:** Design the PostgreSQL schema based on GoPhish. Create tables for Users, Groups, Targets, SMTP\_Profiles, Email\_Templates, Landing\_Pages, Campaigns, Results, Events, etc., adding organization\_id FK to each. Apply proper constraints (PK, FKs, unique indexes). For example, a **Campaigns** table references smtp\_profile\_id, email\_template\_id, landing\_page\_id, and created\_by\_user\_id. Add tenant\_id (alias organization\_id) to each table to segregate data ([Designing Your Postgres Database for Multi-tenancy | Crunchy Data Blog](https://www.crunchydata.com/blog/designing-your-postgres-database-for-multi-tenancy#:~:text=With%20this%20approach%20you%27re%20using,you%27re%20joining%20on%20that%20key)).
* **Technologies/Libraries:** Sequelize (models and migrations) or an equivalent ORM. Use migrations to apply schema changes.
* **Folder/File Structure:** Under backend/src/models/, create files per model (e.g. User.js, Organization.js, Group.js, Target.js, EmailTemplate.js, LandingPage.js, SmtpProfile.js, Campaign.js, Result.js, Event.js). Each model includes fields and associations.
* **APIs/Endpoints:** No new endpoints yet; this step focuses on DB. However, can create /api/migrate or use sequelize-cli scripts to run migrations.
* **Database Tables/Models:**
  + **organizations**: id (PK), name, timestamps.
  + **users**: id, first\_name, last\_name, email (unique per org), password\_hash, is\_admin (bool), organization\_id (FK→organizations.id), timestamps.
  + **groups**: id, name, description, organization\_id (FK), timestamps.
  + **targets** (email recipients): id, group\_id (FK), first\_name, last\_name, email, position, timestamps.
  + **smtp\_profiles**: id, name, host, port, username, password, from\_name, from\_email, organization\_id, timestamps.
  + **email\_templates**: id, name, subject, html\_content, text\_content, organization\_id, timestamps.
  + **landing\_pages**: id, name, html\_content, redirect\_url, organization\_id, timestamps.
  + **campaigns**: id, name, smtp\_profile\_id (FK), email\_template\_id (FK), landing\_page\_id (FK), send\_date, status (Scheduled, Sent, etc.), organization\_id, created\_by\_user\_id (FK), timestamps.
  + **results/events** (phishing feedback): results table with id, campaign\_id (FK), target\_id (FK), email, sent\_at, opened\_at, clicked\_at, submitted\_at, etc. Alternatively, separate **events** table logging each action (open, click, submit) with timestamp.
  + Add **Constraints**: use foreign keys and ON DELETE CASCADE where appropriate. Example: campaigns → organization\_id FK, targets → group\_id FK, etc.
* **Frontend Components:** No UI yet, but create TypeScript/JavaScript interfaces or Redux slices (if using state management) matching these models.
* **Example Prompt:**
  + “Generate a Sequelize migration for an organizations table. Include columns: id (primary key), name (string), createdAt, updatedAt.”
  + “Write a Sequelize model definition for EmailTemplate with fields: id, name, subject, htmlContent, textContent, organizationId (with foreign key constraint), createdAt, updatedAt.”

**Step 3: Authentication & User Management Module**

* **What to Build:** Implement user registration, login, and password reset. Build the **Users** module so each user is linked to an organization. Include JWT-based authentication and role checks (e.g. admin vs regular user). Add password reset via token emailed to user.
* **Technologies/Libraries:** Express routers, Sequelize models (User), bcrypt (password hashing), jsonwebtoken for JWT, nodemailer (or an email service) for sending reset links. Use an email template for reset instructions.
* **Folder/File Structure:**
* backend/
* src/
* modules/
* user/
* user.model.js (Sequelize User model)
* user.controller.js
* user.service.js
* user.routes.js
* user.routes.js
* auth.middleware.js
* emailService.js
* **APIs/Endpoints:**
  + POST /api/auth/register: Create a new user (requires organization\_id and role).
  + POST /api/auth/login: Accept email+password, respond with JWT.
  + POST /api/auth/forgot-password: Generate a reset token, save in password\_reset\_tokens table, and send email.
  + POST /api/auth/reset-password: Validate token and update password (prevent reuse of the same password).
  + GET /api/users: List users (admin only, filter by organization).
  + POST /api/users: Create user (admin only).
  + PUT /api/users/:id: Update user (admin or self).
  + DELETE /api/users/:id: Remove user (admin).
* **Database Tables/Models:**
  + **users** table as above.
  + **password\_reset\_tokens**: id, user\_id (FK), token, expires\_at. Foreign key to users.id.
* **Frontend Components:**
  + **Login Screen:** Email, password fields.
  + **Registration Screen:** For creating first user or organization admin. Possibly combined with organization setup.
  + **Forgot Password / Reset Password Pages:** As seen in the existing PhishNet UI (like ResetPassword.jsx) – fields for new password and confirmation. Use visual feedback for password strength (reuse code similar to [16]).
  + **User Management Page:** List of users in the organization, buttons to create/edit/delete. Use cards or tables styled with Tailwind (resembling the PhishNet existing style).
* **Example Prompt:**
  + “Write an Express route for POST /api/auth/login that checks user credentials, uses bcrypt to compare password hash, and returns a JWT token upon success.”
  + “Generate a React component for a Login form (email and password inputs) that calls the /api/auth/login endpoint and handles JWT storage.”

