Republic of the Philippines

Region V Bicol

BICOL UNIVERSITY

Polangui, Albay

Computer Studies Department

Lab Submission for Week 12

(Phase 2 and Phase 3)

ProFolio

Where you let your work shine and watch opportunities follow

Member	Role
Ofracio, Aran Joshua P.	Project Manager
Dolon, Ella Mae R.	Frontend Developer
Lumayad, Victor Ronel P.	Backend Developer
Dela Cruz, Paulina C.	Database Manager
Odoño, Alona P.	GitHub Manager
Noble, Noelissa S.	Documentation Officer

BSIT 2A

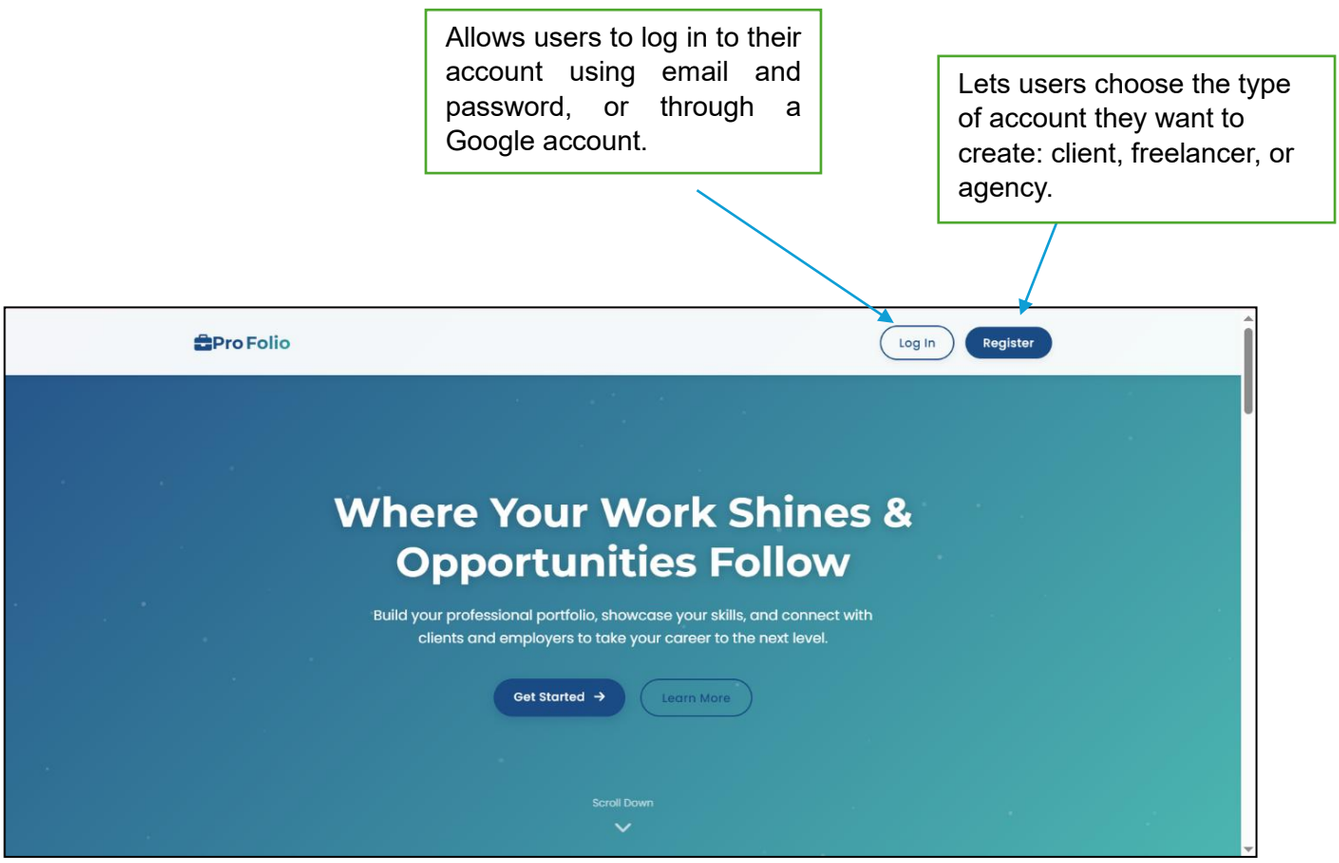
Dr. Guillermo V. Red, Jr.

Course Professor

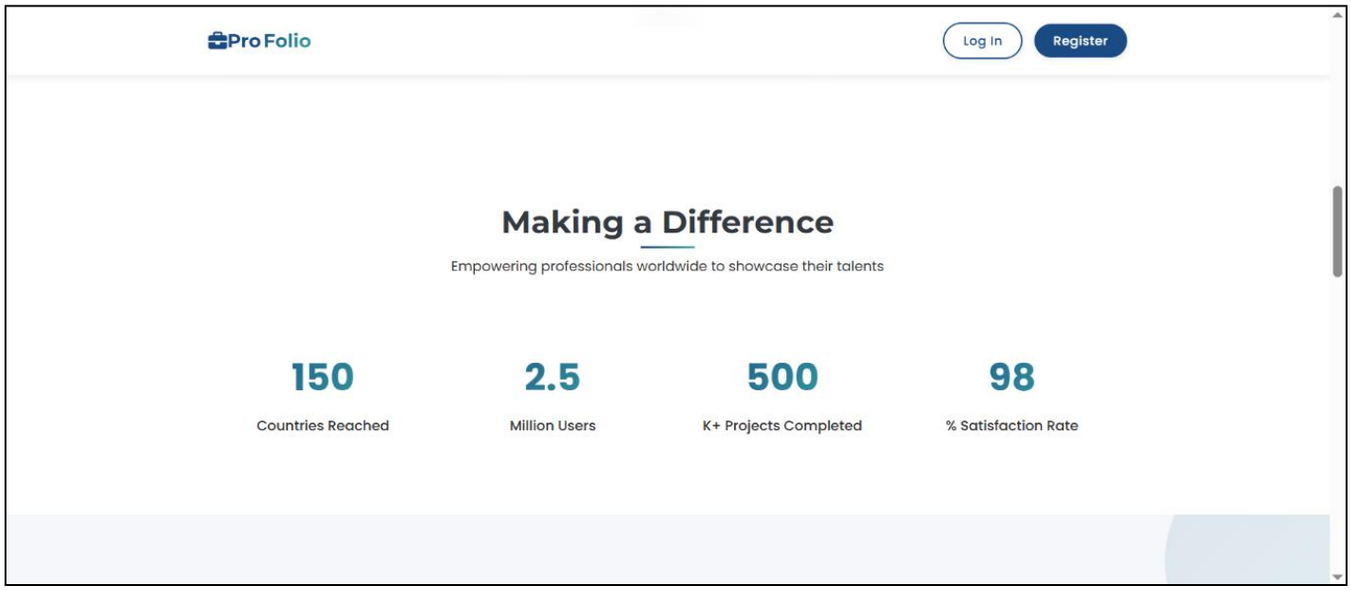
April 20, 2025

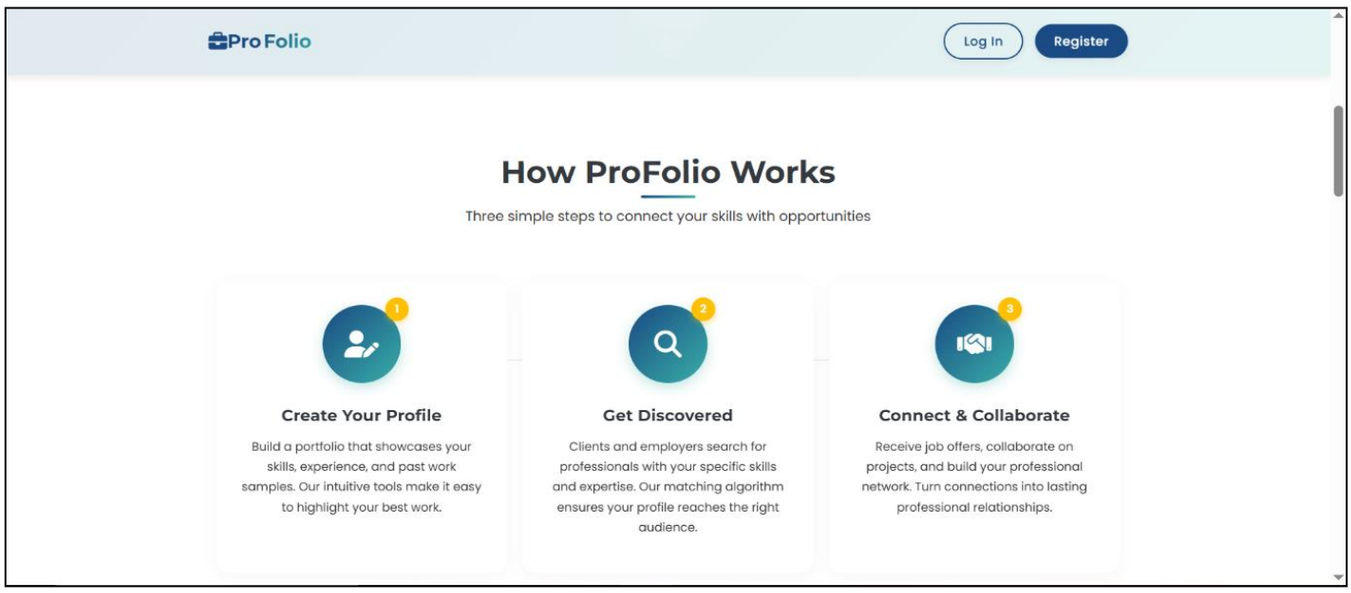
Screenshots of ALL working frontend pages and Annotations showing which feature/use case each page covers

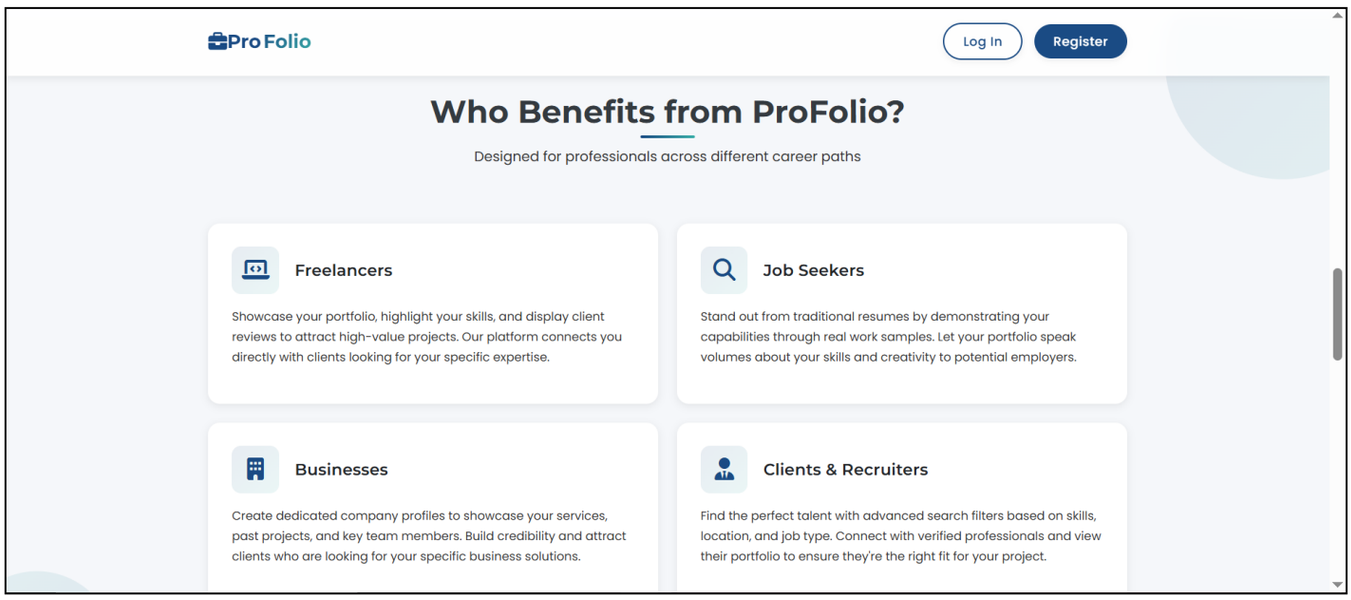
- ## Home Page
  - **Feature:** Main Home Page
  - **Use Case:** The home page is the first thing users see when visiting the platform. It introduces what ProFolio is about, showcases its features, explains how it works, and provides clear navigation for logging in or signing up based on user type. when they visit the platform. It introduces what ProFolio is about, highlights its core features, and guides users to get started. It includes sections like navigation, feature highlights, how the system works, and important footer links. of the platform. From here, users can choose to register as a client, freelancer, or agency, or log in to their existing account. Each section is designed to introduce users to the platform's purpose, features, and how it works.



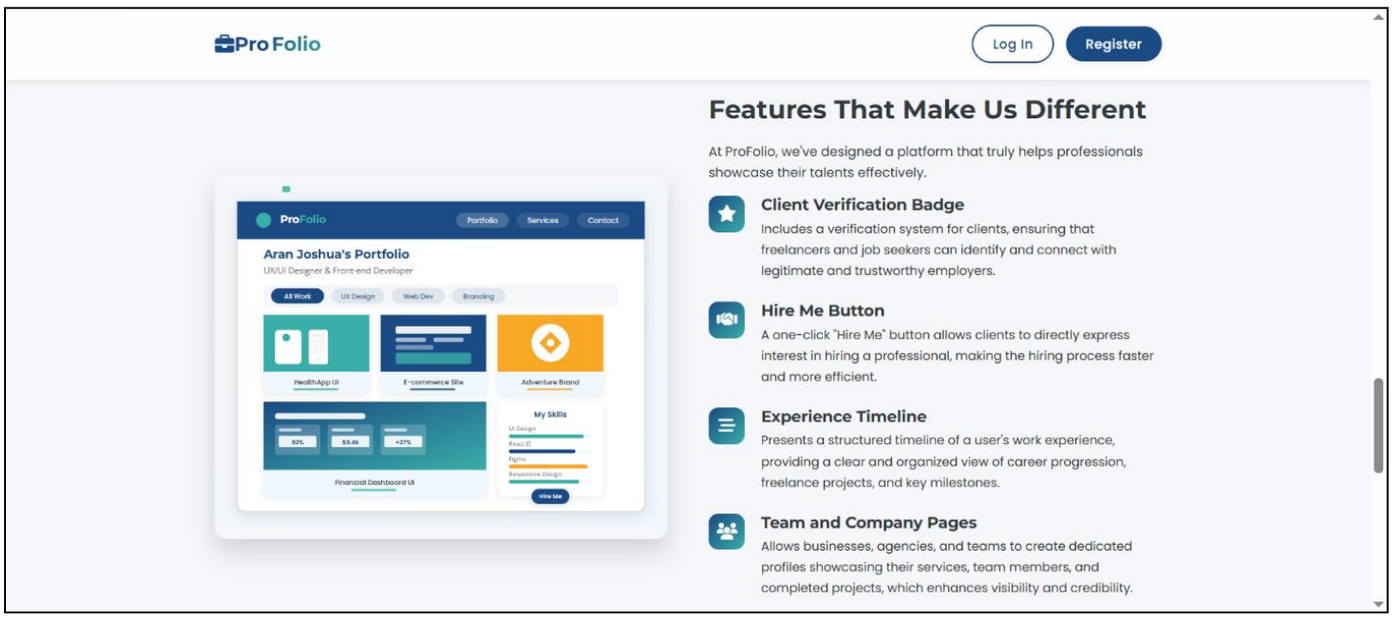
Scrolling Down in Homepage

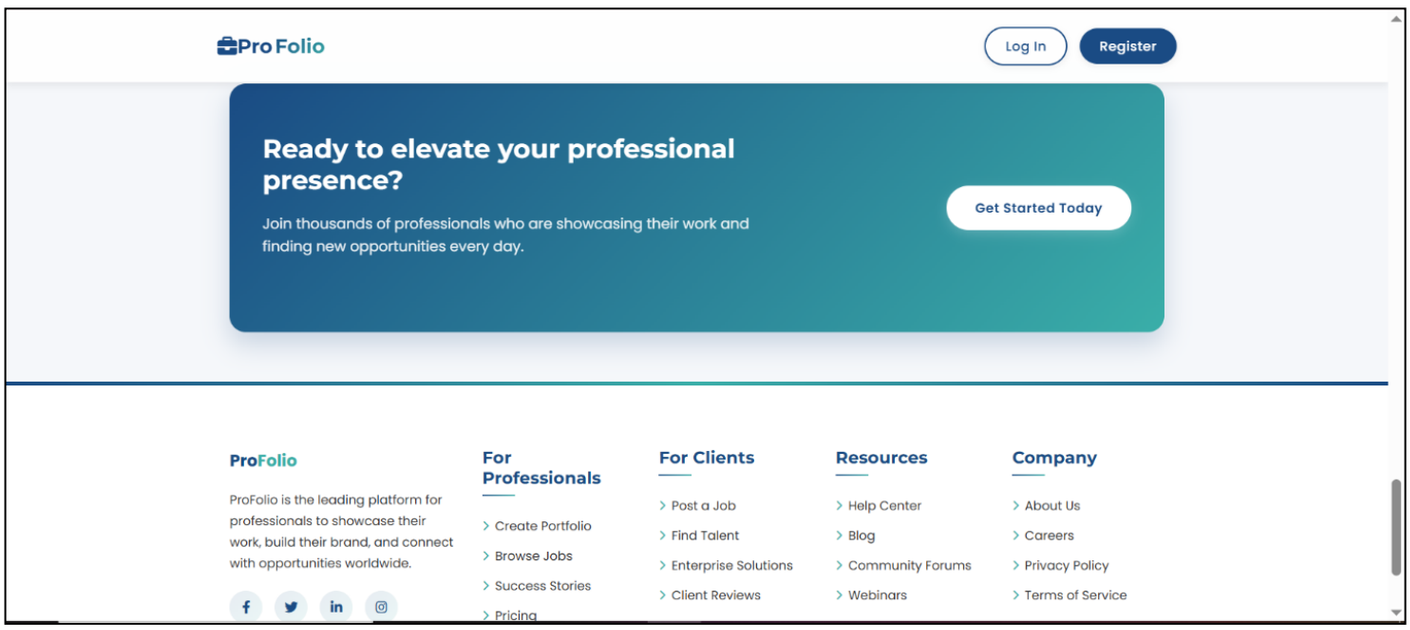




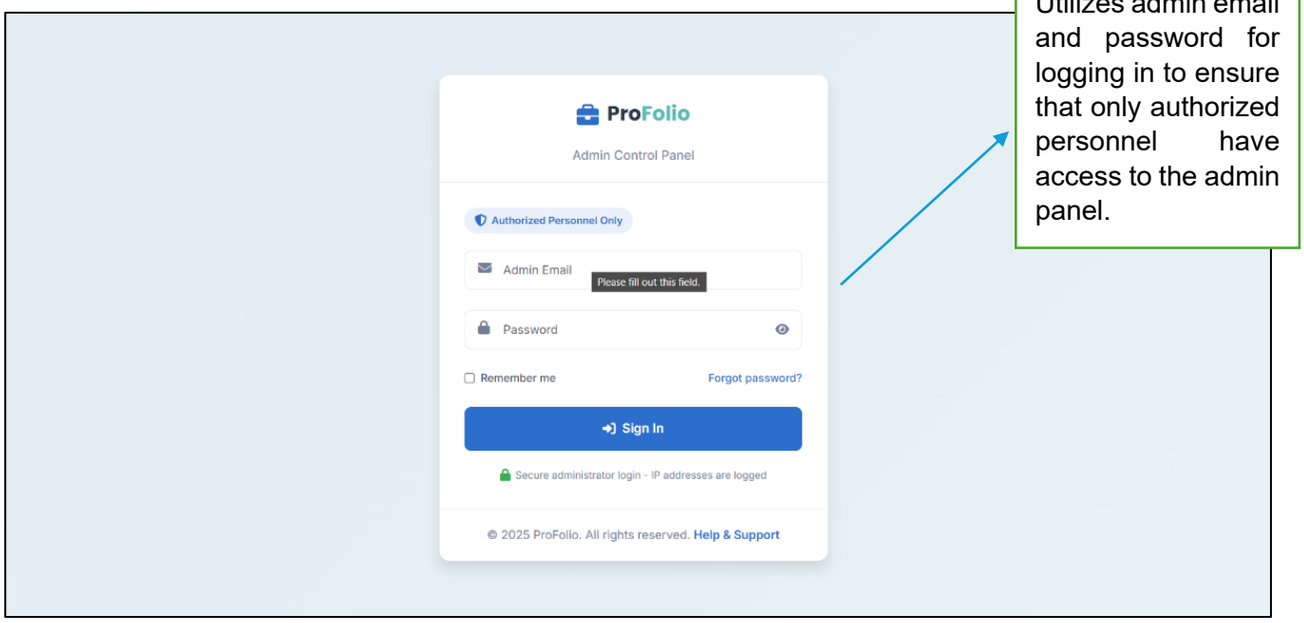


Scrolling Down in Homepage

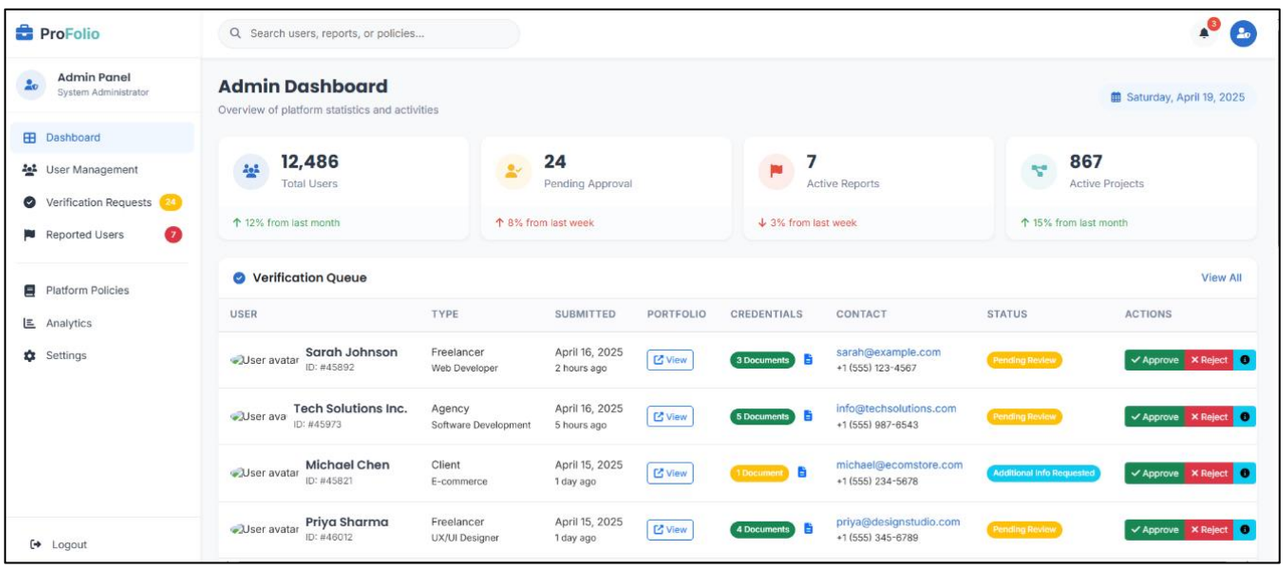




- **Admin Login**
  - **Feature:** Login form for administrators
  - **Use Case:** Allows admin users to securely log into the dashboard using their credentials. It includes input validation and backend integration to ensure only authorized users can gain access to the admin system.

