Republic of the Philippines
POLYTECHNIC UNIVERSITY OF THE PHILIPPINES
College of Computer and Information Sciences
Sta. Mesa Manila

In Partial fulfilment of the requirements in course

COMP 016
WEB DEVELOPMENT

FINAL PROJECT

BSCS 3-1N | | M/TH 12:00PM-03:00PM/12:00PM-02:00 PM

Submitted by:

Name	Student Number
Arrey, Ceferino A.	2022-09047-MN-0
Flores Jr., Mark Achiles G.	2022-09016-MN-0
Guiang, Isaeus B.	2022-08884-MN-0
Reyes, Rainier Joshua D.	2022-09129-MN-0
Vicente, Jedric Knight G.	2022-09372-MN-0

Submitted to:

MR. SEVERINO M. BEDIS JR.
Subject Instructor

Date Submitted:
June 19, 2025

Republic of the Philippines
POLYTECHNIC UNIVERSITY OF THE PHILIPPINES
College of Computer and Information Sciences
Sta. Mesa Manila

I. TITLE

Kontak: A Shared Contact Management Application with Role-Based Access Control

II. INTRODUCTION

- **Project Background**

a. Overview

Kontak is a Laravel-based web application developed to streamline the way teams and organizations manage and share contact information. The application provides a centralized system for storing and organizing contacts, allowing users to access essential details efficiently and securely. A key feature of Kontak is its implementation of role-based access control (RBAC), which ensures that users have permission only to view or modify the contacts relevant to their designated roles, such as Admin, Editor, or Viewer.

The project addresses the common problem of securely managing shared contact data within a team environment, eliminating the risks and inefficiencies associated with scattered spreadsheets or manual record-keeping. Users can create, update, search, and filter contact records, while administrators can assign roles and monitor user activity for added security.

The main objectives of the project were to build a secure, user-friendly, and collaborative contact management platform using Laravel's powerful features. This includes applying Laravel's authentication system, middleware, Eloquent ORM, and database migrations to ensure the application is scalable and maintainable.

b. Context and Motivation

Contact management systems are essential tools used by individuals, teams, and organizations to store, organize, and access personal or professional contact information efficiently. These systems are especially valuable in collaborative environments where multiple users need access to a shared list of contacts for communication, coordination, or client management. Unlike storing contact details in spreadsheets or personal devices, a contact management system centralizes data in a secure and structured way, often supporting features like search, categorization, access permissions, and activity logging.

The decision to build a contact management application like Kontak was driven by the need for a simple yet secure platform where users can manage and share contact data with appropriate access controls. In many organizations, sensitive contact information must be handled with care, granting access only to authorized personnel and preventing unauthorized changes or data exposure. By integrating Role-Based Access Control (RBAC), Kontak ensures that users only perform actions permitted by their role, enhancing both security and usability.

This type of application offers several potential benefits. It improves team productivity by making important contact information easily accessible, reduces duplication and inconsistency, and strengthens data security through structured permissions. Additionally, building Kontak provided an opportunity to apply core Laravel development concepts such as authentication, authorization, and database management in a real-world scenario, making it both a practical tool and a valuable learning experience.

c. Relevance of Laravel

Laravel was chosen as the framework for building Kontak because of its powerful features, modern architecture, and developer-friendly environment, all of which made it an ideal choice for developing a secure and scalable contact management system. One of Laravel's key strengths is its built-in support for authentication and authorization, which made implementing Role-Based Access Control (RBAC) efficient and reliable. Laravel also integrates seamlessly with community packages like spatie/laravel-permission, which simplifies the process of assigning roles and permissions to users which is an essential component of this project.

Another major reason for choosing Laravel is its focus on security. Features such as input validation, hashed passwords, and secure authentication guards helped ensure that sensitive contact data in Kontak remains protected against common web vulnerabilities. Laravel also promotes the use of best practices through its well-structured MVC (Model-View-Controller) architecture, making it easier to maintain clean and organized code.

In terms of development speed, Laravel offers a wide range of tools and features—such as artisan commands, migrations, Eloquent ORM, and blade templating that significantly accelerate the development process. These tools allowed for rapid prototyping and consistent implementation of CRUD operations, user management, and data handling.

Basically, Laravel provided the right balance of security, scalability, and development efficiency, making it the perfect framework for building a role-based contact management application like Kontak.

III. OBJECTIVES

- To develop a Laravel-based contact management system that allows users to create, read, update, and delete (CRUD) contact records efficiently.
- To implement Role-Based Access Control (RBAC) by integrating Laravel's built-in authorization tools, enabling users to have different levels of access (e.g., Admin, Editor, Viewer) based on assigned roles.
- To demonstrate the practical use of Laravel's MVC architecture, migrations, Eloquent ORM, and Blade templating in a real-world application.
- To host the application on a scalable cloud environment using Amazon EC2, ensuring availability and performance in a production setting.
- To design a user-friendly interface that simplifies navigation and makes managing contacts intuitive for all user types.

IV. METHODOLOGY

- **Laravel Framework**
    a. Laravel (v5.16.0)
- **Technology Stack**
    a. **PHP (v8.2.22)** - Core server-side language that powers the entire Laravel application.
    b. **MySQL (v8.0.39)** - Primary database storing all users, contacts, and contact books data.
    c. **Laravel Eloquent ORM** - Database abstraction layer handling all the model relationships and queries.
    d. **Laravel Authentication** - Built-in user management system powering login, registration, and access control.
    e. **Blade Templates** - Laravel's templating engine renders all the views and layouts.

f. **Tailwind CSS** - Utility-first CSS framework providing all styling and responsive design.
g. **Amazon EC2** - Cloud server hosting the Laravel application in production.
h. **Amazon RDS** - Managed MySQL database service hosting the production database.
i. **Vite** - Modern build tool compiling and serving the CSS/JS assets with hot reloading.
j. **Composer (v2.7.7)** - PHP dependency manager installing and autoloading all Laravel packages.
k. **Laravel UI** - Authentication scaffolding package providing login/register views and styling.
l. **Axios** - JavaScript HTTP client handling AJAX requests in the settings forms.