**Step 4: Organization & Multi-Tenancy Handling**

* **What to Build:** Ensure multi-tenancy support throughout. Add an **Organization** (or Account) management module. When a new organization signs up, create its entry. All queries must filter by organization\_id. Optionally implement middleware to extract organization\_id from JWT or user’s profile to enforce tenant-scoped data access ([Designing Your Postgres Database for Multi-tenancy | Crunchy Data Blog](https://www.crunchydata.com/blog/designing-your-postgres-database-for-multi-tenancy#:~:text=With%20this%20approach%20you%27re%20using,you%27re%20joining%20on%20that%20key)).
* **Technologies/Libraries:** Express middleware to attach req.organizationId based on the authenticated user’s JWT claims. Sequelize scopes (optional) can enforce organization\_id in queries.
* **Folder/File Structure:** Can continue under modules/user or create a separate modules/organization.
* **APIs/Endpoints:**
  + POST /api/organizations: (Admin use) Create a new organization.
  + GET /api/organizations/:id: Get organization details.
  + (Alternatively, organizations created implicitly when a user registers under a new org).
* **Database Tables/Models:** Already have **organizations** from Step 1. Ensure relationships: e.g., a User belongsTo Organization; other tables belongTo Organization.
* **Frontend Components:**
  + **Organization Setup Page**: (Shown on first login if no org exists) to create an org name.
  + **Switch Organization** (if multi-org access is allowed per user).
  + **Organizations Management Page:** Only for super-admin; list of orgs, details, etc.
* **Example Prompt:**
  + “Implement an Express middleware authorizeTenant that reads organizationId from the JWT payload and attaches it to req so controllers use it for queries.”
  + “Generate a Sequelize scope on models so that queries automatically filter by organization\_id.”

**Step 5: Groups (Target Lists) Module**

* **What to Build:** Build the **Groups** module to manage email recipient lists. A Group contains multiple Targets (recipients). Include CSV import for adding many targets. Each group belongs to an organization and is created/edited by a user.
* **Technologies/Libraries:** Sequelize models for Group and Target. Use a CSV parsing library (e.g., papaparse or csv-parse) for bulk import.
* **Folder/File Structure:**
* backend/
* src/
* modules/
* groups/
* group.model.js (Group with organization\_id)
* target.model.js (Target with group\_id)
* group.controller.js
* group.service.js
* group.routes.js
* **APIs/Endpoints:**
  + GET /api/groups: List all groups (filter by org).
  + POST /api/groups: Create a new group (with name, description).
  + PUT /api/groups/:id: Update group.
  + DELETE /api/groups/:id: Delete group (cascade delete targets).
  + GET /api/groups/:id/targets: List targets in a group.
  + POST /api/groups/:id/targets: Add a new target to group.
  + DELETE /api/targets/:id: Delete a target.
  + **CSV import:** POST /api/groups/:id/import (multipart/form-data CSV file) – parse CSV rows and insert targets.
* **Database Tables/Models:**
  + **groups** (as above).
  + **targets**: id, group\_id (FK), first\_name, last\_name, email, position, timestamps. Add unique constraint on (group\_id, email) to avoid duplicates.
* **Frontend Components:**
  + **Groups List Page:** Table of groups, buttons to add/edit/delete.
  + **Group Details Page:** When selecting a group, show list of targets. Provide form to add one target and an “Import CSV” button.
  + **Import Targets Dialog:** A modal to upload CSV file. Show success/error message after import.
  + Use components like tables, form inputs, file upload controls. Style similar to existing PhishNet (light text on dark background).
* **Example Prompt:**
  + “Write a Sequelize model Target with fields: id, firstName, lastName, email, position, groupId (foreign key). Ensure email is validated and unique per group.”
  + “Generate a React component GroupList that fetches /api/groups and displays them in a table, with buttons for Create/Edit/Delete.”