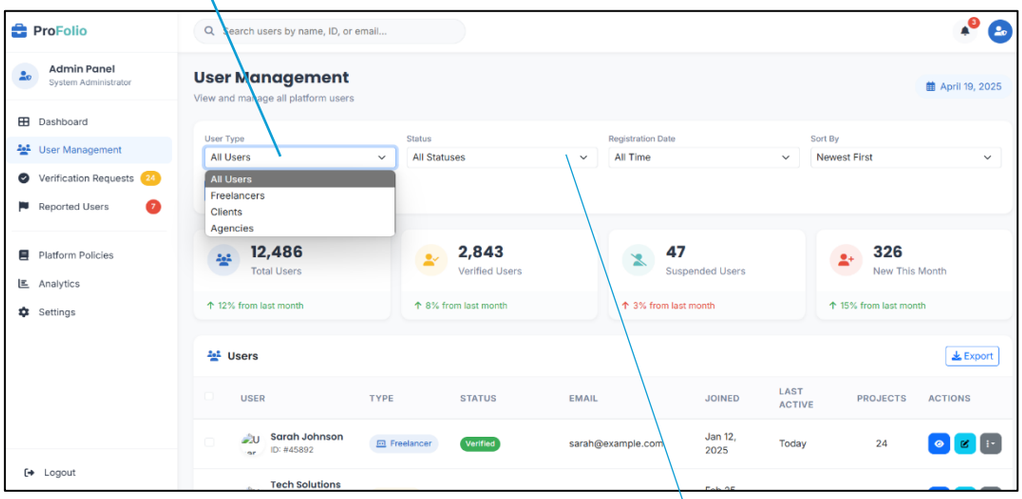


- **Admin - Dashboard**
  - **Feature:** Main interface for administrative control
  - **Use Case:** Serves as an overview page of the platform, providing a high-level summary of platform activity, user engagement, and administrative tasks.

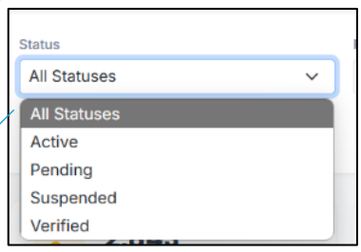


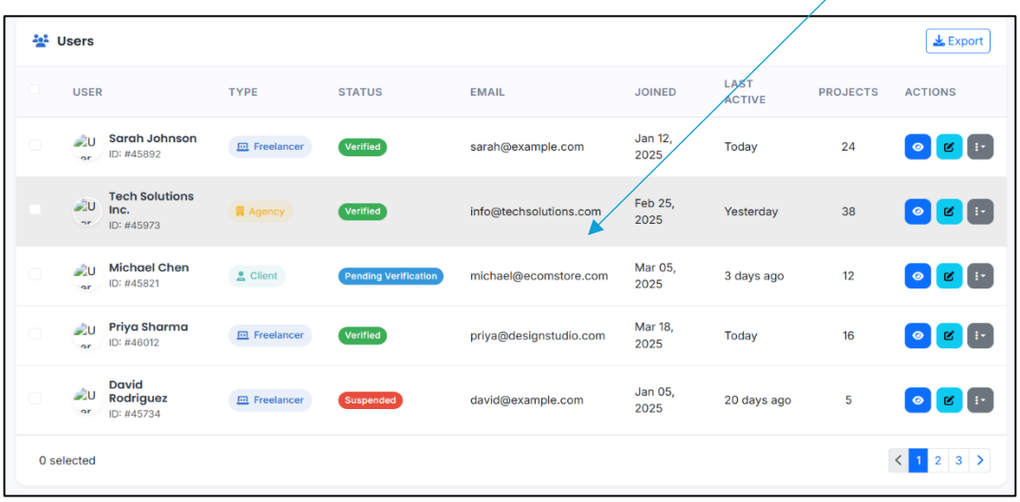
- **Admin – User Management**
  - **Feature:** User management interface
  - **Use Case:** Enables the admin to view and manage a list of all registered users. Admins can filter users by type or account status to locate specific user groups. This makes it easier to organize users and monitor platform activity.

User Type Filter – lets the admin filter users by account type (All Users, Freelancer, Client, Agencies).



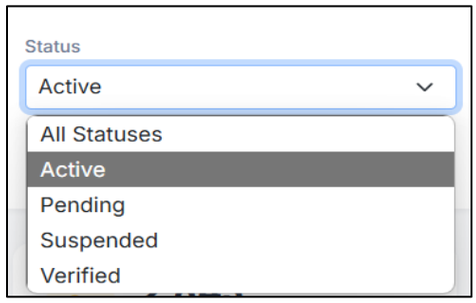
When **'All Users'** is selected and the filter is applied, the system first displays the message: "Filters applied successfully!"

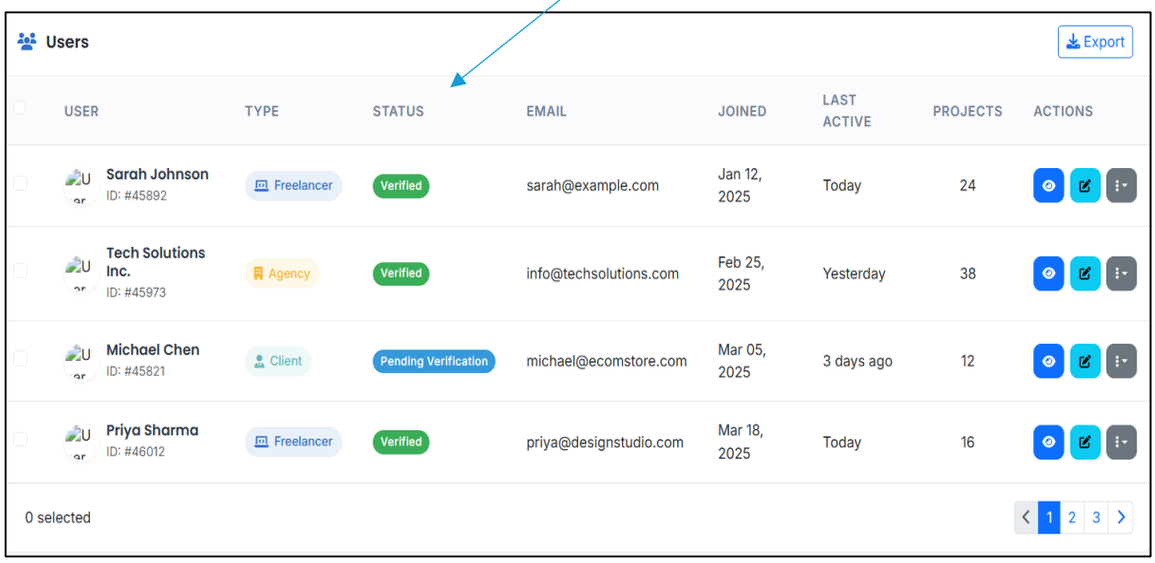




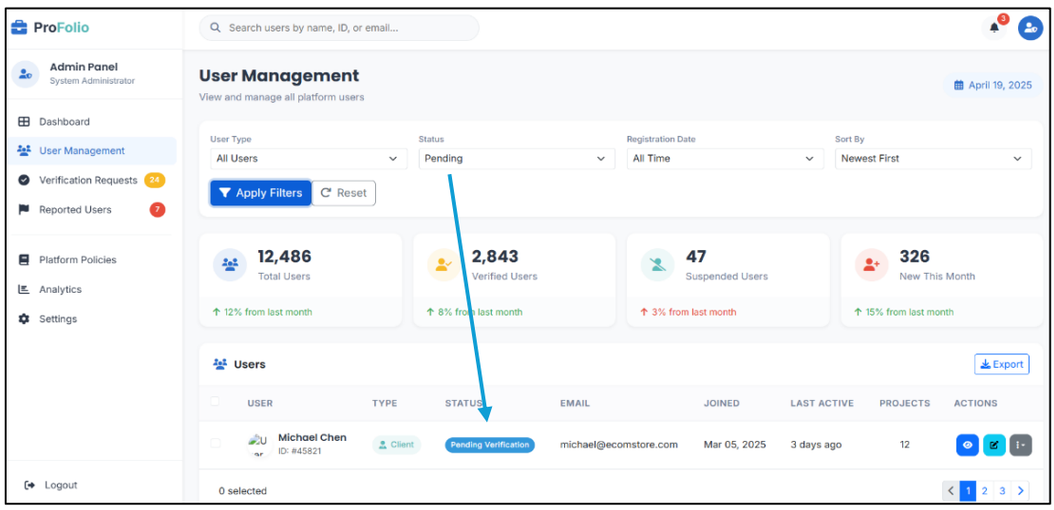
When **'All Statuses'** is selected, the system will show users of all statuses (verified, pending verification, suspended, etc.)

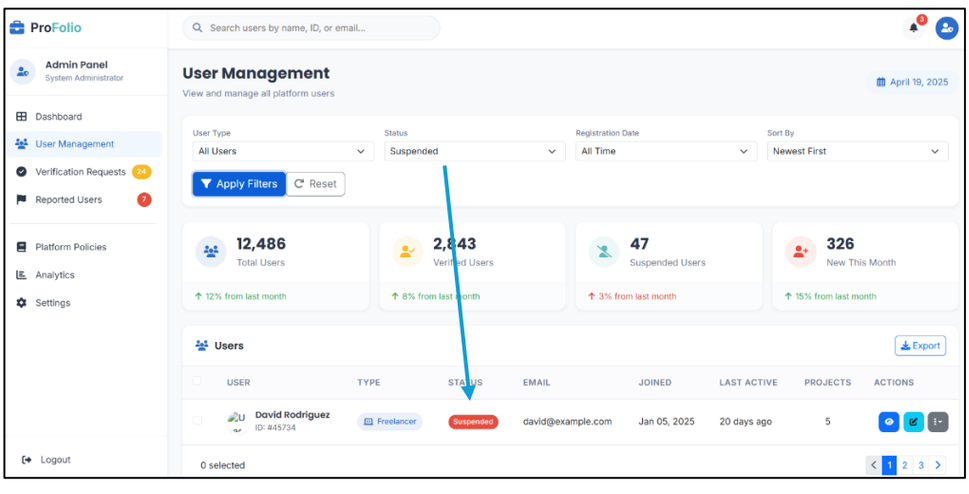
When **'Active'** is selected, the list shows users who currently have access to their accounts and are allowed to use the platform's features.





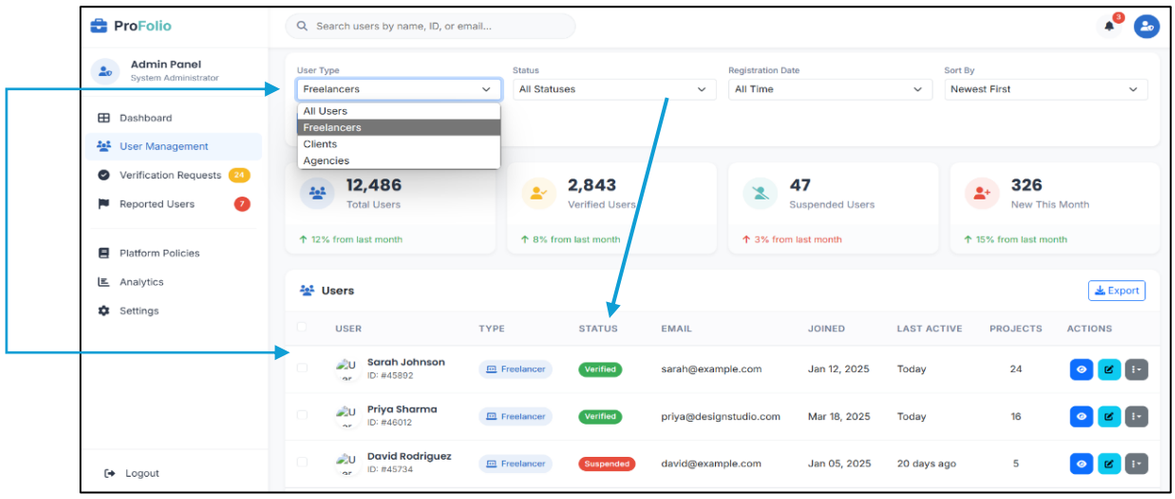
When '**Pending**' is selected, only users whose accounts are still under review or awaiting approval will be shown, helping the admin easily identify users who require further action.



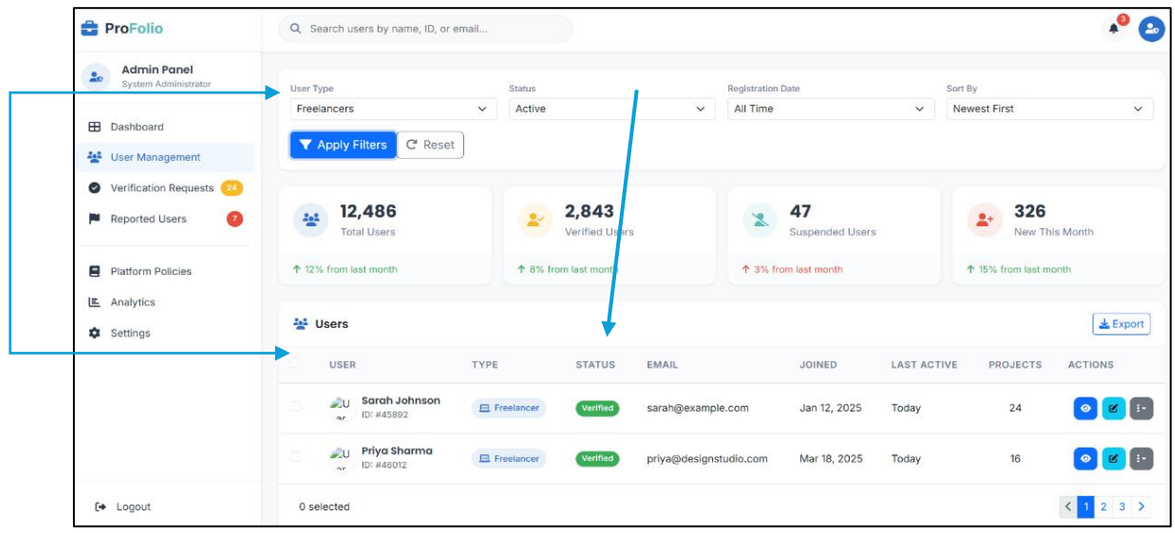


When '**Suspended**' is selected, the system shows users who have been restricted from accessing the platform due to violations of terms or other administrative reasons

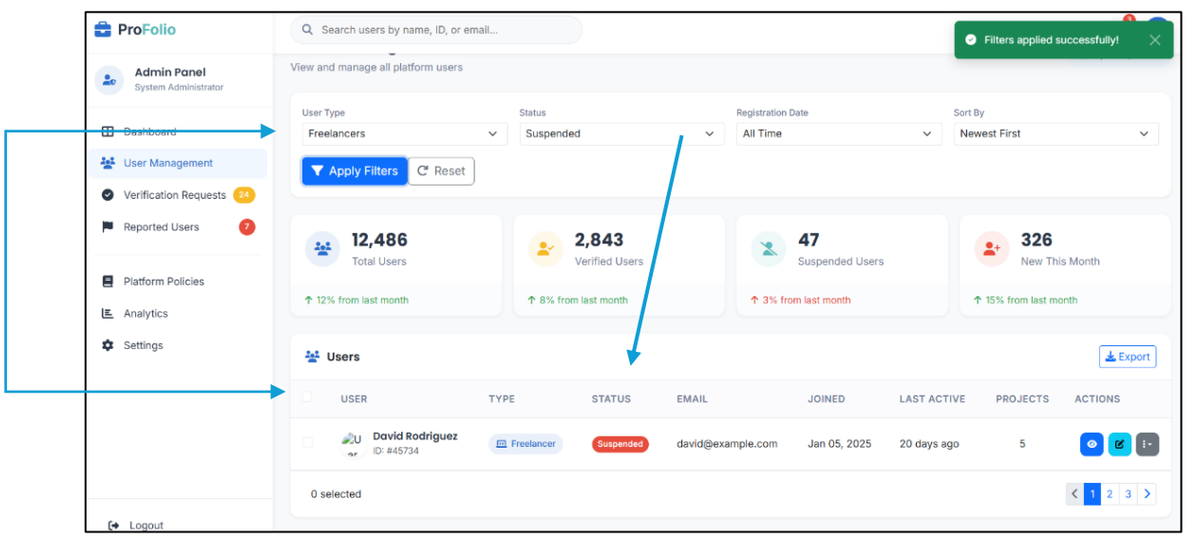
If **Freelancer** is selected and **Verified** is applied, the system will display only freelancers whose accounts have been verified.



If **Client** is selected and **All Statuses** is applied, the system will display only clients, regardless of their account status. If the only status available for clients is **Pending**, the system will display only clients with a Pending status.

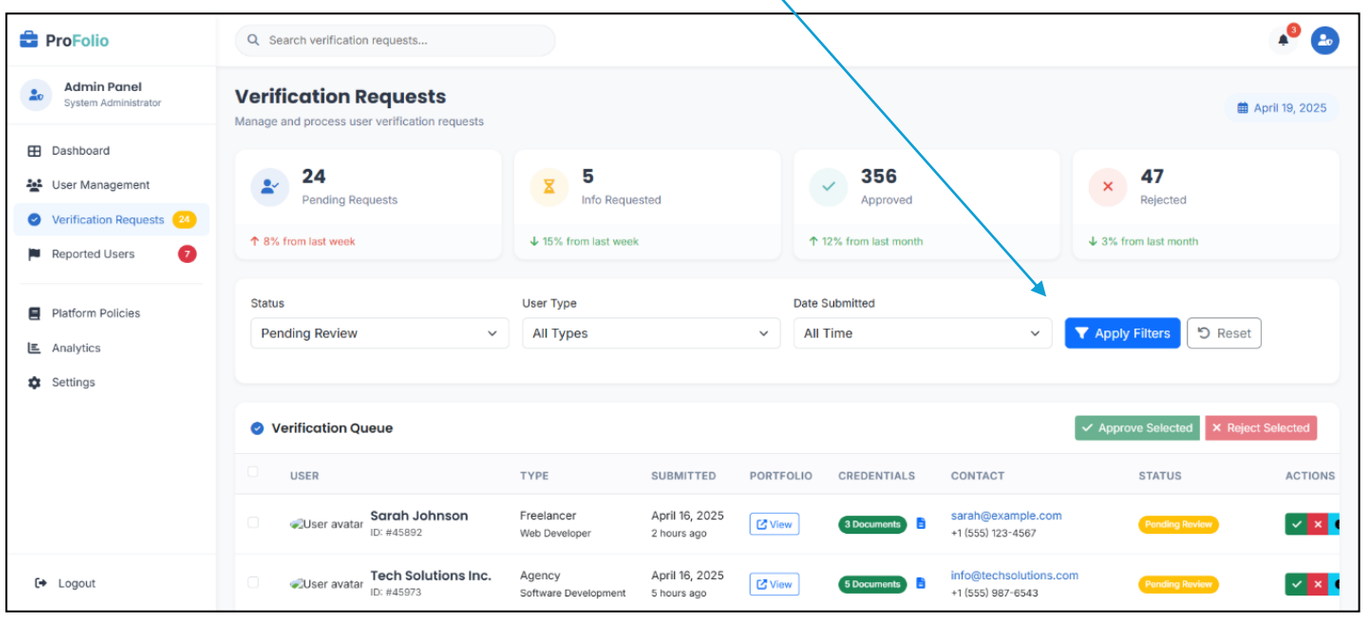


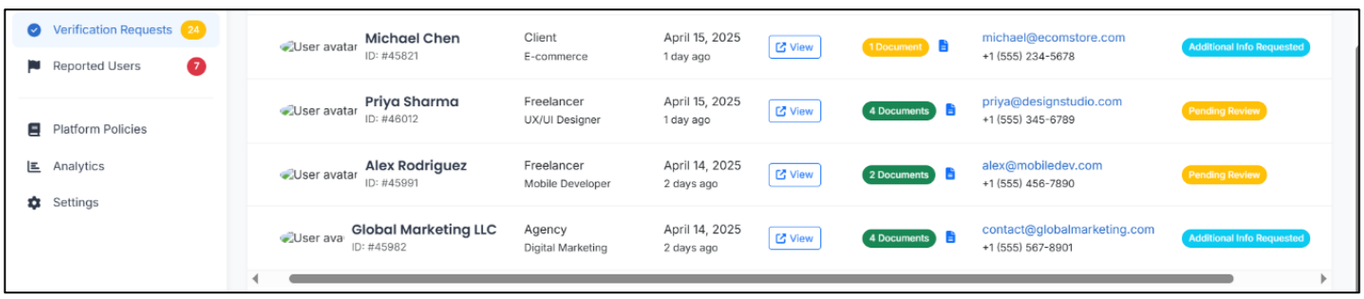
If **Agency** is selected and **All Statuses** is applied, the system will display only agencies, regardless of their account status. If the only status available for agencies is **Verified**, the system will display only agencies with a Verified status.