- **Development Process**
  - **a. Database Design**

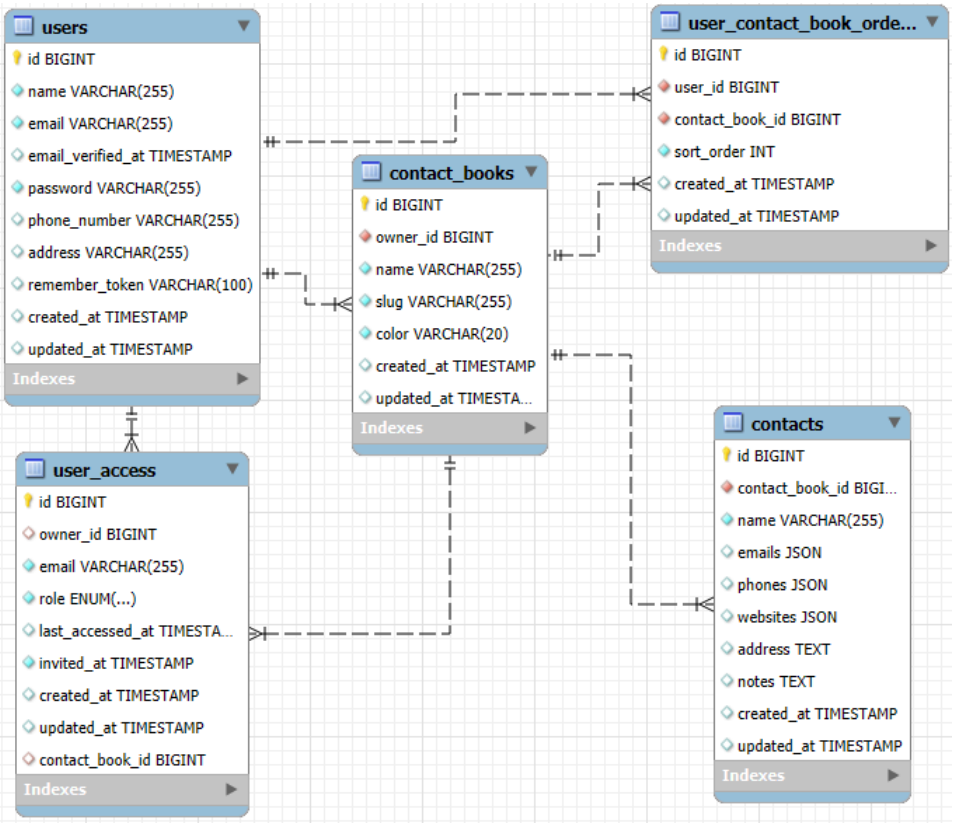


Figure 1. Kontak Schema

i. While designing, we began by defining the user entity's parameters for authentication, including name, email, and password. We added the phone number and address for additional information about a specific user.
ii. Next, we designed the contact book entity to contain the numerous contacts and connect them to a user. Each contact book has a name, a color, and a slug used to access the specific contact book. We chose a slug instead of the contact book name to refer to the contact book because slugs don't change, and we can ensure that they're unique. Additionally, slugs are randomly generated, which makes it harder for bad actors to try and guess the route or URL to a specific contact book.
iii. Next, we designed the pivot table user_access to finally connect the users to the contact books. All users have their own personal contact book by default, which they can share with others, but they also have access to other contact books that are shared with them. They can either be an admin (with full CRUD) or an audience (with view-only access).
iv. Lastly, we designed the contacts to contain the information that the users need to keep track of. Each contact may have one or more phone number, email, and/or website. They can also be given an address, and the people with access to the contact book can set and read notes about these contacts.
v. The extra table "user_contact_book_order" is just a table that helps us persist the order in which users arrange their contact books in the sidebar.

b. Model Creation

i. While implementing the Laravel models, we began by extending the User model from Laravel's built-in Authenticatable class to leverage Laravel's authentication system. We defined the fillable attributes as name, email, password, phone_number, and address, with automatic password hashing through Laravel's casting system. The model includes multiple relationship methods like ownedContactBooks(), accessibleContactBooks(), and complex query methods for retrieving contact books with custom ordering.
ii. Next, we implemented the ContactBook model with fillable attributes for name, color, owner_id, and slug. The model defines

a belongsTo relationship with the User model for ownership, and hasMany relationships for both contacts and user access records. We implemented custom methods like canBeAccessedBy() and canBeEditedBy() to handle permission logic directly in the model.

iii. The UserAccess model serves as our pivot table connecting users to contact books with role-based permissions. It includes fillable attributes for contact_book_id, email, and role, with a belongsTo relationship to ContactBook and a custom relationship to User based on email matching. This model handles the shared access logic between users and contact books.

iv. The Contact model was designed to handle flexible contact information storage using JSON casting for arrays. The fillable attributes include contact_book_id, name, phones, emails, websites, address, and notes, with the multiple contact methods (phones, emails, websites) cast as arrays in the database. We implemented custom accessors like getInitialsAttribute(), getPrimaryEmailAttribute(), and helper methods like addPhone(), addEmail() to manipulate the JSON data structure.

v. Finally, the UserContactBookOrder model serves as a simple persistence layer for user interface preferences. It contains fillable attributes for user_id, contact_book_id, and sort_order, with belongsTo relationships to both User and ContactBook models. This model enables users to customize their sidebar contact book ordering while maintaining the data across sessions.

vi. Each model leverages Laravel's Eloquent features like automatic timestamp management, relationship eager loading, and custom attribute casting to create a robust and maintainable data layer for the contact management system.

c. Controller Implementation

i. While implementing the Laravel controllers, we began by utilizing Laravel's built-in authentication controllers from the Auth namespace, including LoginController, RegisterController, ForgotPasswordController, and others. These controllers extend Laravel's base authentication classes and handle user registration, login, password resets, and email verification with minimal custom code, leveraging Laravel UI's authentication scaffolding.

ii. Next, we implemented the SettingsController as a comprehensive user management hub with constructor middleware to ensure authentication. The controller includes methods like updateProfile(), updatePassword(), and updateContactBook(), each with robust validation rules and dual response handling for both traditional form submissions and AJAX requests. We implemented permission checking using custom model methods like canEditContactBook() to ensure users can only modify contact books they own.

iii. The ContactController follows Laravel's resource controller pattern with full CRUD operations for contact management. It includes index(), create(), store(), show(), edit(), update(), and destroy() methods, each protected by authentication middleware. The controller handles complex form data for multiple phone numbers, emails, and websites by processing arrays and storing them as JSON in the database through Eloquent model methods.

iv. We designed the ContactBookOrderController as a specialized controller handling the drag-and-drop reordering functionality. The updateOrder() method processes arrays of contact book IDs and uses the UserContactBookOrder model to persist user preferences for sidebar organization.