**Step 6: SMTP Profiles Module**

* **What to Build:** Build **SMTP Profiles** to store sending configurations (SMTP host, port, credentials, etc.). Each organization can have multiple SMTP profiles. Encrypt sensitive fields (password) in the DB or use environment variables / secrets vault.
* **Technologies/Libraries:** Sequelize model for SmtpProfile. Use crypto or a library to encrypt/decrypt credentials (or rely on DB encryption).
* **Folder/File Structure:**
* backend/
* src/
* modules/
* smtp/
* smtpProfile.model.js
* smtpProfile.controller.js
* smtpProfile.routes.js
* **APIs/Endpoints:**
  + GET /api/smtp\_profiles: List SMTP configs (org-specific).
  + POST /api/smtp\_profiles: Create new profile (name, host, port, user, pass, from name/email).
  + PUT /api/smtp\_profiles/:id: Update profile.
  + DELETE /api/smtp\_profiles/:id: Delete.
* **Database Tables/Models:**
  + **smtp\_profiles**: id, name, host, port, username, password\_encrypted, from\_name, from\_email, organization\_id, timestamps.
* **Frontend Components:**
  + **SMTP Profiles Page:** Table of profiles with Edit/Delete actions.
  + **Add/Edit SMTP Form:** Form fields for host, port, username, password, from name/email.
* **Example Prompt:**
  + “Write an Express route for POST /api/smtp\_profiles that saves an SMTP profile (encrypt the password before saving) using Sequelize.”
  + “Generate a React form component SmtpProfileForm with controlled inputs for host, port, user, and password, and a submit handler calling the API.”

**Step 7: Email Templates Module (with Import/Export)**

* **What to Build:** Build **Email Templates** feature. Templates include a subject, HTML body, and plain-text body. Add the new requirement: **Import/Export** HTML. Users should be able to import an HTML file to create a template, or export an existing template’s HTML to a file.
* **Technologies/Libraries:** Sequelize model for EmailTemplate. Use a rich-text or code editor for HTML content on frontend (e.g. a WYSIWYG or simple textarea). File handling (browse/import HTML file, trigger download).
* **Folder/File Structure:**
* backend/
* src/
* modules/
* emailTemplate/
* emailTemplate.model.js
* emailTemplate.controller.js
* emailTemplate.routes.js
* **APIs/Endpoints:**
  + GET /api/templates: List templates.
  + POST /api/templates: Create a template (send name, subject, htmlContent, textContent).
  + PUT /api/templates/:id: Update.
  + DELETE /api/templates/:id: Delete.
  + **Import:** POST /api/templates/import – accept multipart/form-data with an HTML file, parse content and create a template record.
  + **Export:** GET /api/templates/:id/export – respond with the HTML content as a downloadable .html file. Set proper Content-Disposition.
* **Database Tables/Models:**
  + **email\_templates** (as above). Ensure name is unique per org.
* **Frontend Components:**
  + **Templates List Page:** List templates with options to edit/delete, plus buttons “Import HTML”, “Export” for each.
  + **Template Form Page:** Fields for Name, Subject, HTML (using a <textarea> or HTML editor component), and auto-generated or entered plain text.
  + **Import Dialog:** File input to upload HTML. On submit, call /api/templates/import.
  + **Export Action:** A button that triggers downloading HTML (by opening /api/templates/:id/export).
* **Example Prompt:**
  + “Write an Express endpoint for POST /api/templates/import that reads an uploaded HTML file (using multer), extracts its content, and creates a new template in the database.”
  + “Generate a React component TemplateEditor with inputs for subject and HTML content, and buttons to import/export the HTML.”

**Step 8: Landing Pages Module**

* **What to Build:** Build **Landing Pages** module for phishing landing page content. Each page has a name, HTML content (and optionally a redirect URL after submission).
* **Technologies/Libraries:** Sequelize model for LandingPage.
* **Folder/File Structure:**
* backend/
* src/
* modules/
* landingPage/
* landingPage.model.js
* landingPage.controller.js
* landingPage.routes.js
* **APIs/Endpoints:**
  + GET /api/pages: List pages.
  + POST /api/pages: Create page (name, htmlContent, redirectUrl).
  + PUT /api/pages/:id: Update.
  + DELETE /api/pages/:id: Delete.
* **Database Tables/Models:**
  + **landing\_pages**: id, name, html\_content, redirect\_url, organization\_id, timestamps.
* **Frontend Components:**
  + **Landing Pages List:** Table of pages with Edit/Delete.
  + **Landing Page Form:** Form to enter name, HTML (maybe a code editor or textarea). Include a preview (if feasible) to show what recipients will see.
* **Example Prompt:**
  + “Write a Sequelize model LandingPage with fields: id, name, htmlContent (TEXT), redirectUrl, organizationId.”
  + “Generate a React component LandingPageForm to create/edit landing pages, including an HTML textarea and preview pane.”