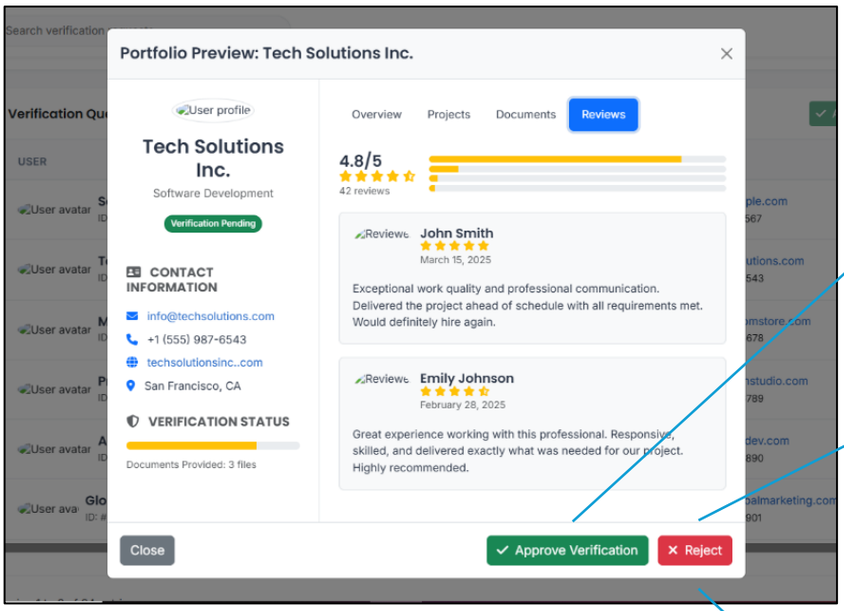


- **Verification Requests**
  - **Feature:** User verification management
  - **Use Case:** This page lets the system administrators to review, approve, or reject verification requests submitted by users who want to be verified on the platform.

Advanced filters for sorting users based on status, user types and verification submission date.

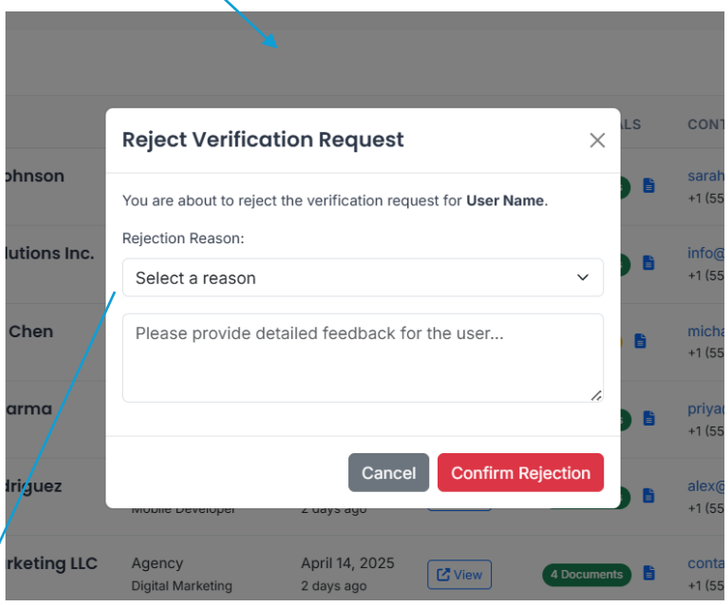




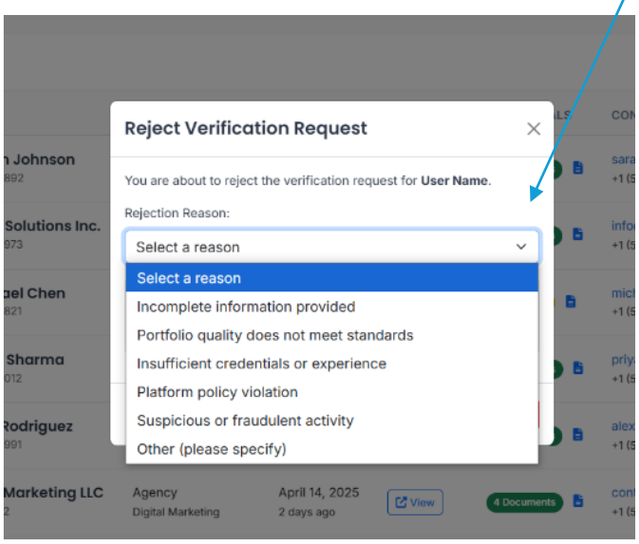


An **'Approve Verification'** button to verify users who meet the platform's requirements and submitted complete, valid credentials.

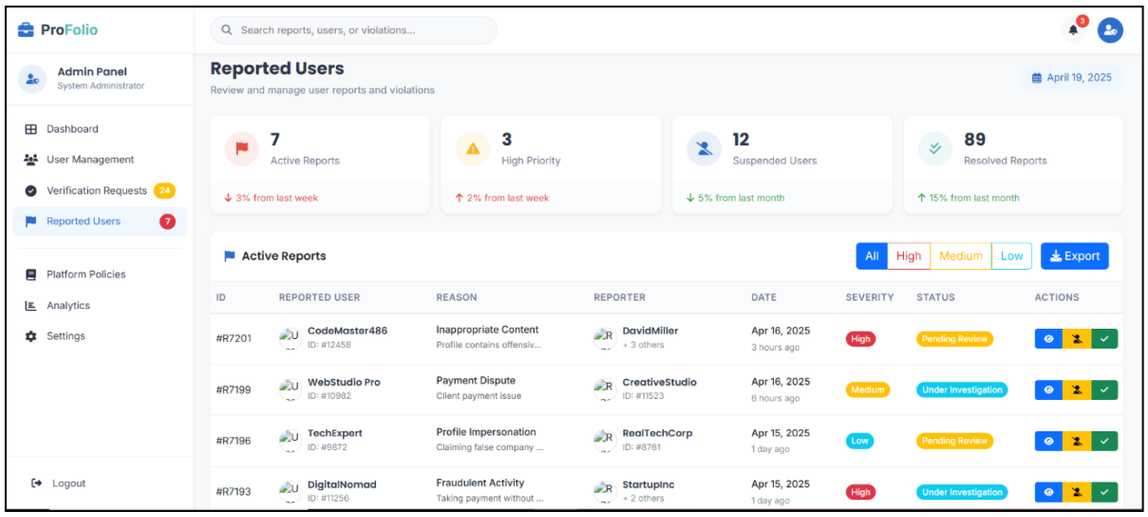
A **'Reject'** button to prevent unqualified, incomplete, or suspicious users from being verified.

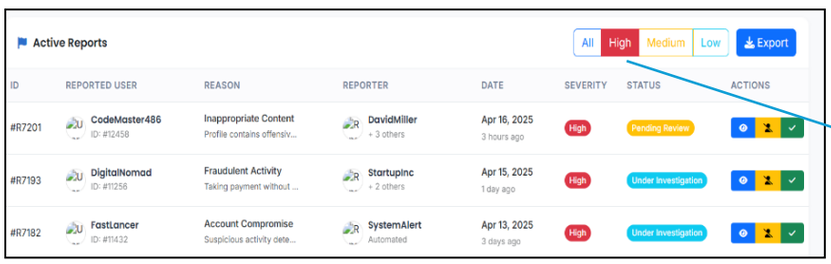


Enables selection of a specific reason for rejecting the verification request, along with an input field for providing detailed feedback.

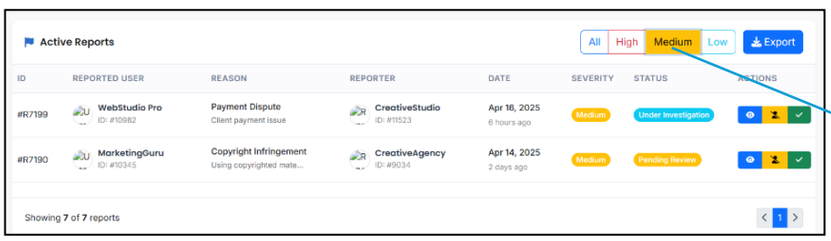


- **Admin – Reported Users**
  - **Feature:** Reported user management with report details and action controls.
  - **Use Case:** This page in the admin panel is used to monitor and manage user behavior on the platform. It allows admins to review reports submitted by other users, including reasons for the report. Based on these details, admins can take appropriate actions such as warning, suspending, or banning users who violate the platform's rules.

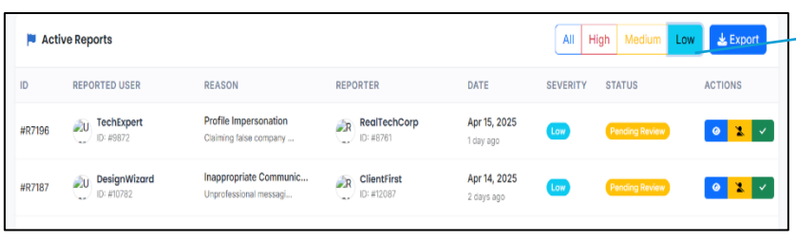




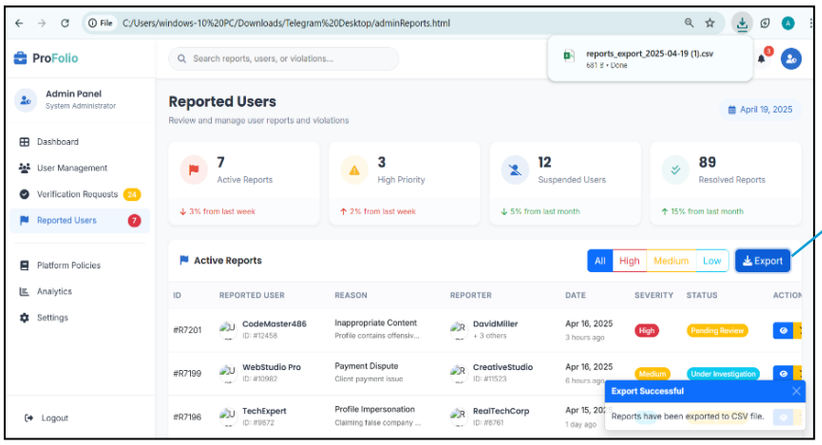
Filters the table to show only high-severity reports or serious issues like fraudulent activity.



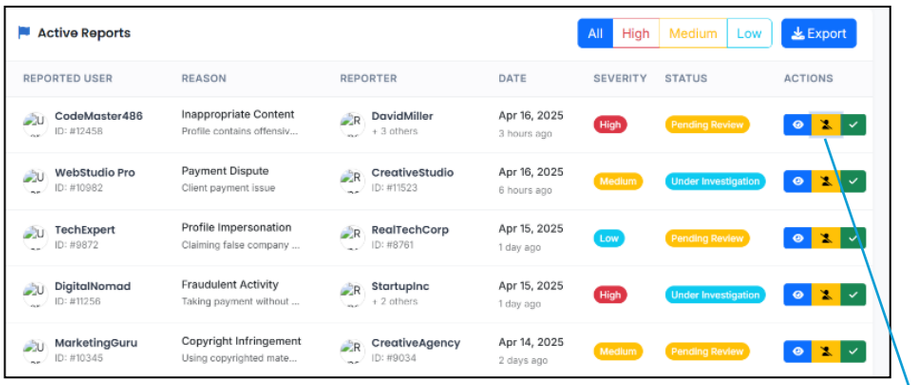
Displays only medium-severity reports, which are less urgent than high ones but still require attention, such as client payment issues or the use of copyrighted material.



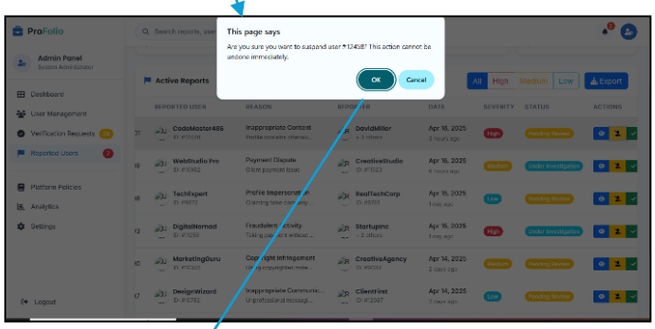
Shows only low-severity reports or minor issues that can be addressed later.



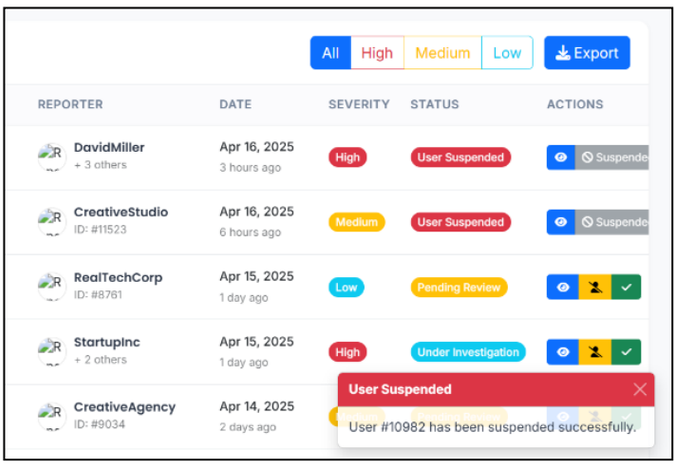
The ‘Export’ button is used to download the report data, specifically the reported users list into a CSV file format. In this screenshot, a message “Export Successful” appears after clicking the button, and a file containing the reports is downloaded.

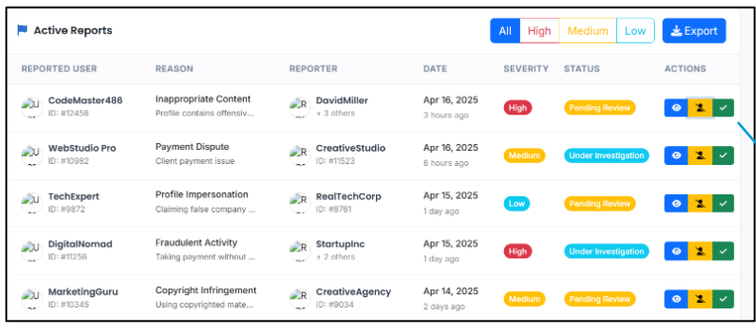


When the suspend icon is clicked, a confirmation dialog pop-up appears, asking the admin to confirm the suspension of the reported user.



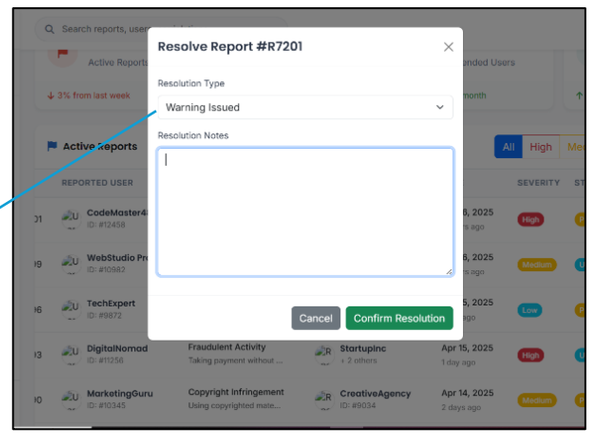
After clicking ‘OK’, a message will immediately appear confirming the successful suspension of the user. In addition, the status of the suspended user will also change from ‘Pending Review’ to ‘User Suspended’.

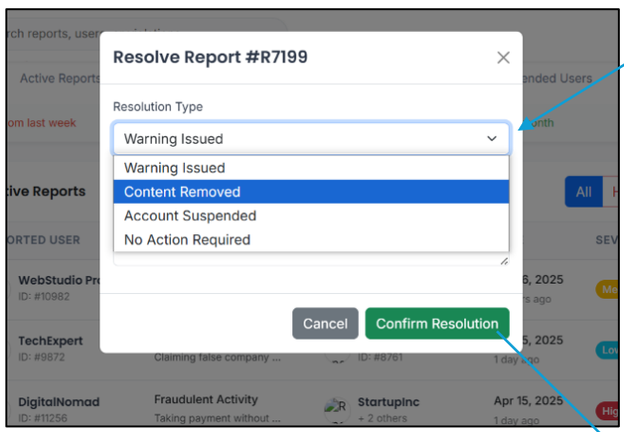




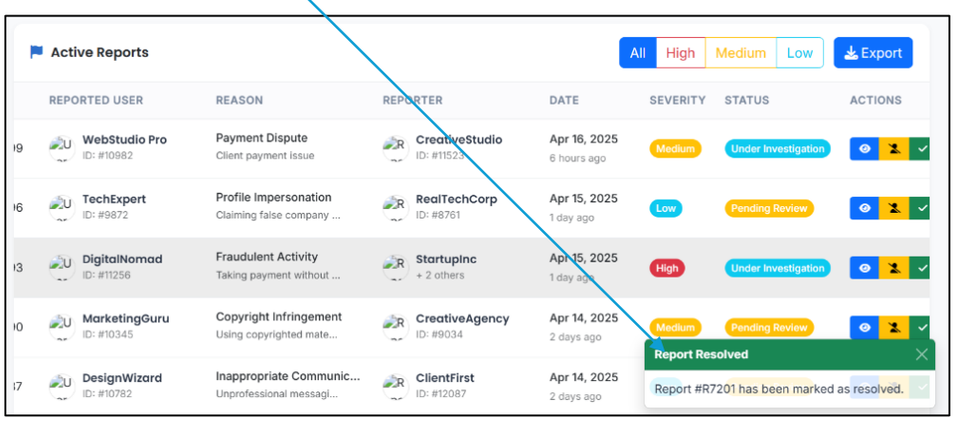
This pop-up allows an admin to resolve a user complaint.

It includes a dropdown of multiple resolution types, giving more flexibility on how reports are handled.



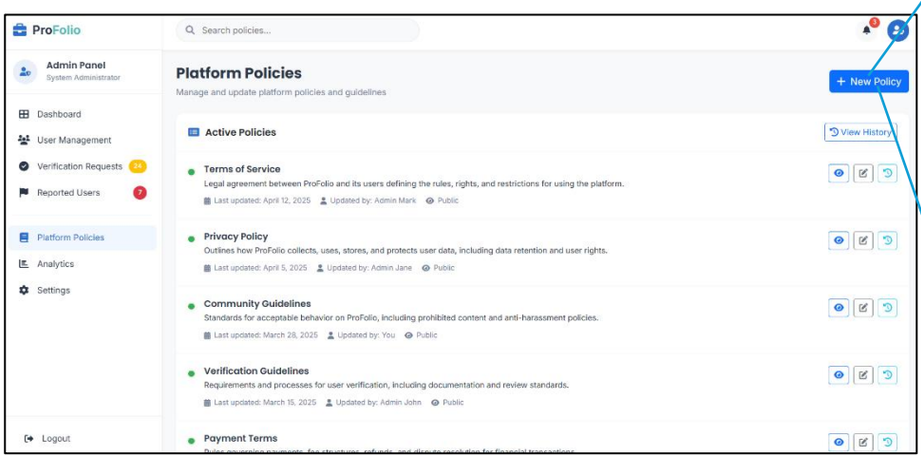


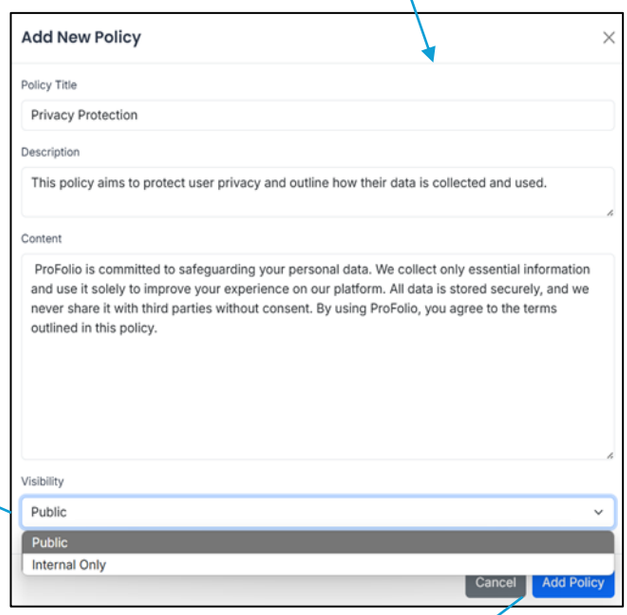
Once the admin confirms a resolution, the system displays a confirmation message at the bottom.