v. The HomeController serves as the main dashboard controller, implementing complex logic to retrieve and display contact books with proper access control. It uses Eloquent relationships and custom model methods to fetch contacts from multiple contact books that the authenticated user can access, handling both owned and shared contact books seamlessly.

vi. Each controller leverages Laravel's dependency injection for automatic model binding, uses the Auth facade for user authentication checks, implements comprehensive validation with custom error messages, and returns appropriate responses for both web and AJAX requests. The controllers also utilize Laravel's redirect system with flash messages for user feedback and maintain clean separation of concerns by delegating business logic to model methods rather than handling it directly in the controller layer.

d. Blade Template Design

i. While implementing the Blade templating system, we began by creating a master layout template (layouts/app.blade.php) that serves as the foundation for all pages. The layout includes responsive HTML structure with TailwindCSS classes, dynamic meta tags using Laravel's csrf_token() helper, and conditional rendering with @auth and @guest directives. We implemented a sophisticated toast notification system using @if statements to display session flash messages with dismissible JavaScript functionality.

ii. Next, we designed the navigation component (layouts/navigation.blade.php) as a reusable sidebar that dynamically renders contact books using Blade's @foreach loops and model relationships. The navigation includes collapsible functionality with CSS classes, user avatar initials generation, and conditional menu items based on user permissions. We utilized Blade's @include directive to modularize the navigation and keep the main layout clean.

iii. The authentication templates leverage Laravel UI's Blade structure while maintaining consistency with our TailwindCSS design system. Templates like auth/login.blade.php and auth/register.blade.php use Blade's form helpers, @csrf tokens for security, and @error directives to display validation messages inline with form fields. We implemented responsive form layouts with proper error handling and success message display.

iv. Our settings interface (settings/index.blade.php) demonstrates advanced Blade templating with complex form handling, including dynamic user access management tables. We used @forelse loops to handle empty states gracefully, Blade's @method directive for HTTP method spoofing in forms, and conditional rendering based on user permissions. The template includes modular partials like settings/partials/user-access-row.blade.php for maintaining clean, reusable code.

v. The dashboard and contact management templates utilize Blade's data binding capabilities to display complex relational data from Eloquent models. We implemented search functionality with form handling, contact card layouts using @foreach loops, and

dynamic content rendering based on contact data availability. The templates include modal systems for editing contacts with pre-populated form fields using Blade's value binding.

vi. Throughout all templates, we leveraged Blade's security features like automatic XSS protection, @csrf token inclusion, and safe output rendering. We implemented conditional styling using Blade's @if statements for dynamic CSS classes, route generation with Laravel's route() helper for maintainable URLs, and localization readiness with proper text structure. The templating system maintains clean separation between presentation logic and business logic while providing a responsive, accessible user interface that adapts to different screen sizes and user permissions.

e. RBAC Implementation

i. While implementing the RBAC system, we began by designing a three-tier permission structure through the UserAccess model, where users can have different roles: owner (full control), admin (full CRUD access), or audience (view-only access). The UserAccess model serves as our permission pivot table with fillable attributes including contact_book_id, email, and role, creating a flexible many-to-many relationship between users and contact books with role assignment.

ii. Next, we implemented permission logic directly in the User model through custom methods like canAccessContactBook() and canEditContactBook(). These methods check if a user is either the owner of a contact book or has been granted specific access through the UserAccess relationship. The canEditContactBook() method specifically validates that users have either ownership or admin-level permissions before allowing modifications, while canAccessContactBook() permits both admin and audience roles to view content.

iii. The ContactBook model includes complementary permission methods like canBeAccessedBy() and canBeEditedBy() that

work in tandem with the User model methods. These methods query the UserAccess relationships to determine if a specific user has the required permissions, creating a bidirectional permission

checking system that can be called from either the user or contact book perspective.

iv. Our controller-level authorization is implemented throughout the application, particularly in the SettingsController where we perform permission checks before executing sensitive operations. Methods like updateContactBook(), addUserAccess(), and removeUserAccess() all include canEditContactBook() validation with proper error responses for both AJAX and traditional requests. We return 403 HTTP status codes for unauthorized access attempts and provide user-friendly error messages.

v. The middleware and route protection ensures that only authenticated users can access protected resources, while the permission methods provide granular control within those authenticated routes. We implemented validation rules in forms to prevent users from assigning themselves permissions or creating duplicate access records, maintaining data integrity and security.

vi. The user interface respects the RBAC system through Blade template conditionals that show or hide functionality based on user permissions. Edit buttons, delete options, and administrative features are conditionally rendered using @if statements that call the permission methods, ensuring the UI reflects the user's actual capabilities. The system also handles edge cases like preventing contact book owners from removing their own access and maintaining at least one owner per contact book.

vii. Throughout the implementation, we leveraged Laravel's built-in authentication system as the foundation, extending it with our custom RBAC layer that operates seamlessly with Eloquent relationships and provides both programmatic and user-interface-level access control for comprehensive security coverage.

f. Routing

i. While implementing the routing system, we began by leveraging Laravel's built-in authentication routes through the Auth::routes() method, which automatically registers all necessary authentication endpoints including login, register, logout, password reset, and email verification. We complemented this with a custom root route that intelligently redirects authenticated

users to the dashboard and guests to the login page using Laravel's Auth::check() helper.

ii. Next, we implemented middleware-protected route groups using the auth middleware to ensure all sensitive operations require user authentication. We organized routes into logical groups with the Route::middleware('auth')->group() pattern, creating a clear security boundary around protected functionality while keeping guest routes separate and accessible.

iii. The settings routes demonstrate our modular routing approach using nested route groups with both prefix('settings') and name('settings.') for clean URL structure and named route generation. We implemented RESTful routing patterns with methods like Route::get(), Route::put(), Route::post(), and Route::delete() for different CRUD operations, including specialized routes like bulkRemoveUserAccess for batch operations and updateOrder for drag-and-drop functionality.

iv. Our contact management routing follows Laravel's resource controller conventions with a dedicated route group using prefix('contacts') and name('contacts.'). We implemented the complete RESTful resource pattern including index, create, store, show, edit, update, and destroy routes, leveraging Laravel's automatic route model binding by using {contact} parameters that automatically inject Contact model instances into controller methods.

v. The specialized functionality routes include the ContactBookOrderController's updateOrder route using Route::put() for handling AJAX-based drag-and-drop reordering. We implemented this as a separate endpoint rather than embedding it in the resource routes to maintain clear separation of concerns and enable specific AJAX handling.

vi. Throughout the routing implementation, we utilized named routes consistently with descriptive names like settings.update-profile, contacts.store, and contact-books.update-order. This approach enables maintainable URL generation in Blade templates using Laravel's route() helper provides flexibility for future URL structure changes without breaking existing links.

vii. The routing system also demonstrates HTTP method diversity with appropriate verb usage: GET for displaying forms and data,

POST for creating new resources, PUT for updates, and DELETE for removal operations. We implemented route model binding extensively, allowing Laravel to automatically resolve model instances from route parameters, reducing controller boilerplate and ensuring consistent model retrieval patterns across the application.