**Step 9: Campaigns Module**

* **What to Build:** Build the **Campaigns** module that ties everything together. A campaign uses a selected group of targets, an email template, a landing page, and an SMTP profile. Implement scheduling: campaigns can be sent immediately or at a future date. On launch, send emails to all targets via the selected SMTP profile. Track results (opened, clicked, submitted).
* **Technologies/Libraries:** Sequelize model for Campaign. Use a job queue or scheduler (e.g. node-cron, Bull/Redis) to send emails asynchronously. For sending, use nodemailer with the chosen SMTP profile. Record each email send and event.
* **Folder/File Structure:**
* backend/
* src/
* modules/
* campaign/
* campaign.model.js
* campaign.controller.js
* campaign.service.js (handles email sending logic)
* campaign.routes.js
* emailSender.js (utility to send emails)
* **APIs/Endpoints:**
  + GET /api/campaigns: List campaigns (with filters by status/org).
  + POST /api/campaigns: Create and schedule a new campaign (data includes groupId, smtpProfileId, templateId, pageId, sendDate).
  + PUT /api/campaigns/:id: Update or cancel scheduled campaign before sending.
  + GET /api/campaigns/:id/report: Get campaign results/metrics.
  + POST /api/campaigns/:id/launch: Immediately launch/send campaign.
* **Database Tables/Models:**
  + **campaigns** (as above). Add status field (e.g. "Draft", "Scheduled", "Running", "Completed").
  + **events**: If using an events table, columns: id, campaign\_id (FK), target\_id (FK), email, event\_type (opened, clicked, submitted), timestamp. Or track in a **results** table as columns (see Step 2).
* **Frontend Components:**
  + **Campaign List Page:** Table of campaigns with status (Draft, Scheduled, Running, Completed), and actions (Edit, Delete, Launch Now, View Report).
  + **Campaign Form Page:** Form to create a campaign: select Group (dropdown), Template, Landing Page, SMTP Profile, and schedule/send date. Validate inputs.
  + **Launch Dialog:** Confirmation to immediately launch.
  + **Reports Screen:** Link from each campaign to view results (see Step 10).
* **Example Prompt:**
  + “Write a Sequelize model Campaign with fields: id, name, groupId, smtpProfileId, templateId, pageId, sendDate, status, organizationId, createdBy, createdAt, updatedAt.”
  + “Generate a Node.js service that, given a campaign, retrieves its targets and uses Nodemailer to send personalized emails. Ensure to schedule it via a job queue.”

**Step 10: Reporting Module**

* **What to Build:** Implement **Reporting** to track campaign metrics. For each campaign, record events when an email is sent, when it is opened (tracking pixel), when links are clicked, and when landing page form is submitted. Generate summary stats (counts and percentages of opens, clicks, submissions). Provide export (e.g. CSV) for raw data.
* **Technologies/Libraries:** Sequelize models for Results/Events. Track opens via an image <img> request endpoint (GET /api/events/open?campaignId=...&targetId=...), clicks via redirect URL (GET /api/events/click?params). Use Express routes or middleware to log these.
* **Folder/File Structure:**
* backend/
* src/
* modules/
* report/
* result.model.js (or event.model.js)
* report.controller.js
* report.routes.js
* **APIs/Endpoints:**
  + GET /api/reports/campaigns/:campaignId: Return aggregated stats (total sent, opened, clicked, submitted, open rate, click rate).
  + GET /api/reports/campaigns/:campaignId/export: Download CSV of each target’s results.
  + **Tracking:**
    - GET /api/track/open: Endpoint hit by an embedded tracking pixel. Expects query params for campaign and target; logs an “open” event in DB, then returns a 1x1 transparent image.
    - GET /api/track/click: Endpoint for clickable URLs. Redirects to the actual target URL after logging a “click” event.
    - POST /api/track/submit: (Optional) Landing page form submission can POST to this endpoint to record a “submitted” event and then redirect.
* **Database Tables/Models:**
  + **results** or **events**: If using separate tables, ensure each has organization\_id (for multi-tenancy) and references campaign and target.
* **Frontend Components:**
  + **Campaign Report Page:** Show charts or summary (e.g. progress bars) of open/click rates. Use tables for lists of users who clicked/submitted. Provide a button to export detailed CSV.
* **Example Prompt:**
  + “Generate an Express GET endpoint /api/track/open that logs an email-open event (insert into events table) and returns a transparent GIF.”
  + “Write a React page CampaignReport that fetches /api/reports/campaigns/:id and displays a summary of total sent, opened, clicked, submitted.”