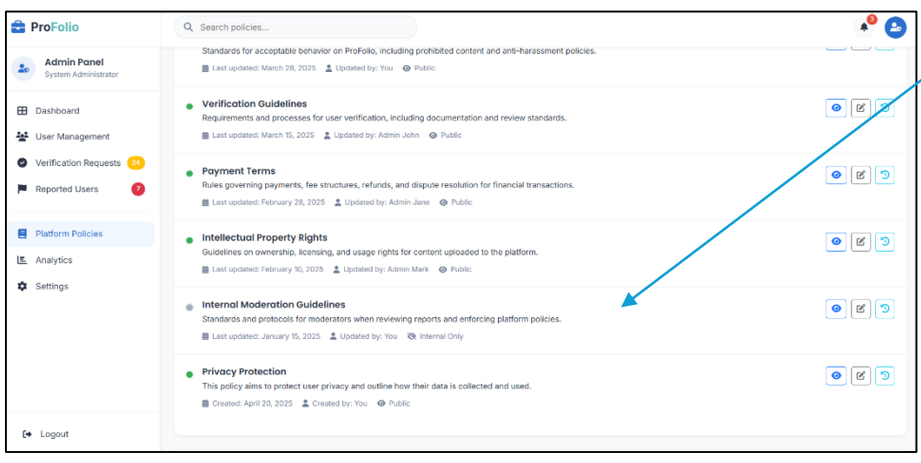
- **Admin – Platform Policies**
  - **Feature:** Platform Policy Management Panel
  - **Use Case:** Allows administrators to add, edit, or publish policies such as Terms of Service, Privacy Protection, or Community Guidelines. This section ensures that the platform maintains transparency and clearly communicates rules to its users.

Allows the admin to create a new policy entry with a custom title, short description, and full content.



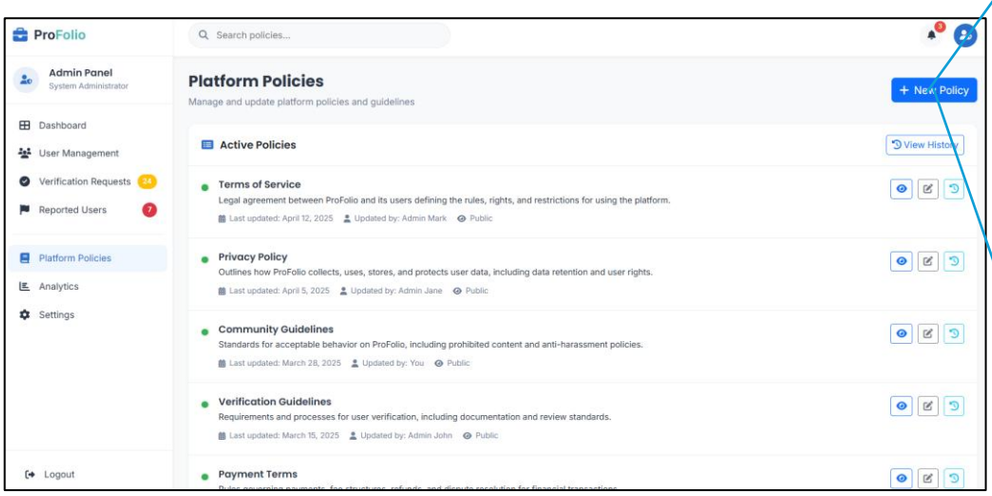


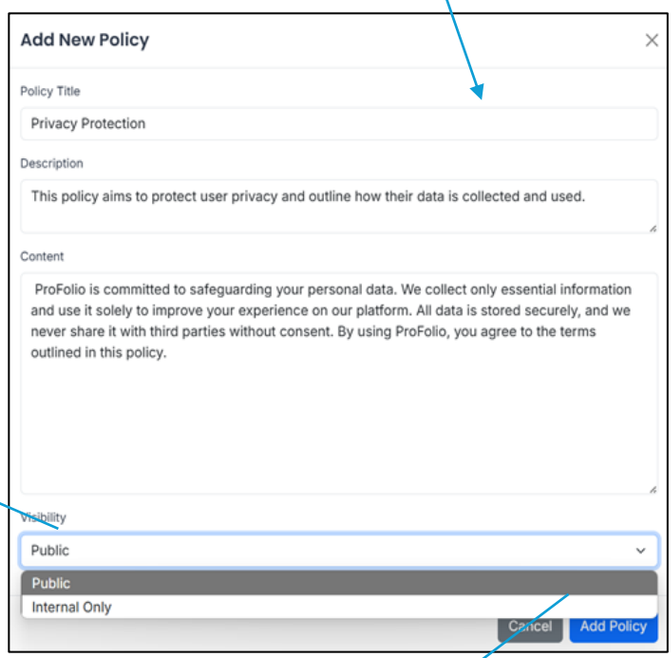
Lets the admin choose whether the new policy is intended for **public display** (visible to users on the website) or **internal only** (visible to admin staff for internal reference).



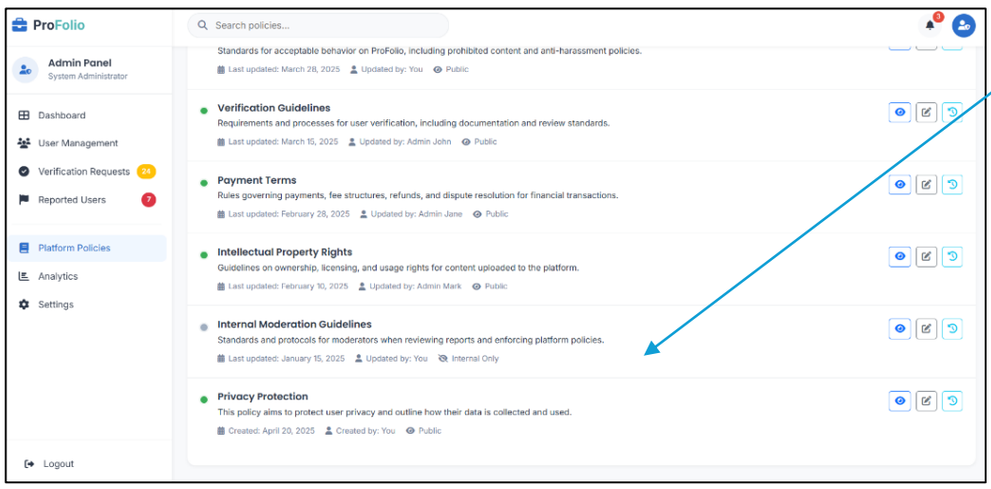
After the admin creates and saves a new policy '**Privacy Protection**' it will automatically appear in this section.

Allows the admin to create a new policy entry with a custom title, short description, and full content.



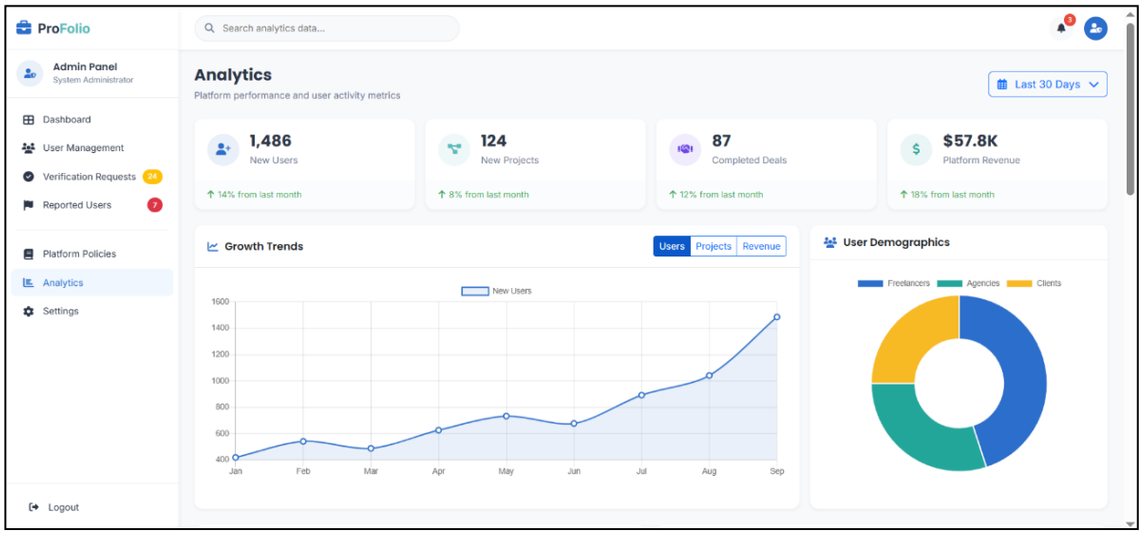


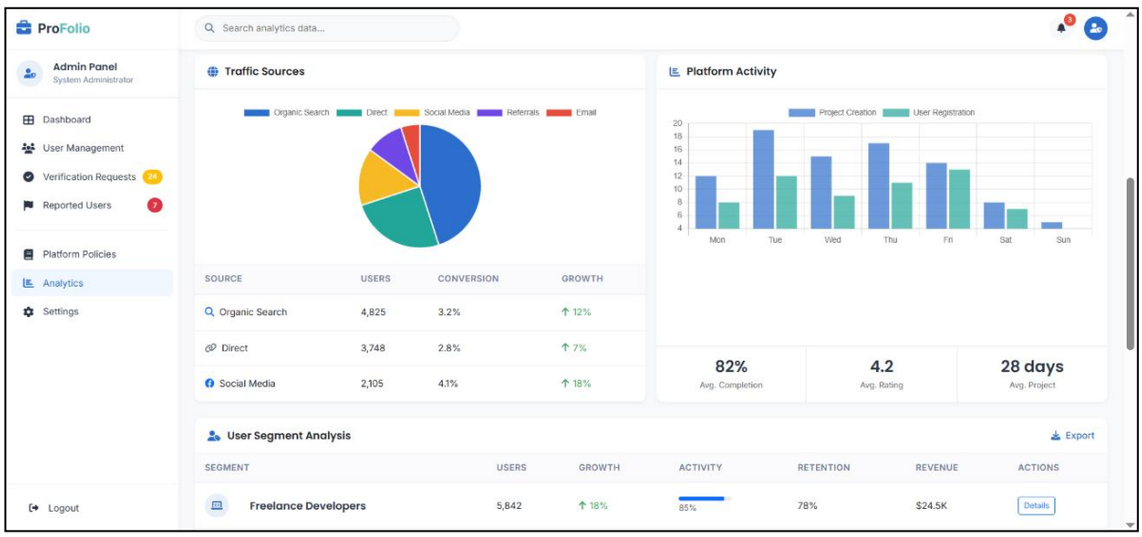
Lets the admin choose whether the new policy is intended for **public display** (visible to users on the website) or **internal only** (visible to admin staff for internal reference).

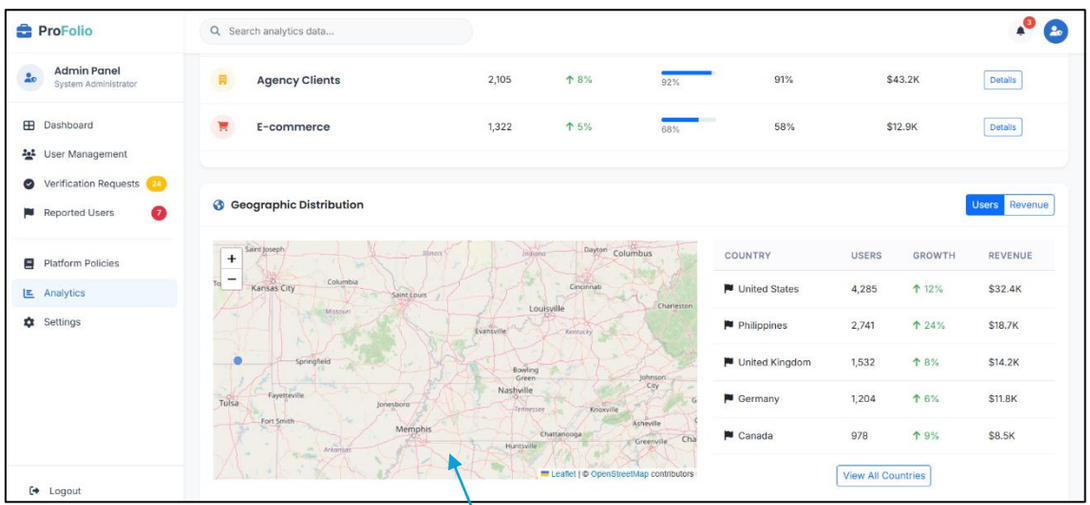


After the admin creates and saves a new policy '**Privacy Protection**' it will automatically appear in this section.

- **Analytics**
  - **Feature:** Tracking and visualization of platform data
  - **Use Case:** This page gives administrators insight to platform usage and engagement, showing key stats like new users, projects, deals, and revenue, along with monthly growth. It helps track progress, spot trends, and make informed decisions. It also has user demographics chart, displaying the types of users on the platform for more targeted strategies.

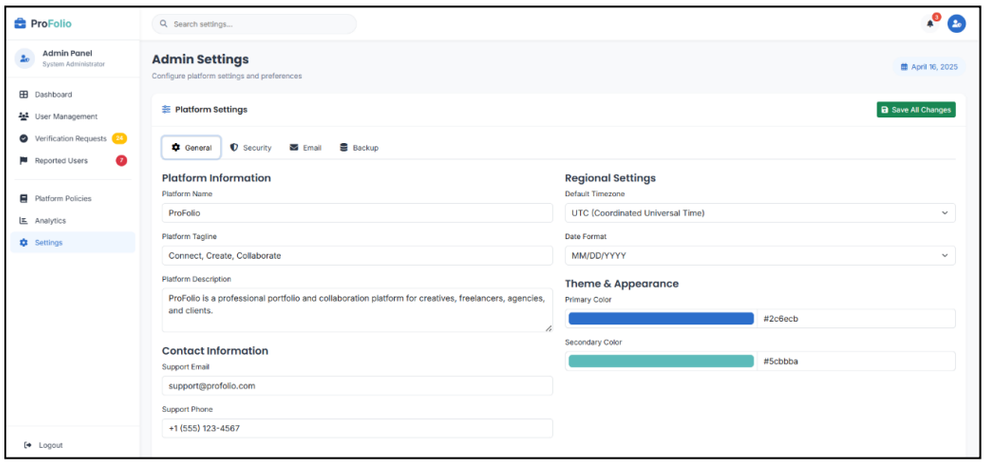


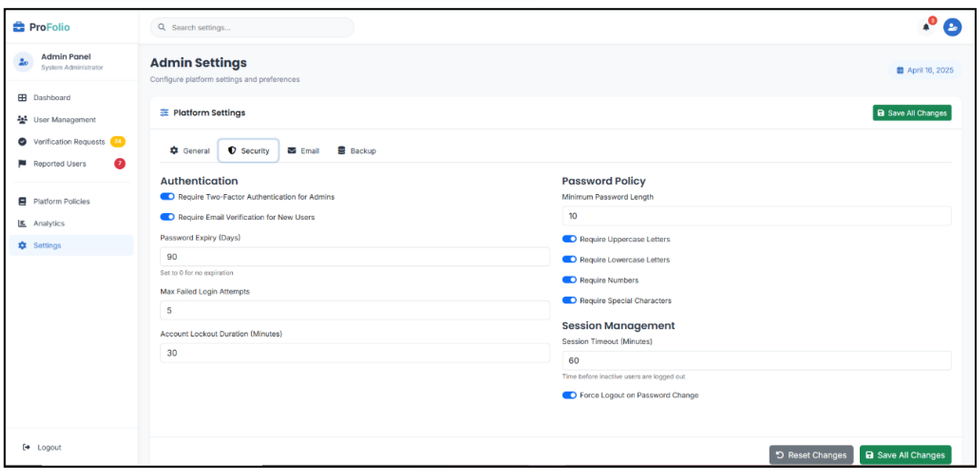


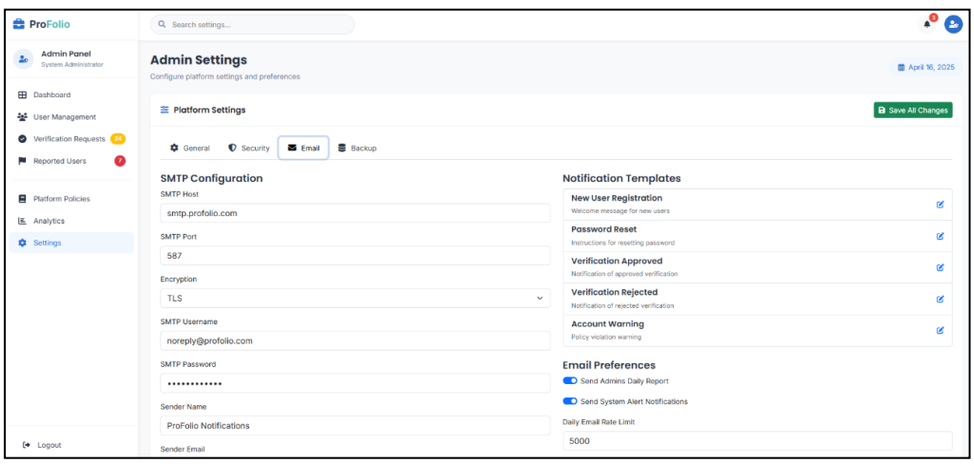


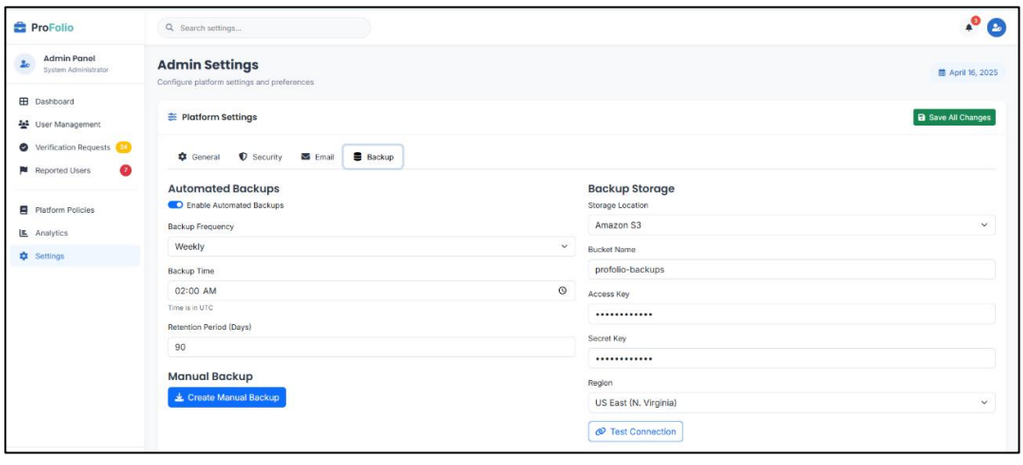
A geographic distribution of users on the platform created using Leaflet. This helps the admin understand which countries are most active or profitable, which can guide decisions related to marketing and support.

- **Settings**
  - **Feature:** Website administration and configuration
  - **Use Case:** Enables website administrators to manage and configure various aspects of the site's functionality, security, and overall system settings. Through this page, admins can customize platform information and site appearance, set security, configure SMTP or email settings, and manage backups.