- **Justification and Alternate Approaches**
    a. **Laravel Eloquent ORM (vs Doctrine ORM, Raw SQL, Query Builder) Why Chosen**: Eloquent's Active Record pattern perfectly matched our contact management needs with intuitive relationship definitions (hasMany, belongsTo). The automatic handling of complex relationships like allAccessibleContactBooksOrdered() and JSON casting for contact arrays would require significantly more boilerplate in Doctrine. Raw SQL would sacrifice Laravel's security features and relationship management, while Query Builder lacks the model-based abstraction essential for our RBAC system.
    b. **Laravel Authentication (vs Custom Auth, Laravel Passport, Auth0) Why Chosen**: The built-in authentication system provided exactly what we needed - simple session-based auth with password resets and email verification. Custom auth would require reinventing security features like password hashing and CSRF protection. Laravel Passport is overkill for a web application that doesn't need API tokens, and Auth0 would add unnecessary third-party dependency and cost for a straightforward user management system.
    c. **Blade Templates (vs Twig, Vue.js/React, Pure PHP) Why Chosen**: Blade's seamless Laravel integration allowed us to leverage features like @csrf, @auth, and automatic XSS protection without configuration. Vue/React would require API development and complex state management for our form-heavy application. Twig lacks Laravel-specific helpers, and pure PHP templates would sacrifice security features and readability that Blade provides out-of-the-box.
    d. **Tailwind CSS (vs Bootstrap, Custom CSS, Bulma) Why Chosen**: Tailwind's utility-first approach enabled rapid prototyping and consistent design without fighting framework opinions. Bootstrap's component-based approach would conflict with our custom contact card layouts and sidebar design. Custom CSS would require maintaining a large stylesheet, while Tailwind's purging keeps our bundle small. The

utility classes make responsive design intuitive with lg:ml-64 and similar responsive prefixes.

e. **Amazon EC2 (vs DigitalOcean, Heroku, Shared Hosting) Why Chosen**: EC2 provides the flexibility needed for PHP 8.2 and Laravel's specific requirements while maintaining cost control. Heroku's pricing becomes expensive at scale and has PHP limitations. DigitalOcean would work but lacks the integrated ecosystem we get with RDS. Shared hosting typically doesn't support modern PHP versions, Composer, or command-line access needed for Laravel deployment.
f. **Amazon RDS (vs Self-managed MySQL, MongoDB, PostgreSQL) Why Chosen**: RDS eliminates database maintenance overhead while providing automatic backups, scaling, and security patches crucial for production. Self-managed MySQL would require significant DevOps work for a contact management app. MongoDB doesn't fit our relational data structure with users, contact books, and permissions. PostgreSQL would work but MySQL's Laravel integration is more mature and our data doesn't require PostgreSQL's advanced features.
g. **Vite (vs Laravel Mix/Webpack, Gulp, Parcel) Why Chosen**: Vite's lightning-fast hot module replacement dramatically improved development speed compared to Laravel Mix's slower Webpack compilation. The native ES modules support and optimized build process perfectly complement our TailwindCSS workflow. Gulp would require manual configuration for modern JavaScript features, while Parcel lacks Laravel-specific plugins. Vite's integration with Laravel is officially supported and future-proof.
h. **Composer (vs Manual Dependencies, PEAR) Why Chosen**: Composer is the de facto standard for PHP dependency management with no viable alternatives. Manual dependency management would be unmaintainable for Laravel's extensive package ecosystem. PEAR is legacy technology that doesn't support modern PHP namespacing or version constraints. Composer's autoloading, version resolution, and security advisories are essential for modern PHP development.
i. **Laravel UI (vs Custom Auth Views, Breeze, Jetstream) Why Chosen**: Laravel UI provided the exact level of authentication scaffolding needed without unnecessary complexity. Custom views would require reinventing form validation, password confirmation, and responsive design. Breeze includes modern stack features we don't need (like Alpine.js), while Jetstream is overkill with team management and

two-factor authentication. Laravel UI gave us clean, customizable Bootstrap-compatible views that integrate perfectly with Tailwind.

j. **Axios (vs Fetch API, jQuery, XMLHttpRequest) Why Chosen**: Axios provides automatic JSON parsing, request/response interceptors, and error handling that our AJAX-heavy settings forms require. The Fetch API lacks built-in JSON handling and has poor error handling for HTTP status codes. jQuery would add unnecessary bloat for just AJAX functionality, while XMLHttpRequest requires an extensive boilerplate. Axios's Laravel integration with automatic CSRF token handling makes form submissions seamless.