**Step 11: Frontend Integration and Styling**

* **What to Build:** Integrate all backend APIs with the React frontend, following the existing PhishNet UI style. Use global components (Header, Sidebar, Buttons, Forms) to ensure a consistent look. Ensure authentication flows protect routes.
* **Technologies/Libraries:** React Router for client routing, Axios for API calls, Zustand/Redux for state (optional), React Query (optional for data fetching caching). Tailwind CSS for styling.
* **Folder/File Structure:** Use the frontend/src/pages/ directory for each screen (e.g. Login.jsx, Dashboard.jsx, Campaigns.jsx, Groups.jsx, Templates.jsx, LandingPages.jsx, SmtpProfiles.jsx, Users.jsx, Reports.jsx, etc.). Place reusable components in frontend/src/components/.
* **APIs/Endpoints:** The frontend will call the endpoints defined above. For example, the Login form posts to /api/auth/login, the Groups page fetches /api/groups, etc.
* **Database Tables/Models:** N/A (frontend calls backend).
* **Frontend Components:**
  + **Global Layout:** A sidebar with navigation links (Dashboard, Campaigns, Templates, Landing Pages, SMTP Profiles, Groups, Users, Reports, Settings), and a top header with org/user info (similar to the PhishNet reference). Use Tailwind to match dark mode colors seen in sample. For instance, dark background bg-[#131313], orange accent color text-orange-500 (as used in [16] for password fields).
  + **Dashboard:** Show high-level stats (total campaigns, total users, quick links). Use cards.
  + **Forms and Tables:** For each module, use consistent form inputs and table designs. Use the provided UI component library (e.g., <Input>, <Button>, <Card> from user code).
  + **Responsiveness:** Ensure the UI is responsive for different screen sizes, like the existing PhishNet design.
* **Example Prompt:**
  + “Generate a React Router setup with protected routes for /dashboard, /campaigns, etc., that checks for a valid JWT token.”
  + “Write a React component CampaignForm using existing <Input> and <Select> UI components to create a new campaign, fetching dropdown options from API (groups, templates, etc.).”

**Step 12: Finalization & Testing**

* **What to Build:** Test each feature end-to-end. Set up environment variables (.env) for sensitive config (DB URL, JWT secret, SMTP credentials for sending test emails). Configure CORS for React→API. Add error handling and input validation (e.g. using express-validator).
* **Technologies/Libraries:** Jest or Mocha for unit tests (optional), Postman/Insomnia for manual API testing, React Testing Library for frontend (optional). ESLint/Prettier for code style. Docker/Docker-compose to containerize Postgres and Node for easier deployment.
* **Folder/File Structure:** Add .env.example files in both backend and frontend.
* **APIs/Endpoints:** No new endpoints. Ensure all are documented (Swagger or README with routes).
* **Database Tables/Models:** Ensure all migrations are complete and sync. Possibly create seed scripts for initial data (admin user, default SMTP if needed).
* **Frontend Components:** Final UI polish: tooltips, loading spinners for API calls, form validation messages. Implement notifications (toasts) for successes/errors (like react-toastify as seen in user code).
* **Example Prompt:**
  + “Write an Express middleware for error handling that returns JSON error messages and logs the error.”
  + “Generate a Postman collection (or describe steps) to test user registration, login, and campaign creation endpoints.”

**Modularity and scalability** are ensured by the feature-based folder structure ([Best Way to Structure an Express.js App: Modular vs Layered Approach Explained | Medium](https://medium.com/@branimir.ilic93/express-js-best-practices-modular-vs-layered-approach-for-medium-and-large-appsintroduction-626e61cc908d#:~:text=What%20is%20Modular%20Architecture%3F)) and clear separation of concerns. Each module’s API, service, and model are self-contained. The **multi-tenant design** isolates each organization’s data via an organization\_id on all tables ([Designing Your Postgres Database for Multi-tenancy | Crunchy Data Blog](https://www.crunchydata.com/blog/designing-your-postgres-database-for-multi-tenancy#:~:text=With%20this%20approach%20you%27re%20using,you%27re%20joining%20on%20that%20key)), simplifying access control. The new HTML import/export feature for templates is integrated seamlessly with existing email template management. This plan provides a step-by-step guide to build PhishNet with clean architecture and GoPhish compatibility.