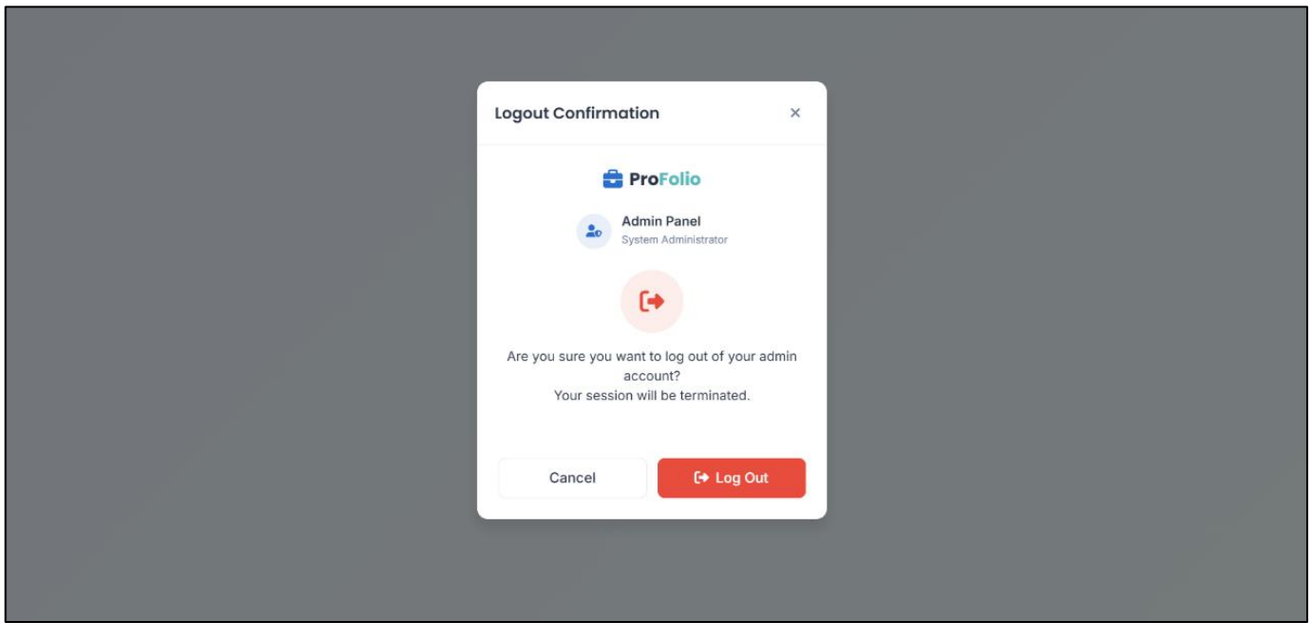




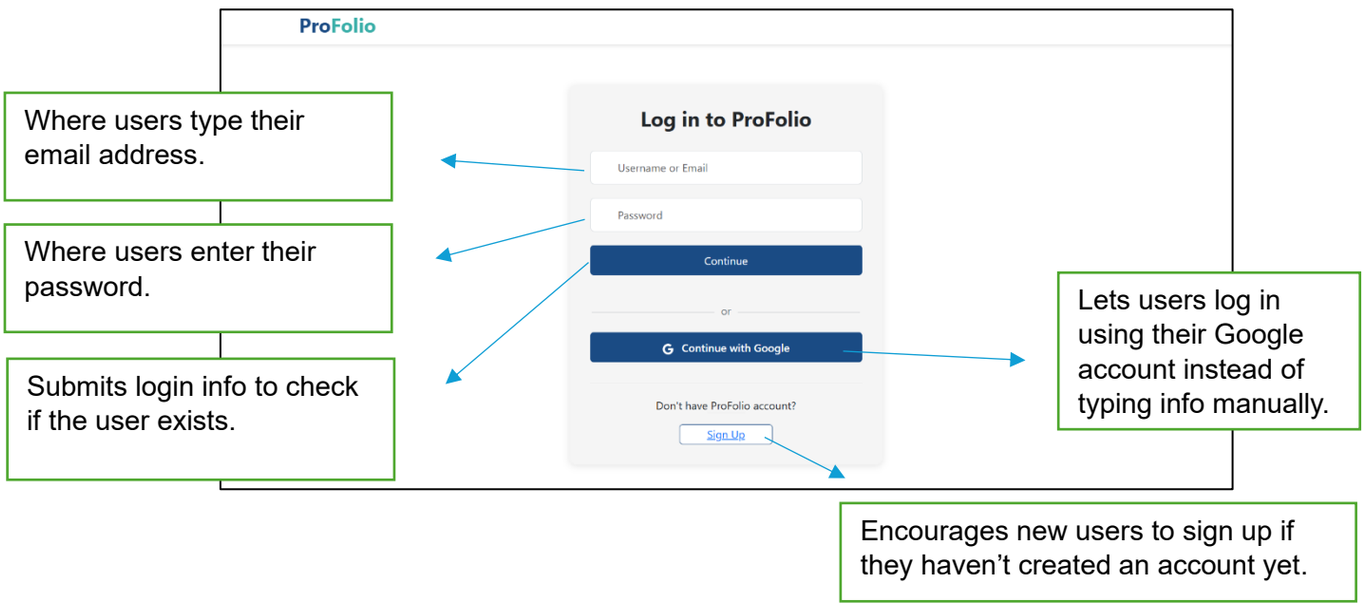




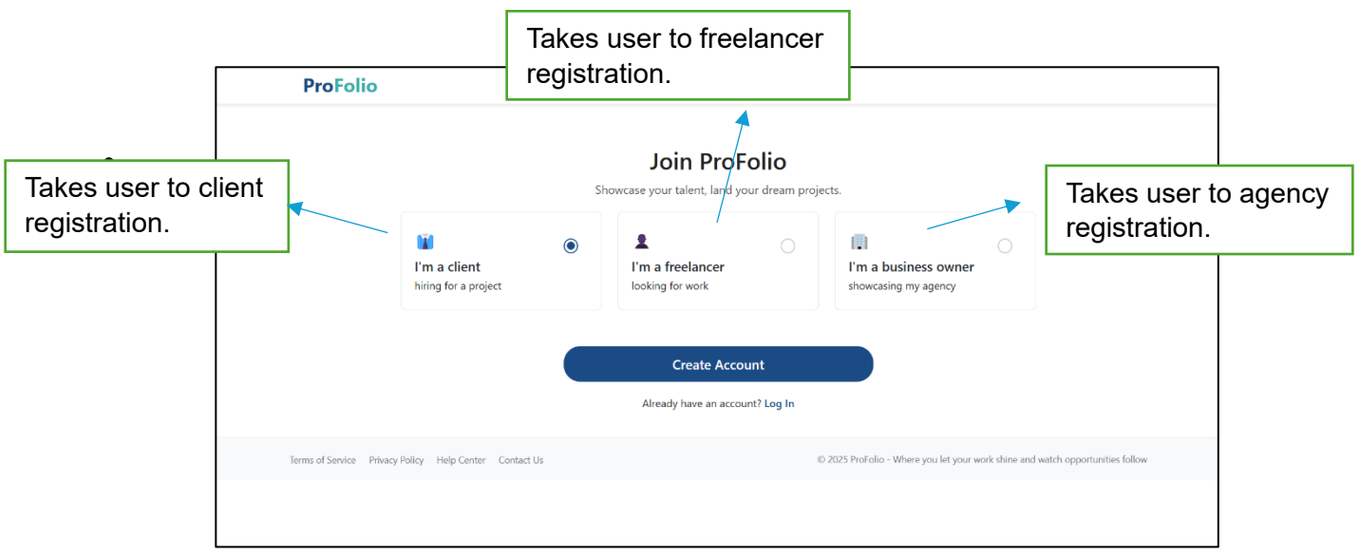
- **Admin Logout**
  - **Feature:** Logout confirmation or button
  - **Use Case:** This functionality allows administrators to securely end their session. Upon logging out, the system clears or invalidates the session variables used for authentication, making sure that no other user can access admin features without logging in again. This supports secure session handling and reinforces system protection from unauthorized access.



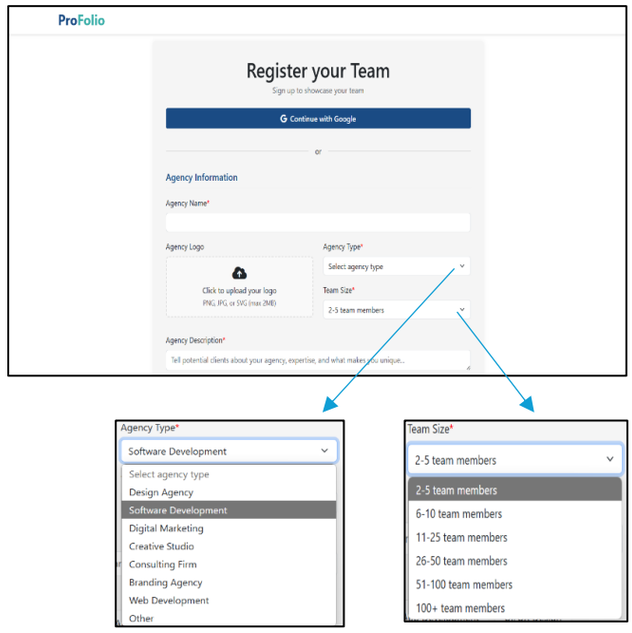
- **User - Login**
  - **Feature:** General User Login Page
  - **Use Case:** Allows users (client, freelancer, agency) to log in to their account using email and password. This form also includes a redirection link to registration pages.



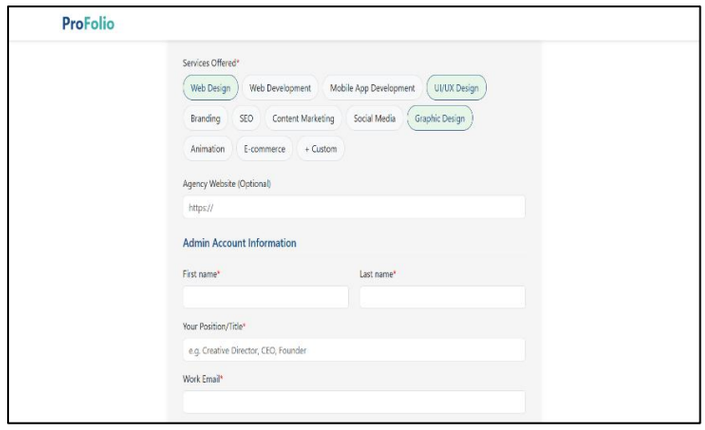
- **Register**
  - **Feature:** User Registration Selection Page
  - **Use Case:** Lets users choose the type of account they want to create — client, freelancer, or agency. This acts as a gateway to the proper registration form based on user role.



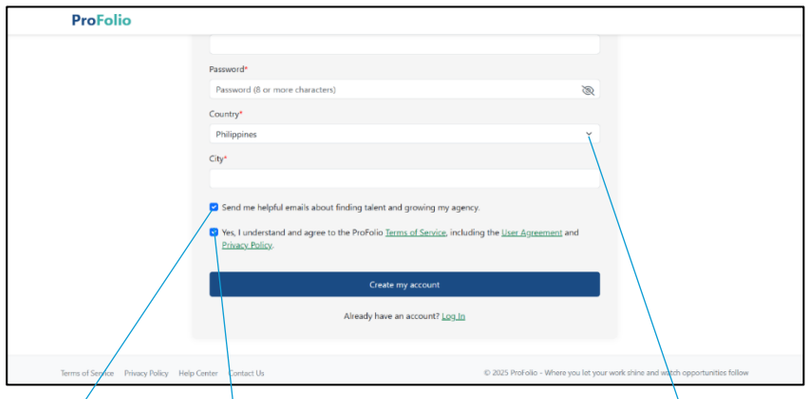
- # Register – as an Agency
  - **Feature:** Agency Registration Form
  - **Use Case:** Allows agencies to register by entering detailed business information, uploading a logo, and defining their services. This form collects data to differentiate agency accounts from freelancers or clients



- **Agency Name Field** – official name of the agency
- **Logo Upload** – file input for agency logo
- **Agency Type Dropdown** – used to select the type/category of agency
- **Team Size Field** – lets agencies specify how many people are in the team
- **Agency Description** – a short paragraph describing the agency's background and focus



- **Services Offered** – allows the user to choose multiple service offerings (e.g., web design, Ui/UX design, graphic design)
- **Agency Website Field** – optional link to the official site
- **First Name and Last Name** – personal details of the account owner
- **Job Title Field** – specifies their role in the agency (e.g., CEO, HR Manager)
- **Work Email Field** – required email address for login and contact



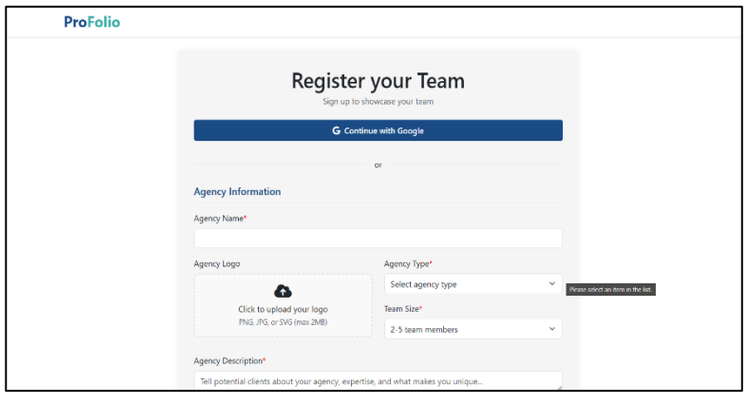
- **Password Field** – required for account security
- **Country and City Fields** – capture agency's location for directory purposes
- **Submit Button** – completes the registration and sends all data to the backend

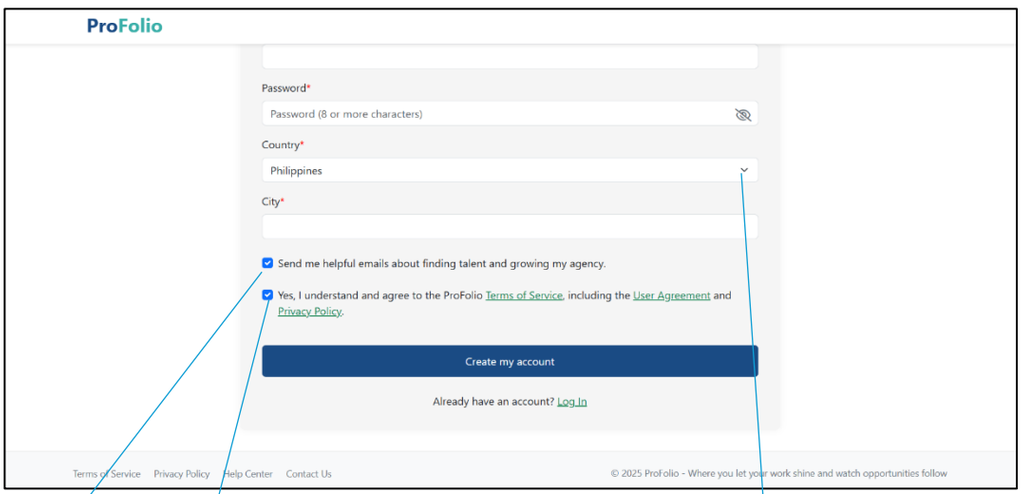
Gives users the option to receive email tips or project suggestion.

Users must agree before creating their account.

- **Register – as a Client**
  - **Feature:** Client Registration Form
  - **Use Case:** This page allows clients to register an account by providing their personal and location information. It includes fields like first name, last name, email, password, and country. The form ensures that each user provides valid and complete details before they can use the platform.

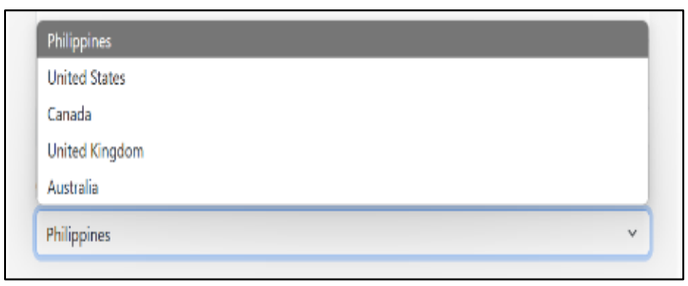
- **First Name/Last Name Fields** – collect the user's full name
- **Email Input Field** – user provides their email for identification
- **Password Field** – stores a hashed version of the password in the database
- **Country Field** – captures the user's location for analytical purposes
- **Submit/Register Button** – sends data to backend for account creation.





Gives users the option to receive email tips or project suggestions.

Users must agree before creating their account.

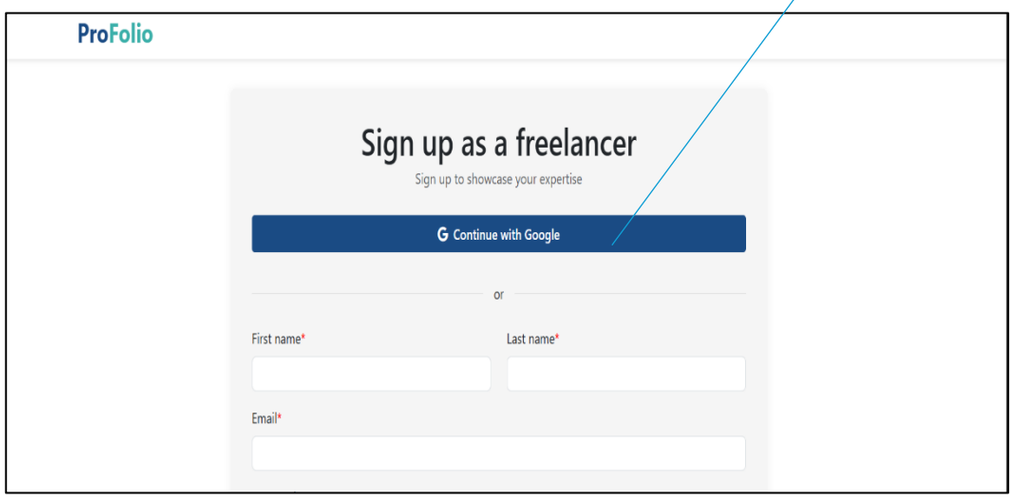


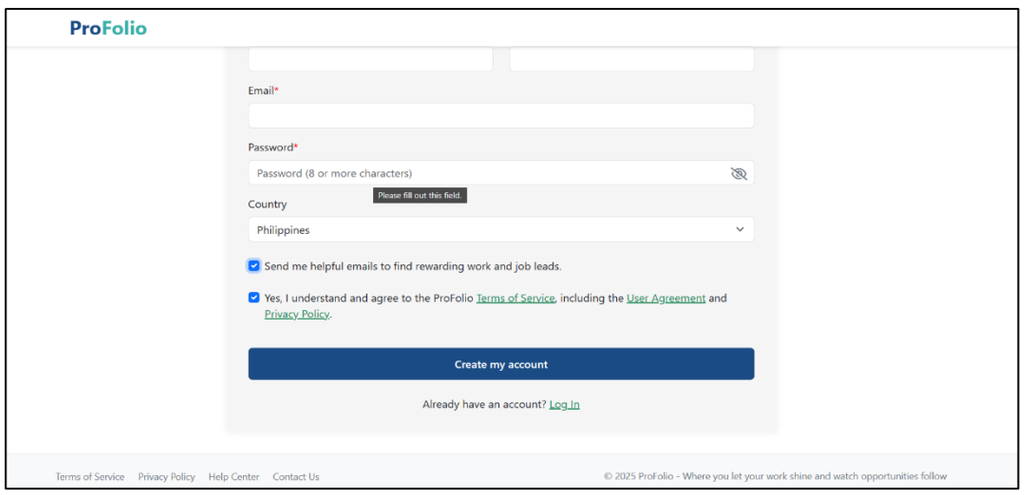
- **Register – as a Freelancer**

➢ **Feature:** Freelancer Registration with Google Sign-In Integration

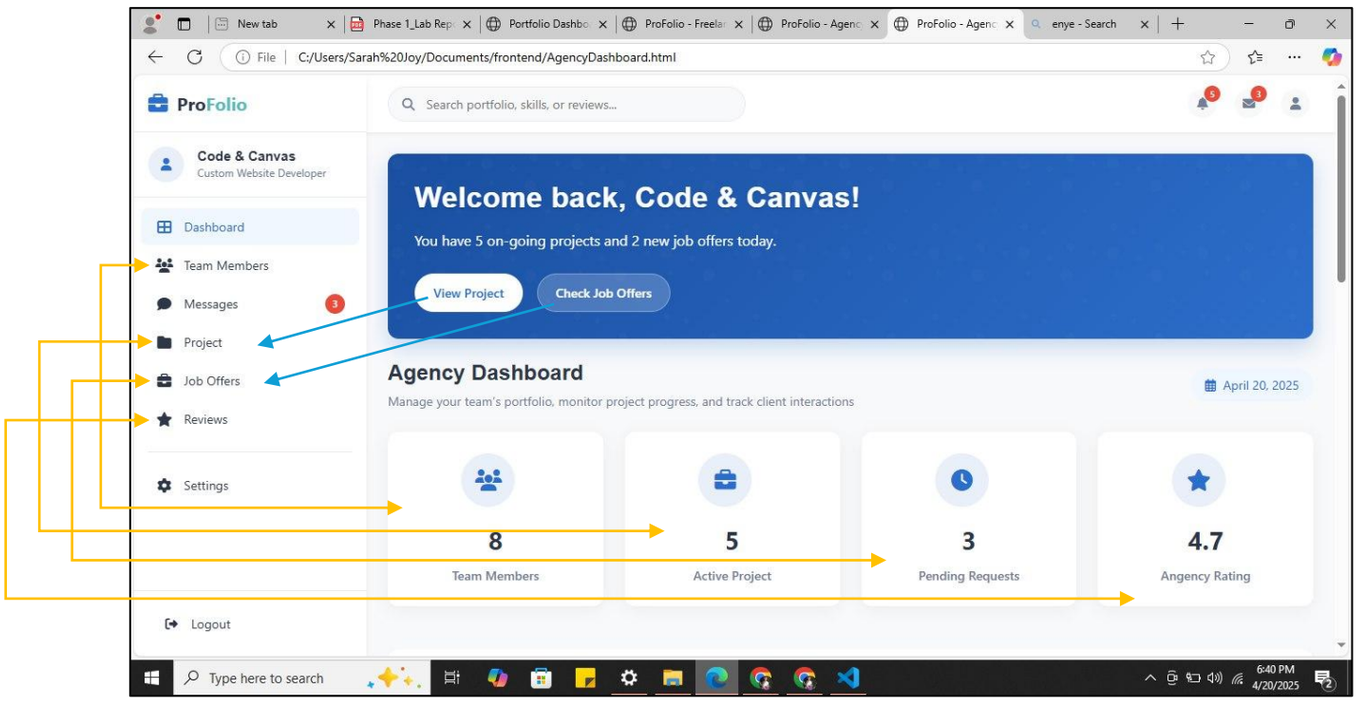
➢ **Use Case:** Enables freelancers to sign up using traditional input fields or by connecting their Google account for faster registration. This improves the user experience and minimizes manual data entry.

Let users sign in or register using their Google account through a secure process. This saves time and avoids manual data entry. Once authenticated, Google shares basic info like name and email with the platform to complete registration or login.

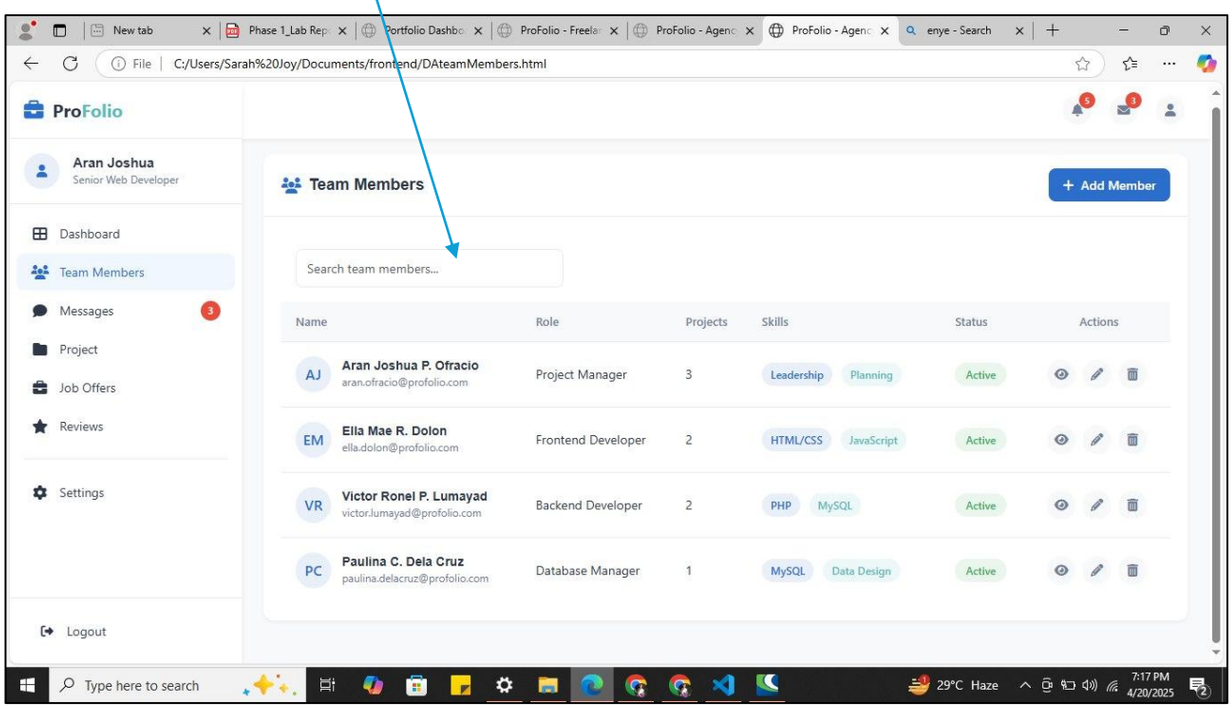




- **Agency – Dashboard**
  - **Feature:** Overview of Agency Dashboard
  - **Use Case:** The dashboard provides agencies an all-in-one view of ongoing projects, available job offers, and tools to search for, add, and manage team members.