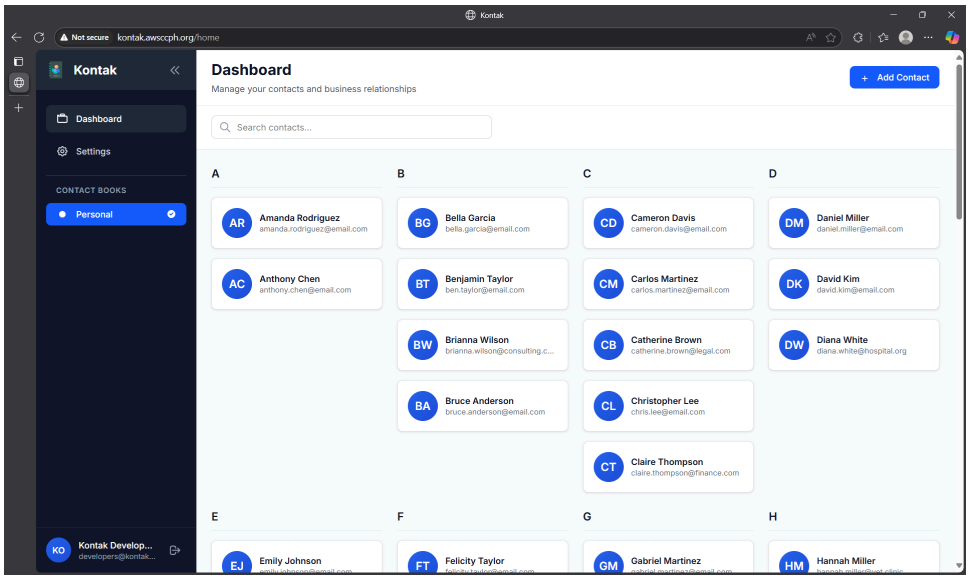
V. EVALUATION PLAN

- **Testing**
  a. **Functional Testing**: The testing was done mostly with manual user testing. We divided our group into developers and quality assurance engineers. Our QA engineers were in charge of clearly defining the functions of each module or feature in our application. After development, the newly added feature gets tested by our QA engineers.
  b. **RBAC Testing**: To make the process of testing easier, we seeded a few users in the database. This helped us expedite functional testing, especially RBAC testing, because we can easily switch between different users with different roles to confirm the effects.
  c. **Usability Testing**: For usability testing, we made sure that all buttons are clickable for both desktop and mobile view. We also made sure that the font size is appropriate for both desktop and mobile view.

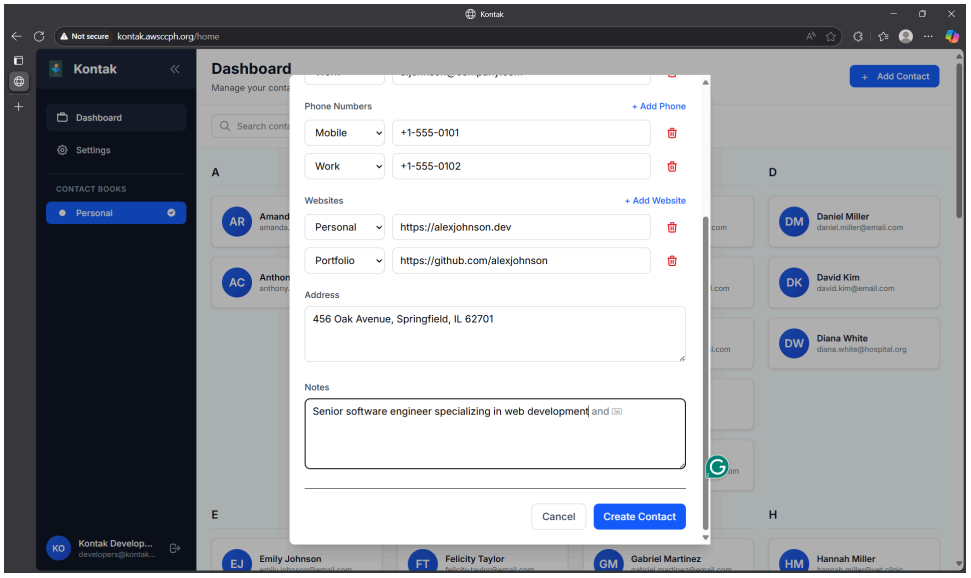
- **Future Enhancements**
  a. Add the ability for users to upload pictures for their contacts.
  b. Add the ability for the user to create more than 1 contact book because currently, they are limited to just having their personal book. This means that there is only 1 contact book per user in the application. If we have 10 users, there are only 10 contact books.

VI. SCREENSHOTS OF THE RESULT

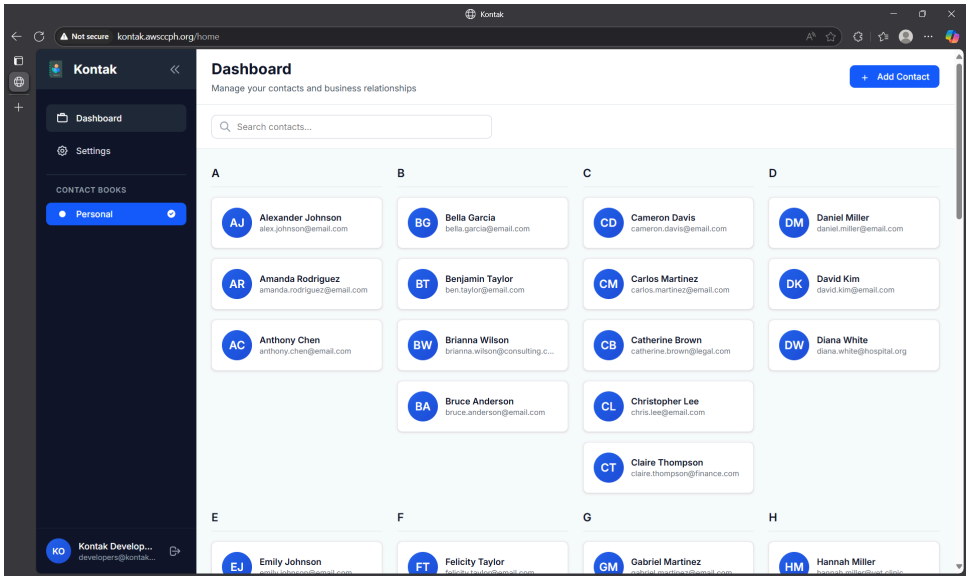
- Main Dashboard or landing page
  The interface below will be displayed after logging in.



- Screenshots of the CRUD interfaces (create, read, update, delete).
  - Create
    The interface below shows when the "+ Add Contact" button is clicked. This button popups a window where one can fill up the necessary details needed when creating a new contact.

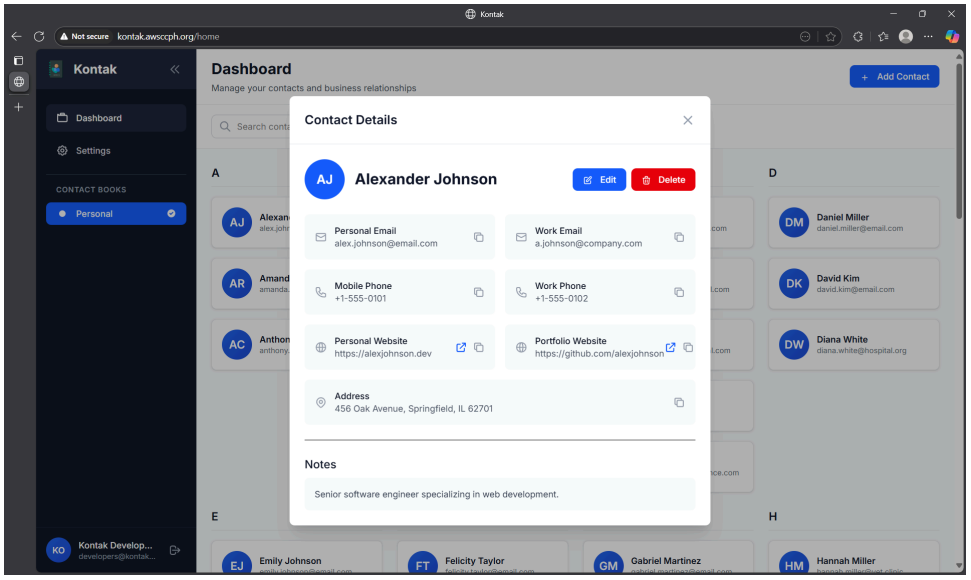


After filling up the details, it will then be added, just like “Alexander Johnson” here in the image below.

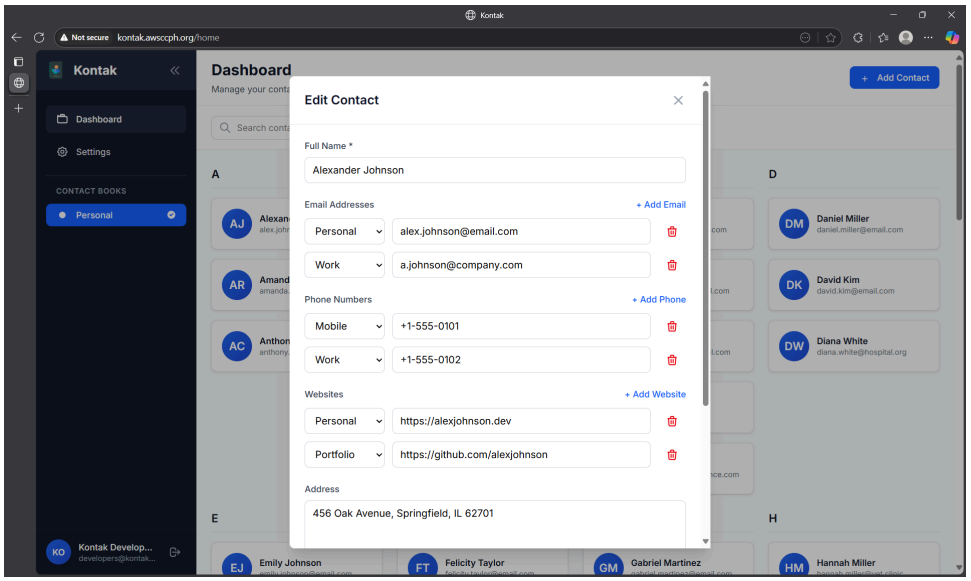


- Read

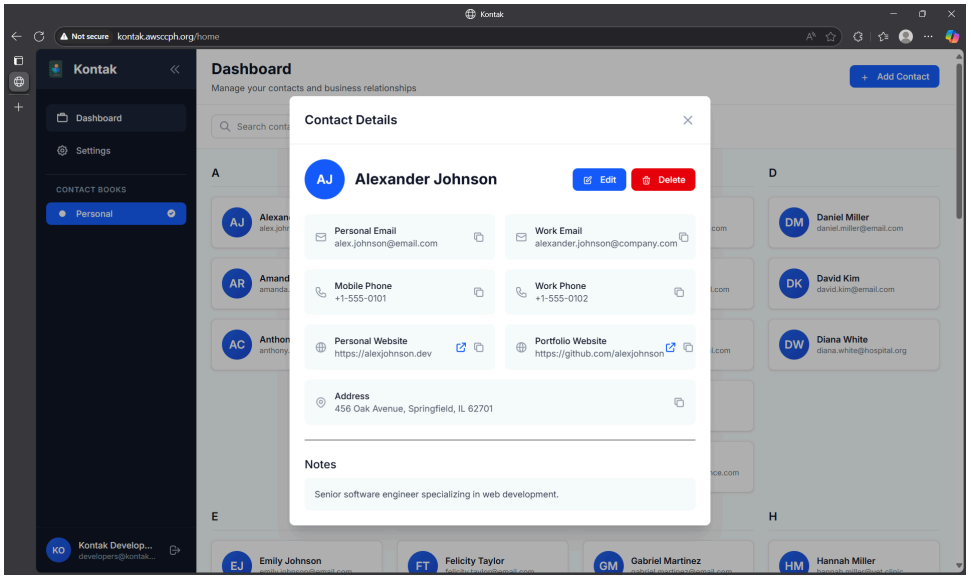
  The newly added contact can be viewed through clicking the contact.



- Update
  After clicking a contact, an edit button can be clicked to edit a contact. In here, we will edit the work email of "Alexander Johnson".

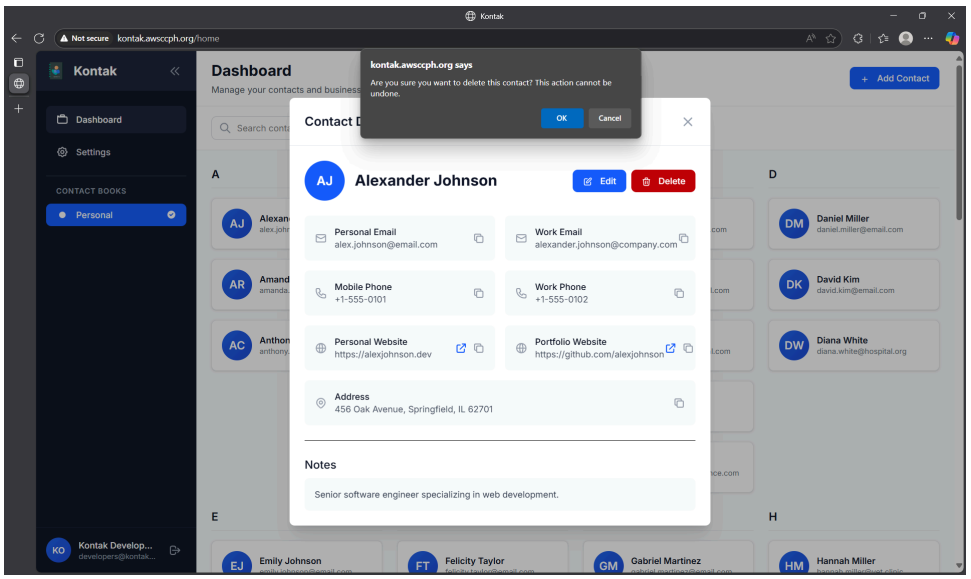


After editing, from a.johnson@company.com to alexander.johnson@company.com, it will then be reflected after clicking the save button.