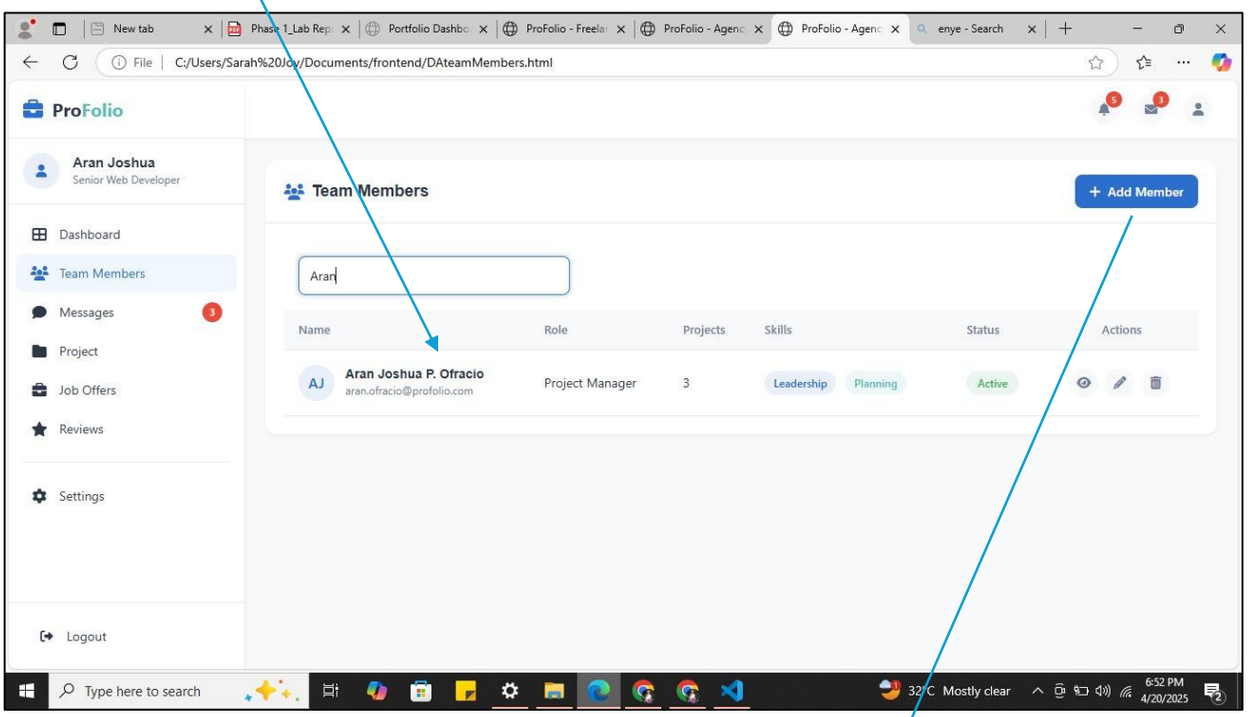


Enables member search.

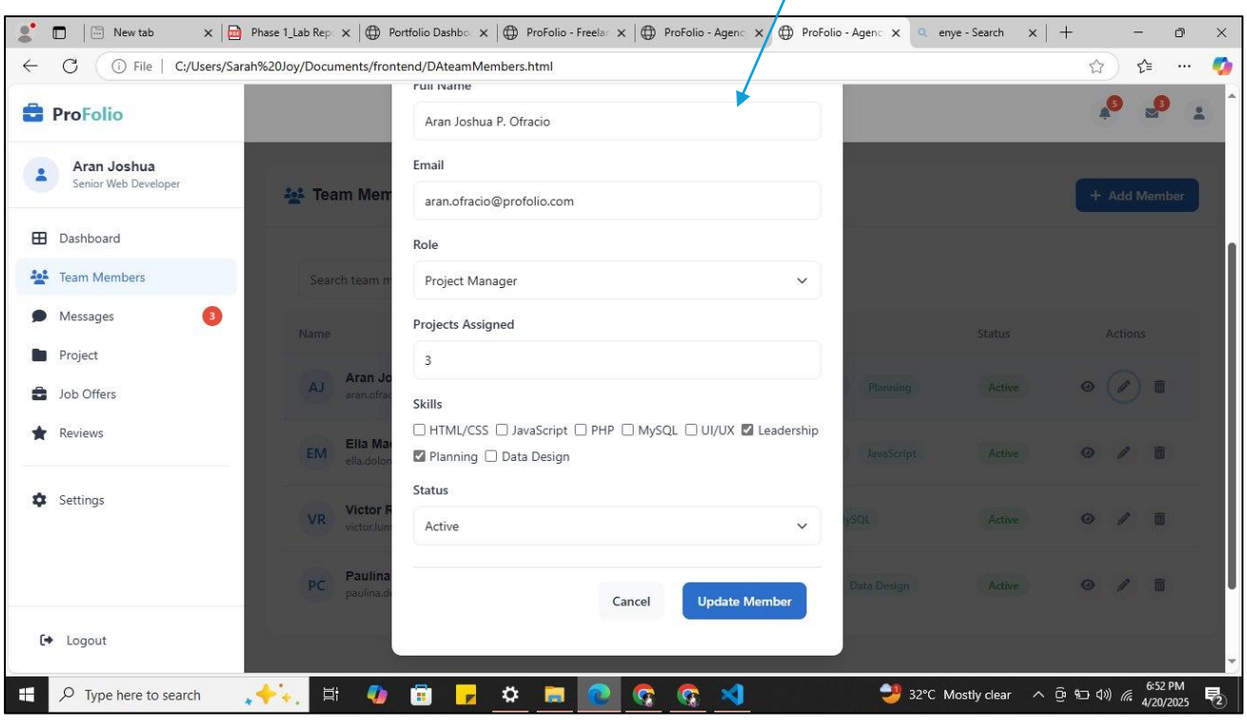


- **Agency – Team Members**
  - **Feature:** Overview of Team Management
  - **Use Case:** Enables agencies to view, organize, and manage their internal team members within the platform.

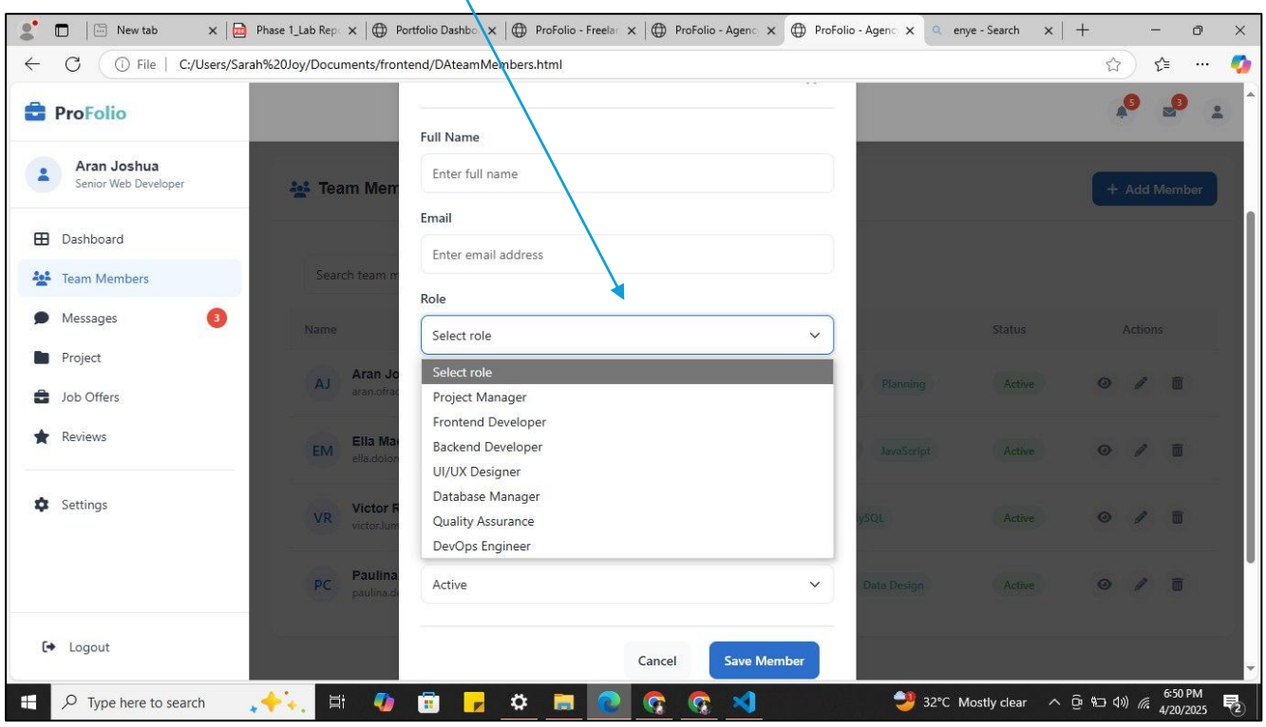
Displays the member being searched for.



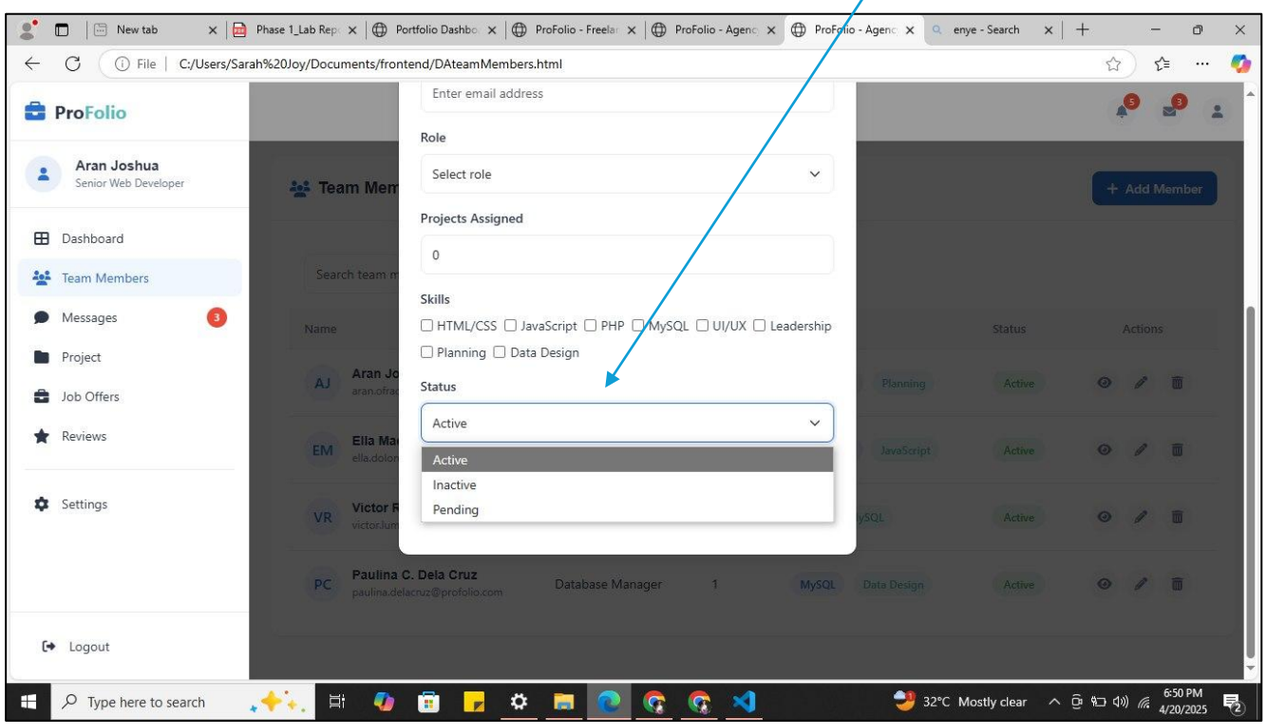
Enables agencies to add members to their teams, input member information, set role and status, and assign projects.

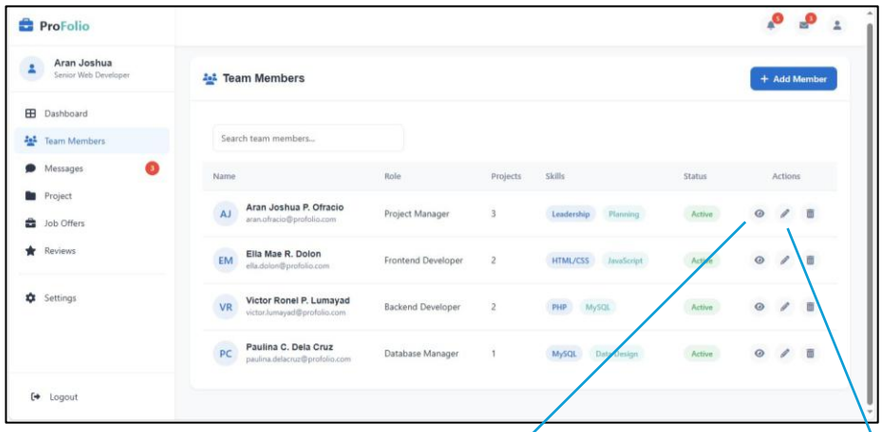


A dropdown containing multiple roles a member can be assigned to.



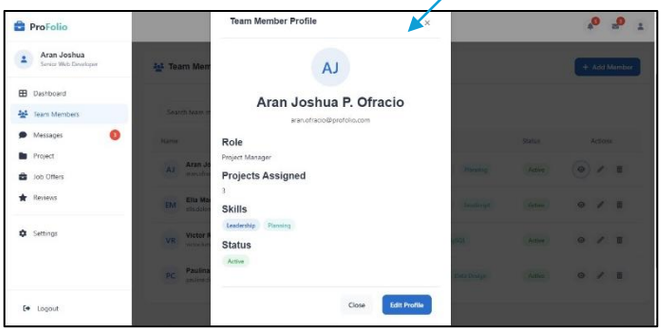
A dropdown menu for selecting a status.

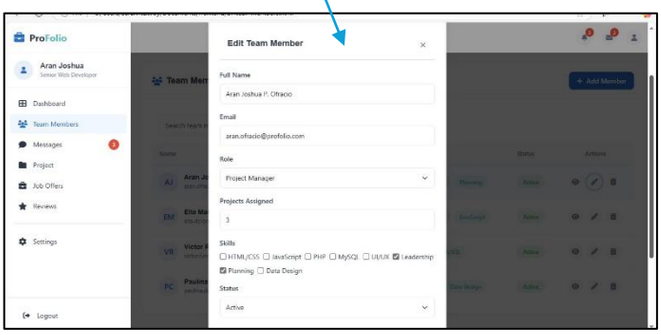


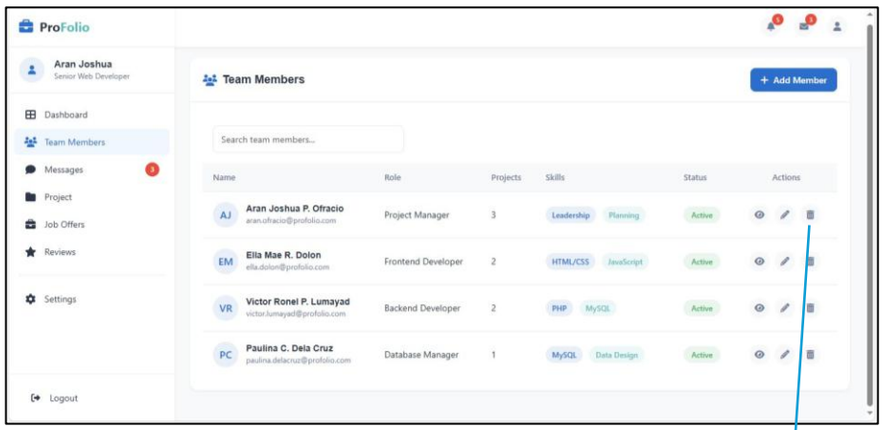


An icon to view the member's profile.

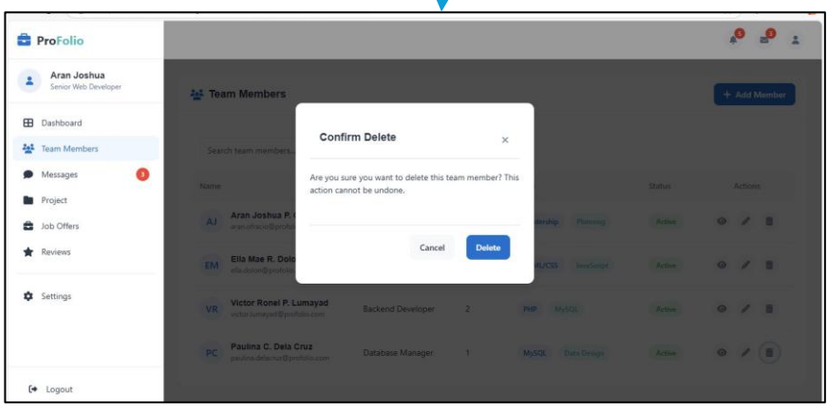
Enables editing of the member's profile.







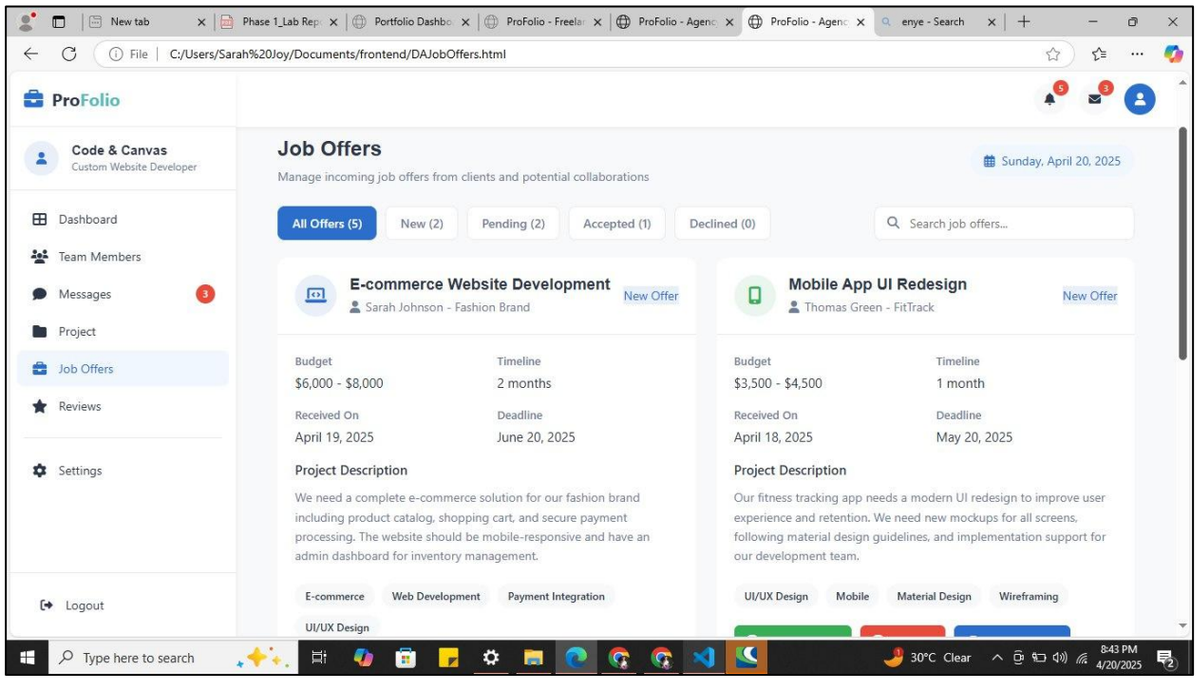
Removes a member from the team.