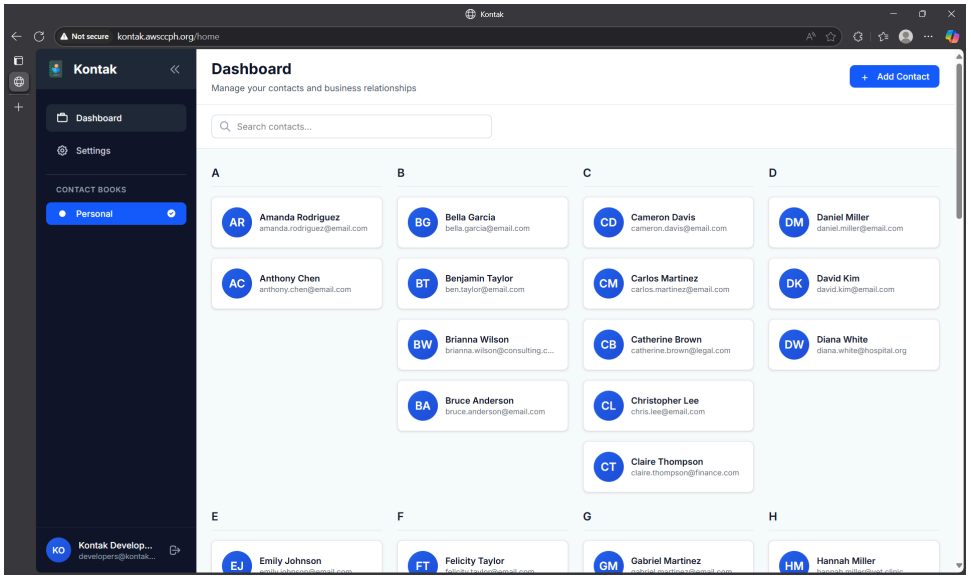


- Delete

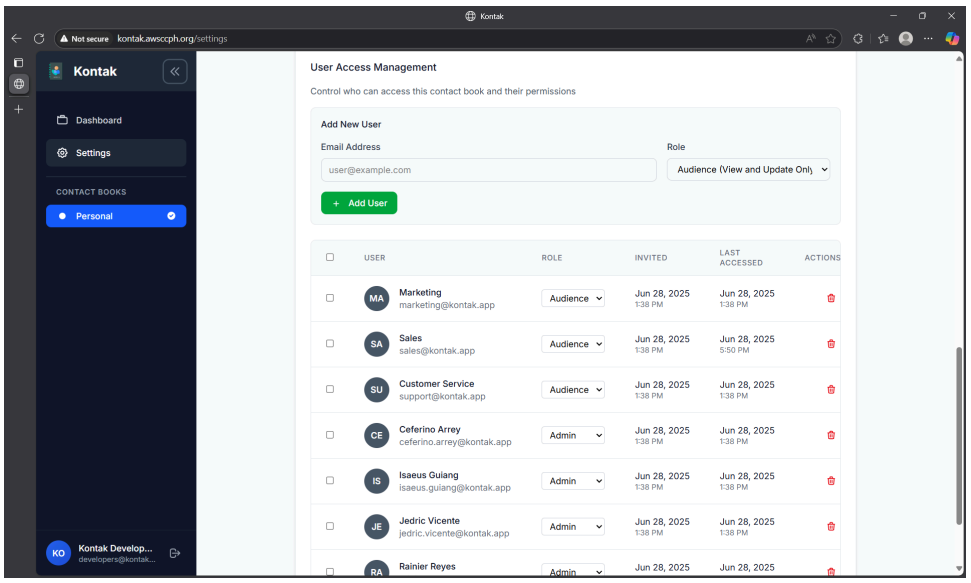
  After clicking a contact, a delete button can be clicked to delete a contact. In here, a popup window asking if you are sure to delete the said contact.



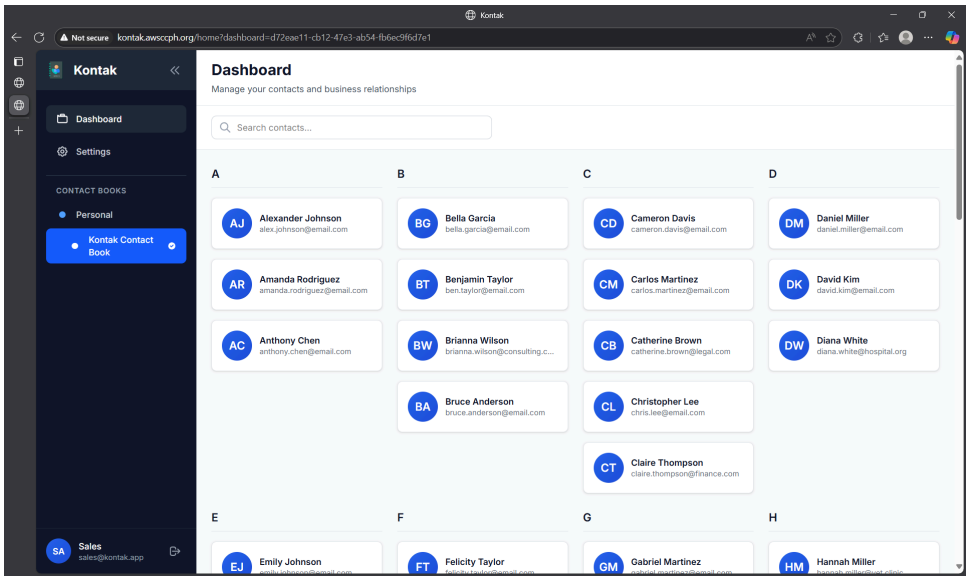
After deleting the said contact will be deleted forever, and will not be seen anymore to the dashboard.



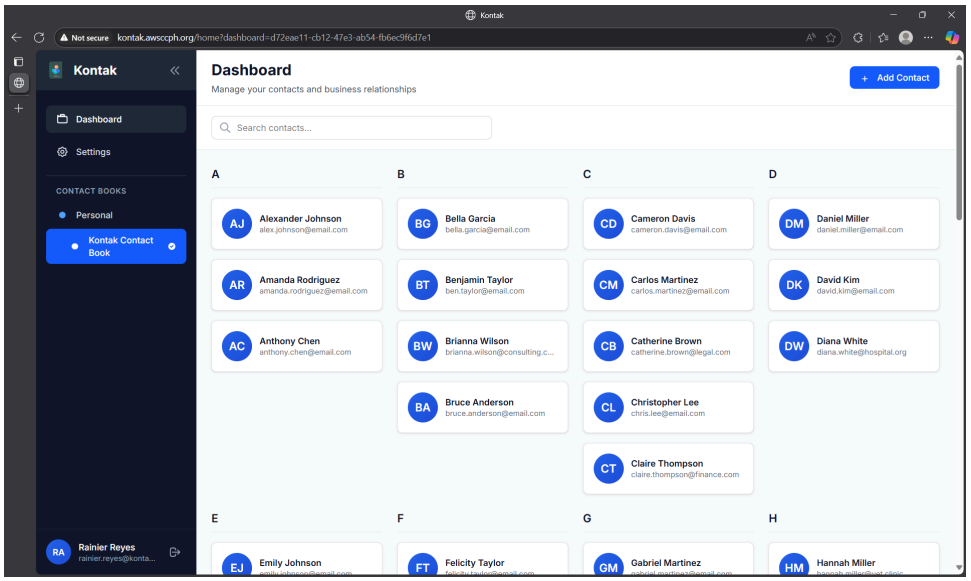
- Screenshots of the user management interface (if you implemented one).
    - Each owner has their own “User Access Management” for their own shared personal contacts. Admin could create, read, update, and delete contacts. On the other hand, the audience can only read and update.



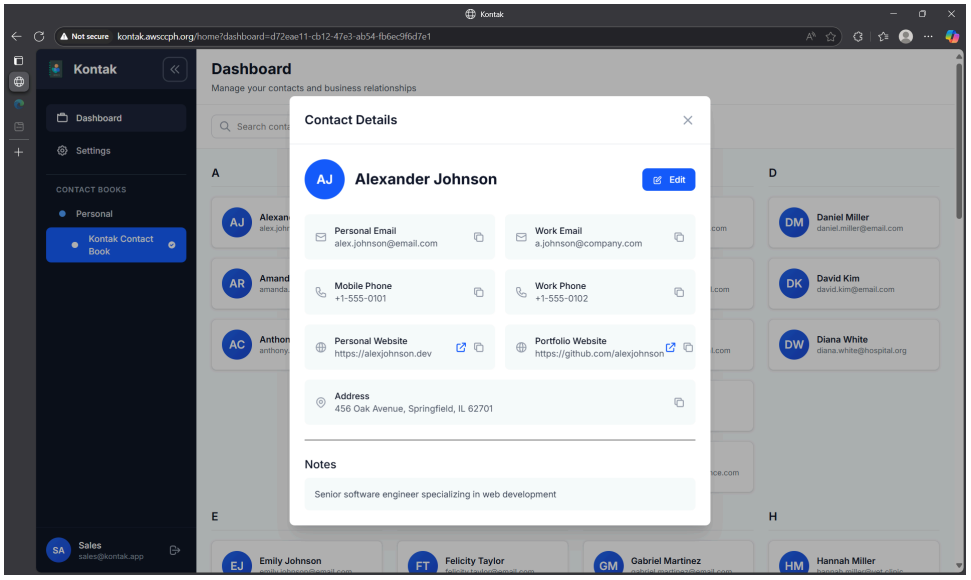
- Screenshots demonstrating RBAC in action (e.g., a user with limited permissions seeing a restricted view).
    - Audience/User (Sales) - Dashboard
      The audience or user only has the capability of read and update, therefore there is no “+ Add Contact” button in the upper right.



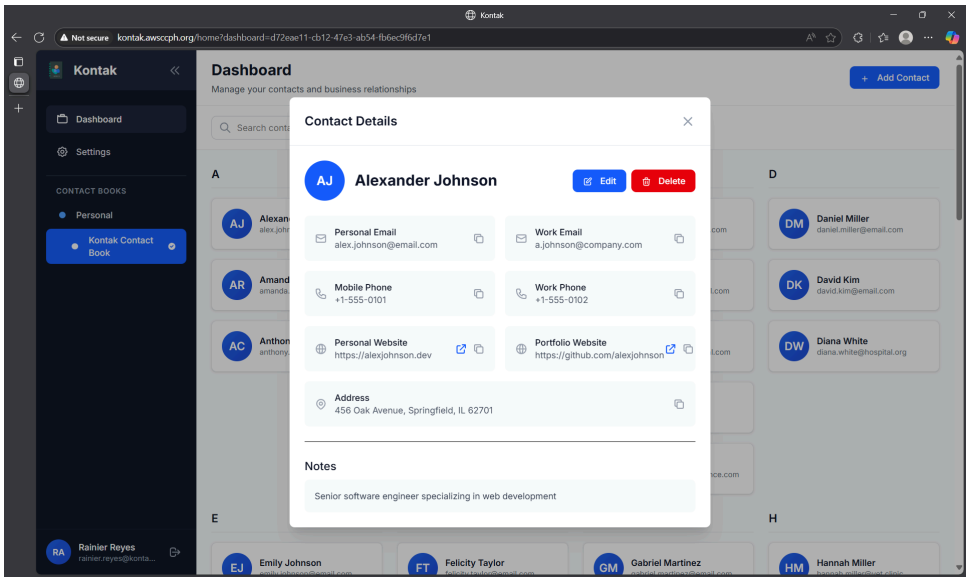
- Admin (Rainier) - Dashboard
  The admin has also the same capability to add new contact as the owner, therefore the add button is present in their interface.



- Audience/User (Sales) - Read and Update
  The audience or user only has the capability of read and update, therefore there is no “Delete” button.



- Admin (Rainier) - Read, Update, and Delete
  The admin can perform activities such as the owner. Therefore, the edit and delete button is present.



VII. SOURCE CODES

- Google Drive Link:
  - Group6-Kontak-Source_Code