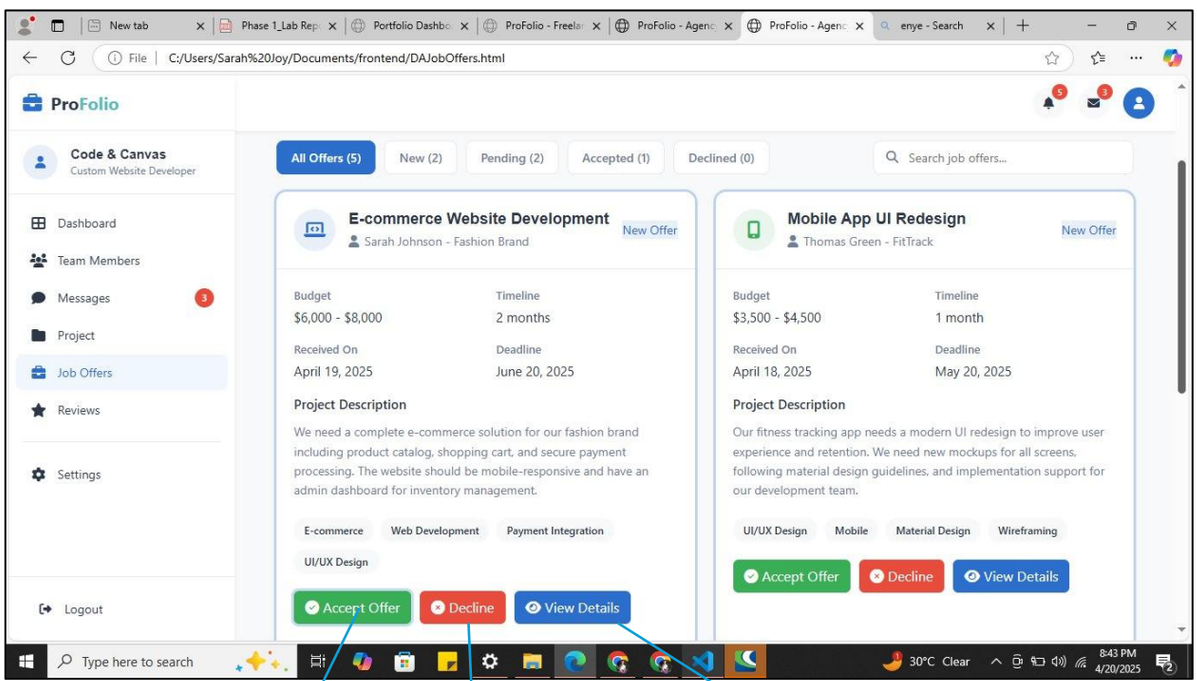


- **Job Offers**

➢ **Feature:** Job Offer Management

➢ **Use Case:** The job offer management section allows agencies to view and manage available job offers, helping them easily track the status of interested clients.



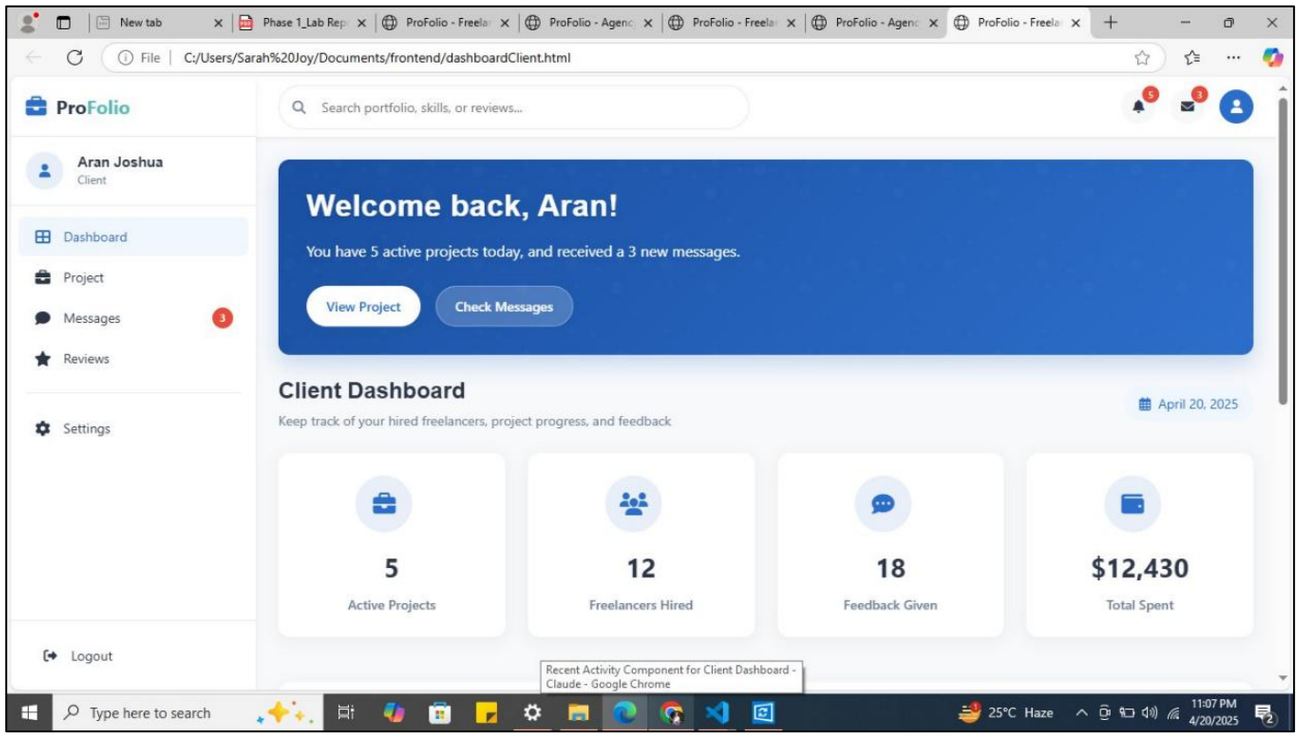


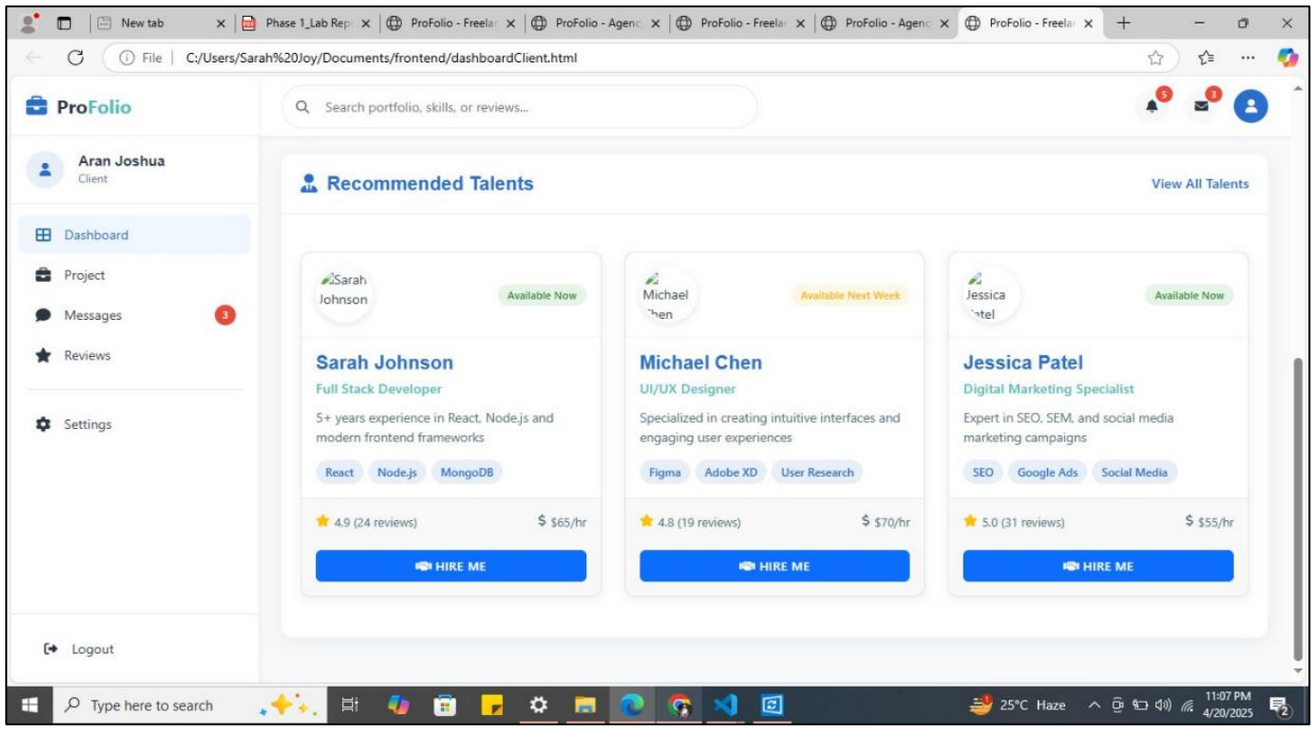
Agree to take on the job and start the project.

Politely reject the offer.

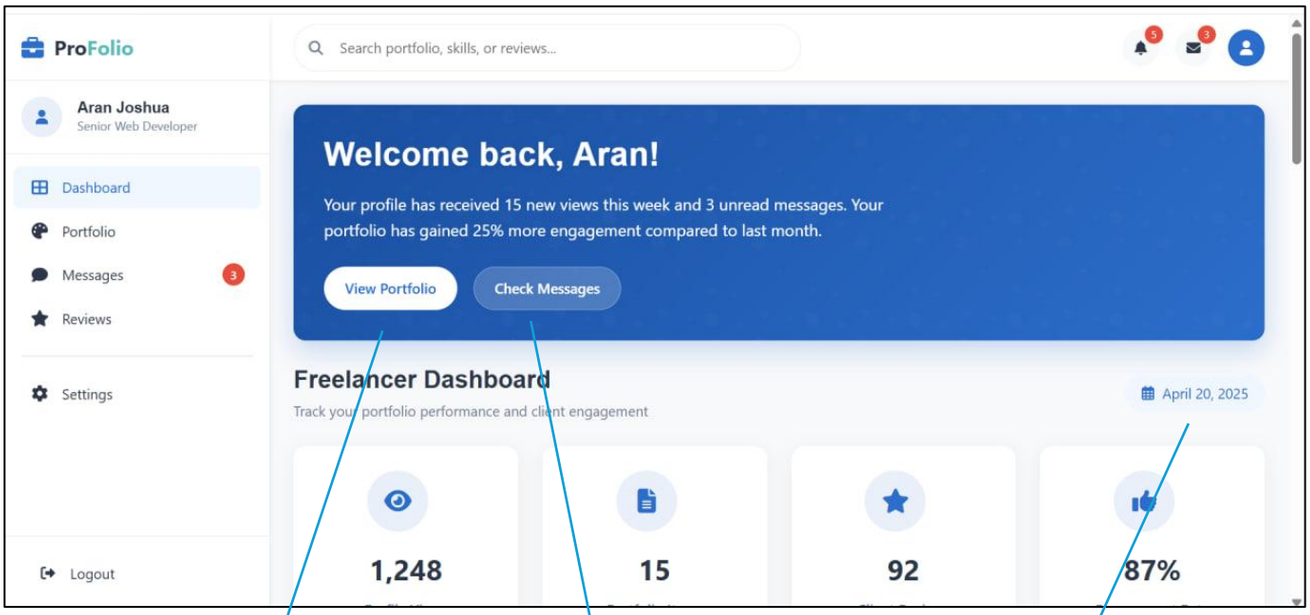
See more information about the job before deciding.

- **Client Dashboard**
  - **Feature:** Overview of Client Dashboard
  - **Use Case:** The dashboard provides clients with a centralized view of active projects, total spending, feedback history, and freelancers hired, while also offering recommended talents with a quick “Hire Me” option for easy hiring.





- **Freelancer - Dashboard**
  - Feature: Overview panel and navigation hub
  - Use Case: This functionality gives freelancers a summary of recent activity, including profile views, messages, and engagement stats. It helps users stay informed about their performance and navigate to key actions like viewing the portfolio or checking messages.



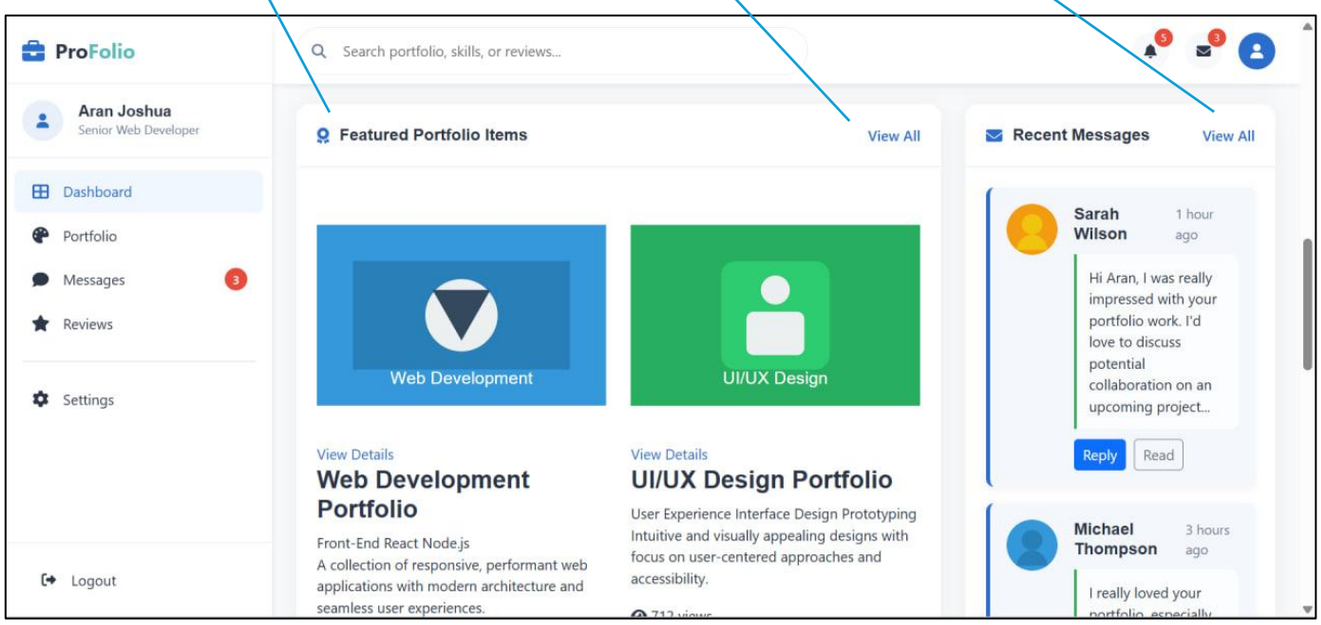
Redirects to the 'Portfolio' tab.

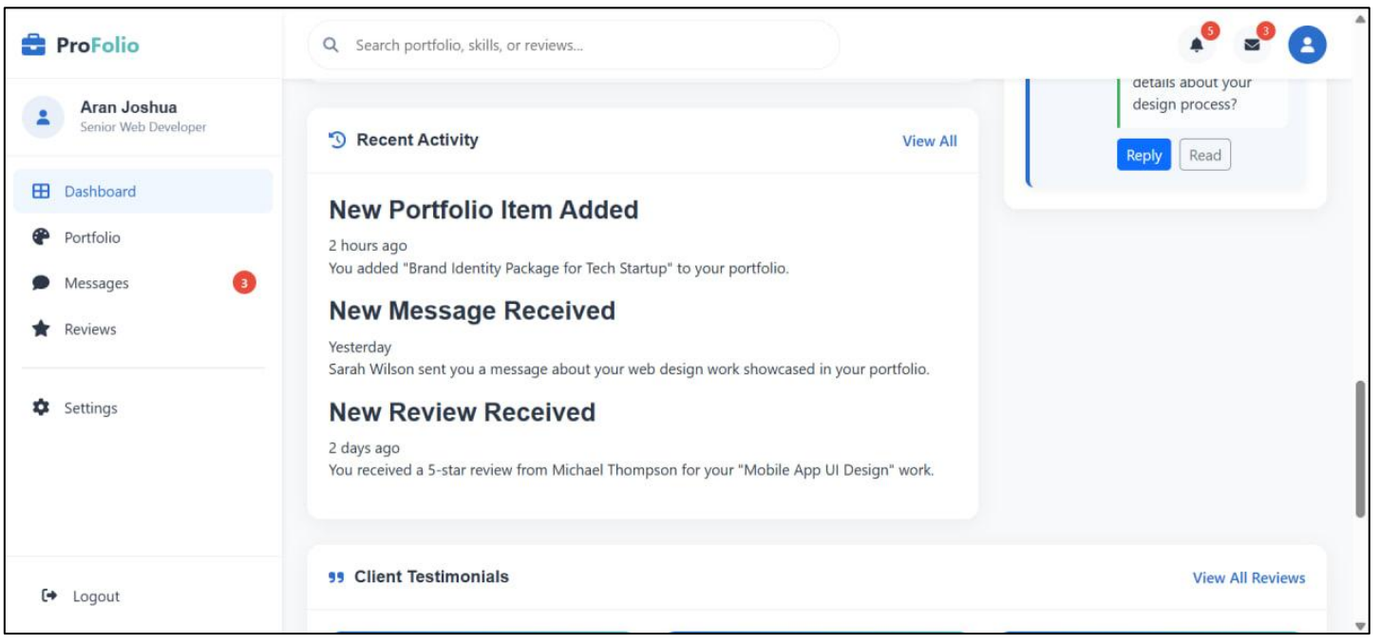
Redirects to the 'Messages' tab.

Displays real-time date for users to keep track of their schedules and time-sensitive agendas.

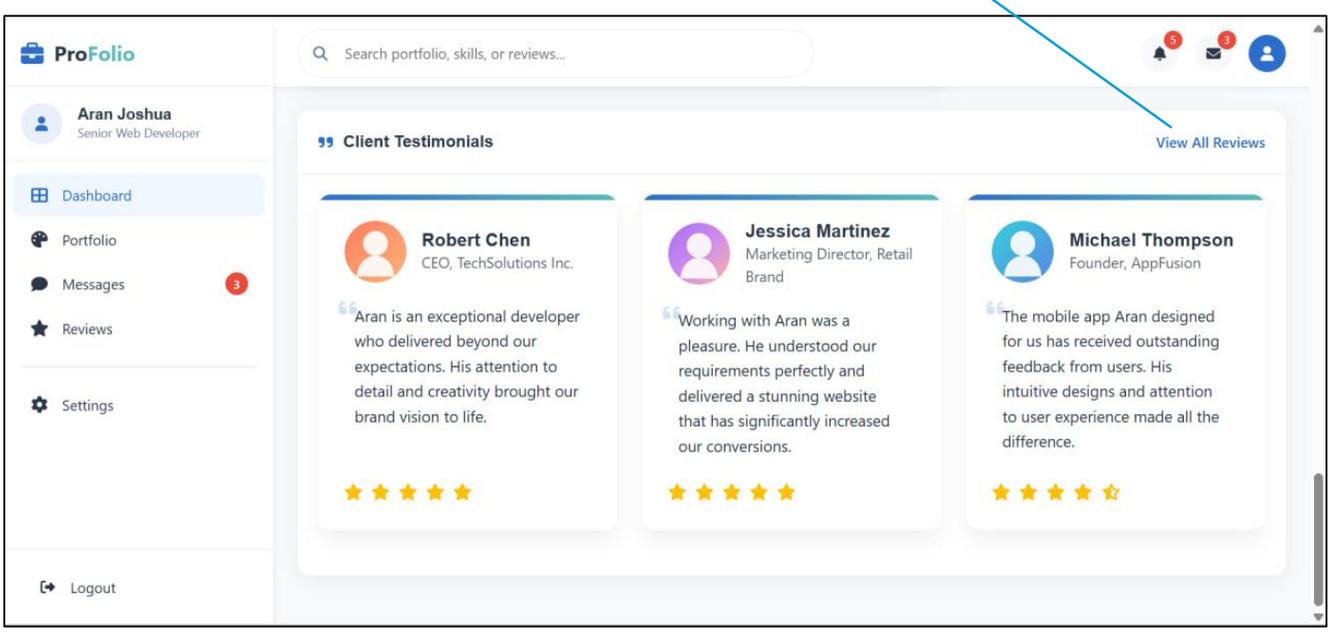
Provides a preview of some of the user's main portfolios that they want to showcase.

Redirects to 'Messages' tab.



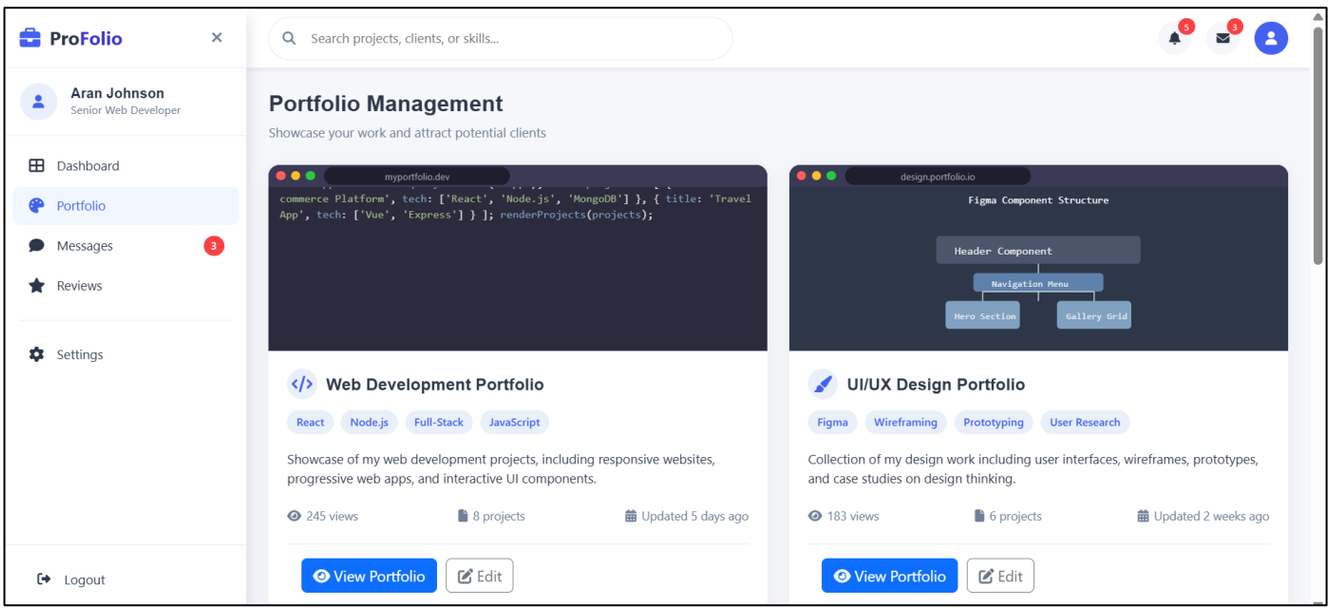


Redirects to 'Reviews tab.

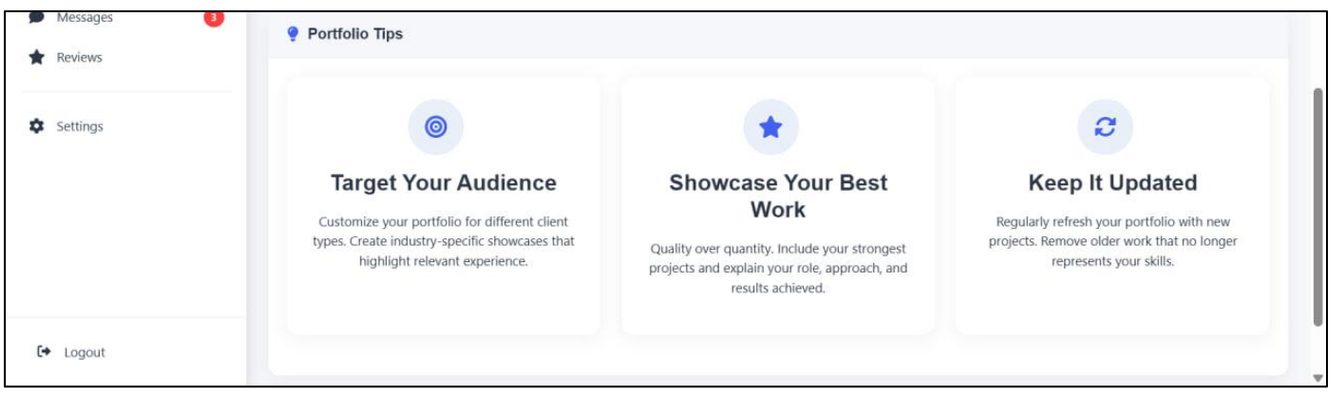


- **Freelancer – Portfolio**
  - **Feature:** Profile and work showcase section
  - **Use Case:** This functionality allows freelancers to present their skills, experience, and completed work. It helps attract potential clients by displaying relevant projects and enhancing the freelancer's professional credibility.

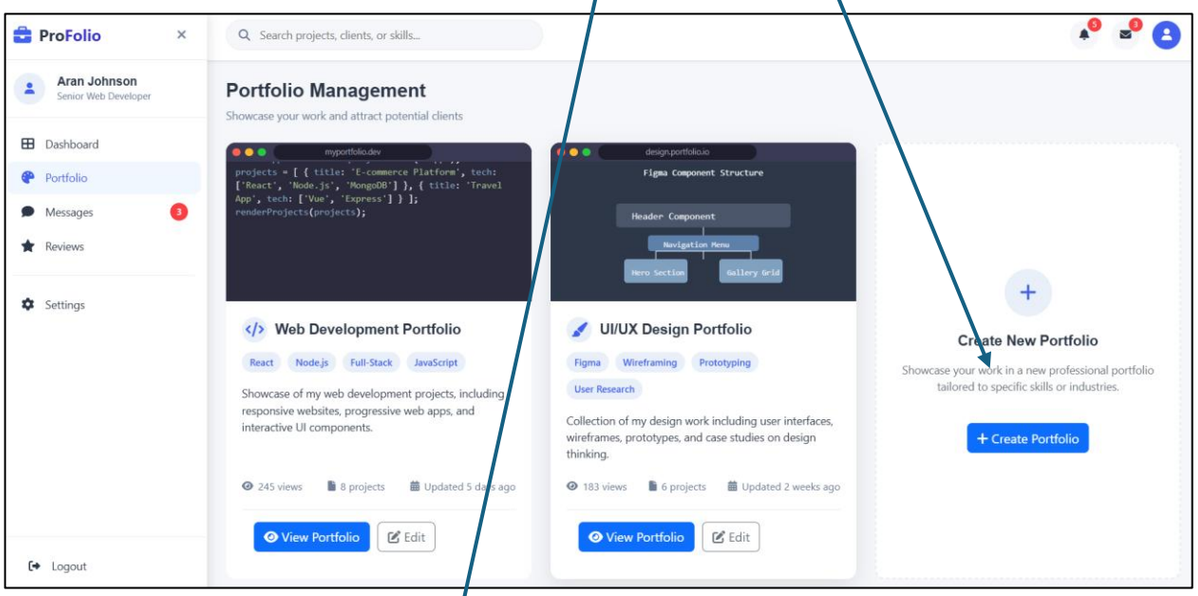
The **Portfolio Management** interface shows categorized project portfolios with tags, view counts, update timestamps, and options to view or edit entries. This section helps users present their work to attract potential clients.

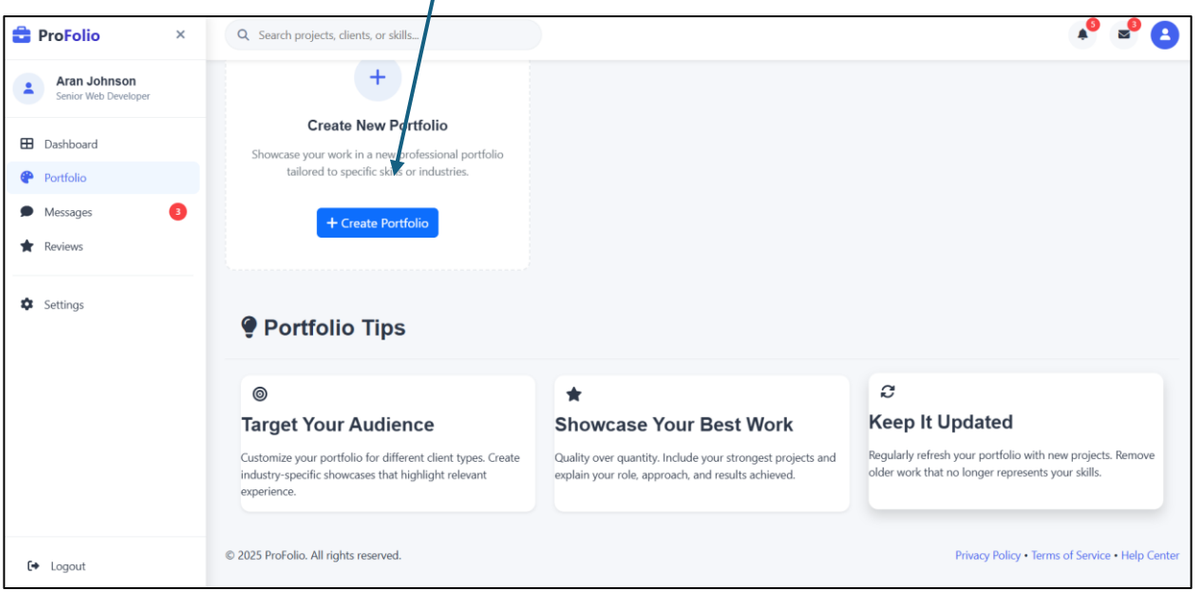


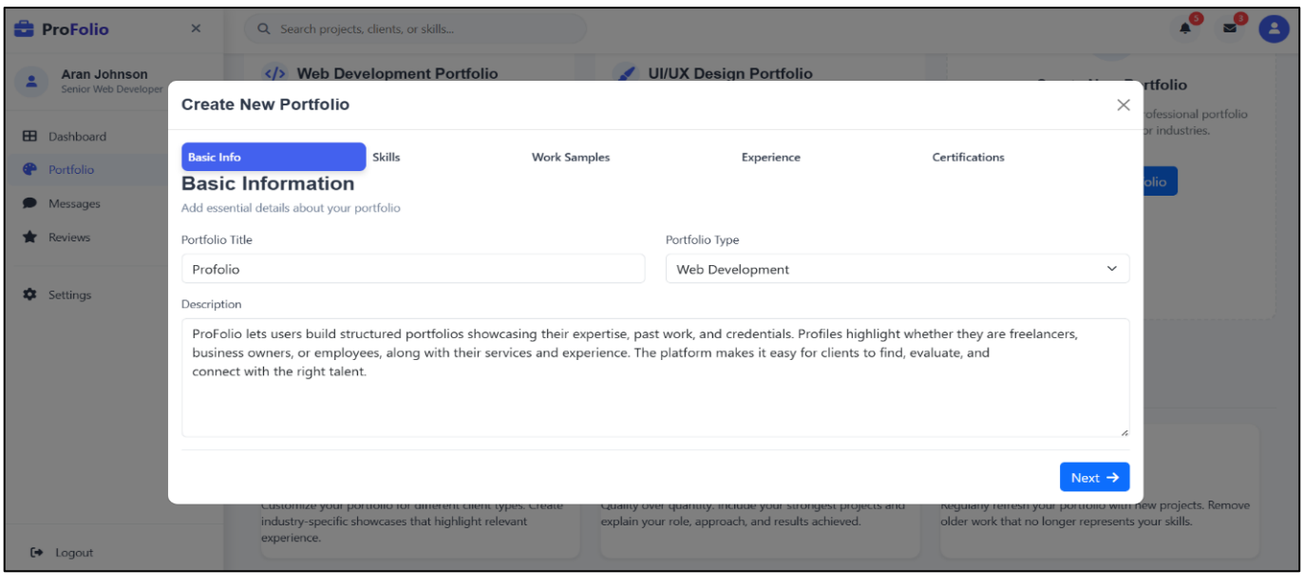
The **Portfolio Tips** section, offering best practices for optimizing project showcases. It highlights three key suggestions: targeting specific audiences, presenting top-quality work, and keeping portfolio content regularly updated.



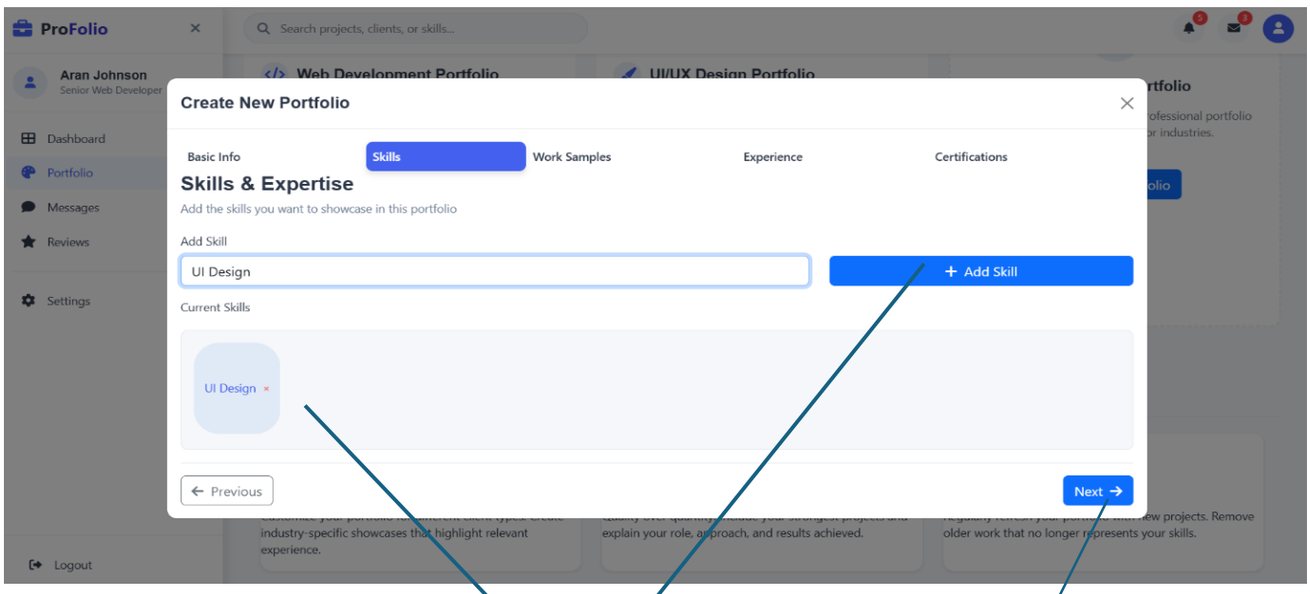
Starts the multi-step form for building a portfolio.





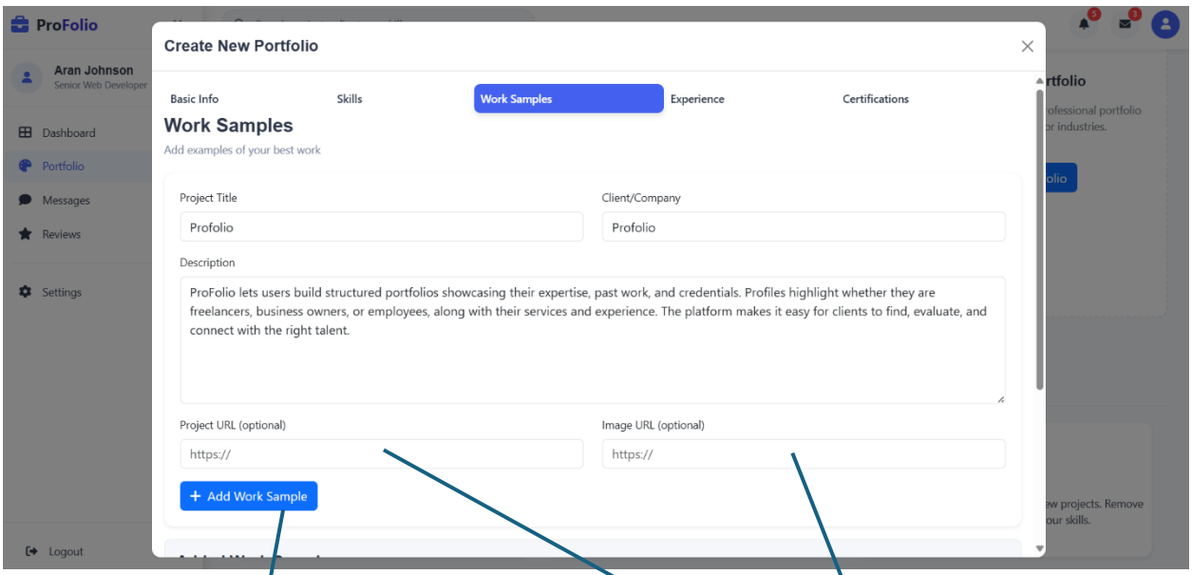


Once the user completes filling out the **Basic Information** section, they can proceed to the next step of the portfolio builder.



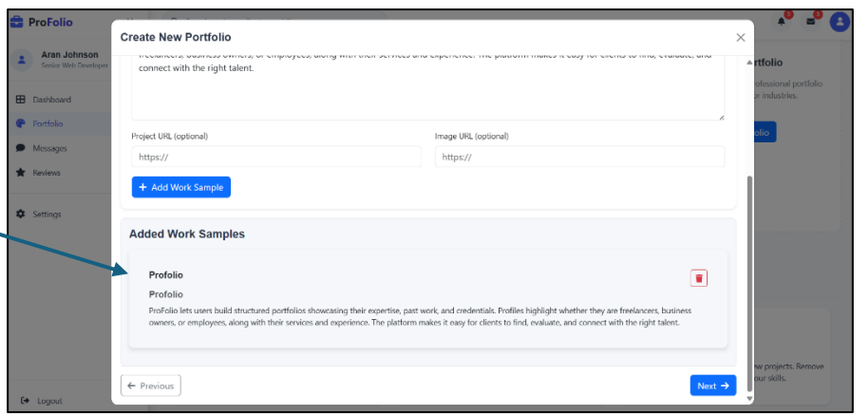
Once a skill is added, it will immediately appear in the **Current Skills** section.

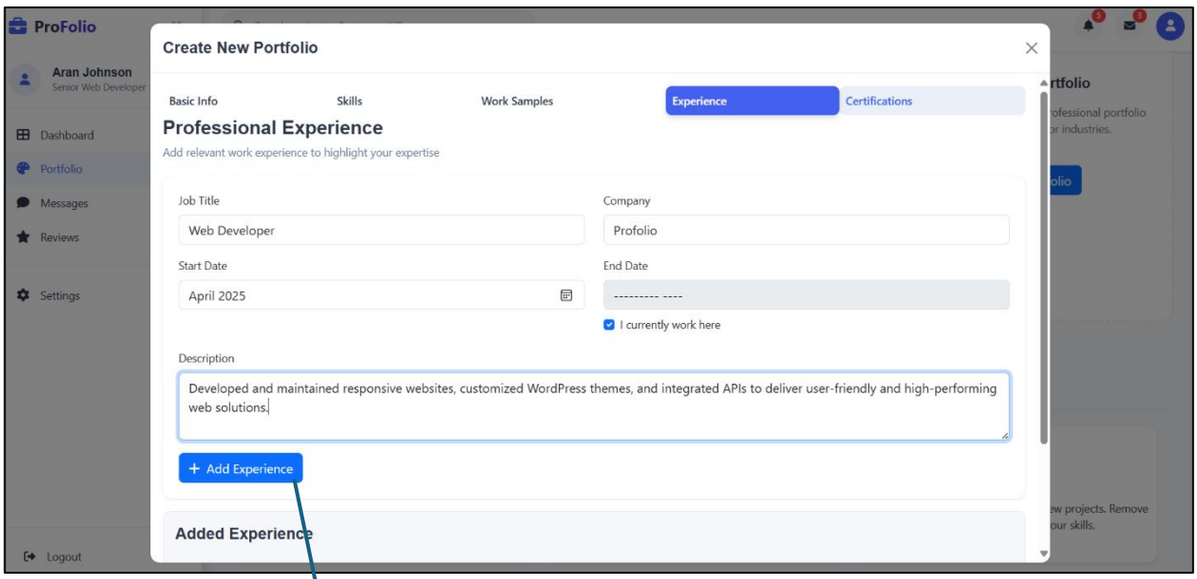
The system proceeds to the next step of the portfolio builder.



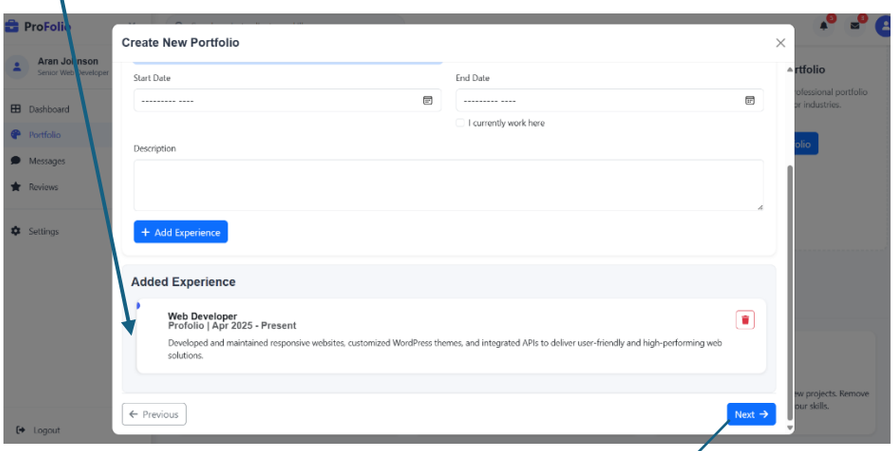
Users can also add a **Project URL** and an **Image URL** when submitting a work sample. These will be included and displayed in the **Added Work Samples** section.

Once the user completes the work sample form and clicks the **“Add Sample”** button, the entered sample is immediately added and displayed in the **“Added Work Samples”** section.

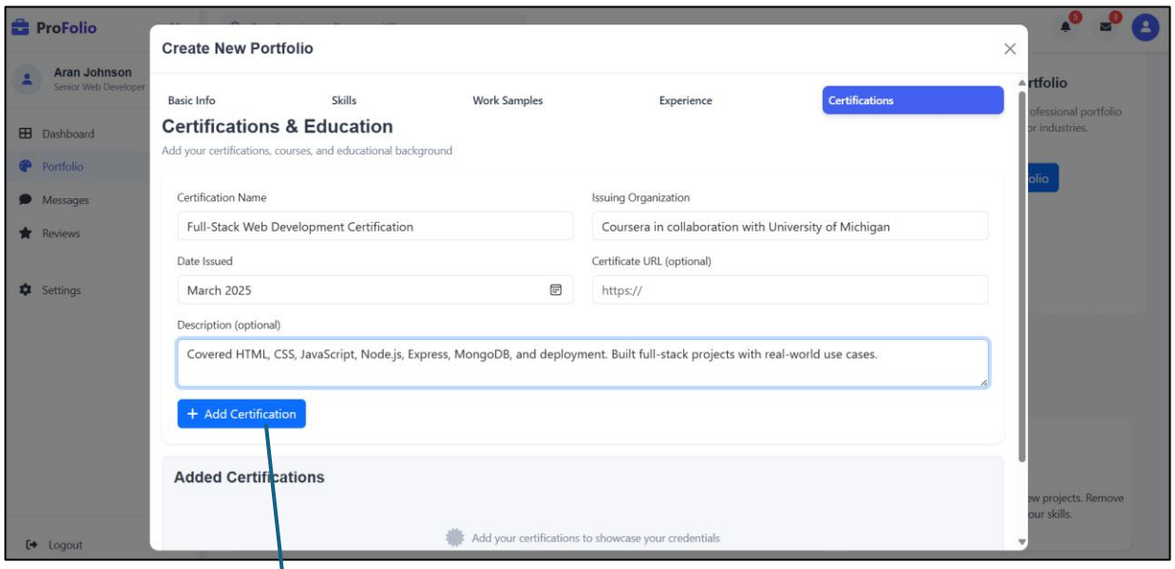




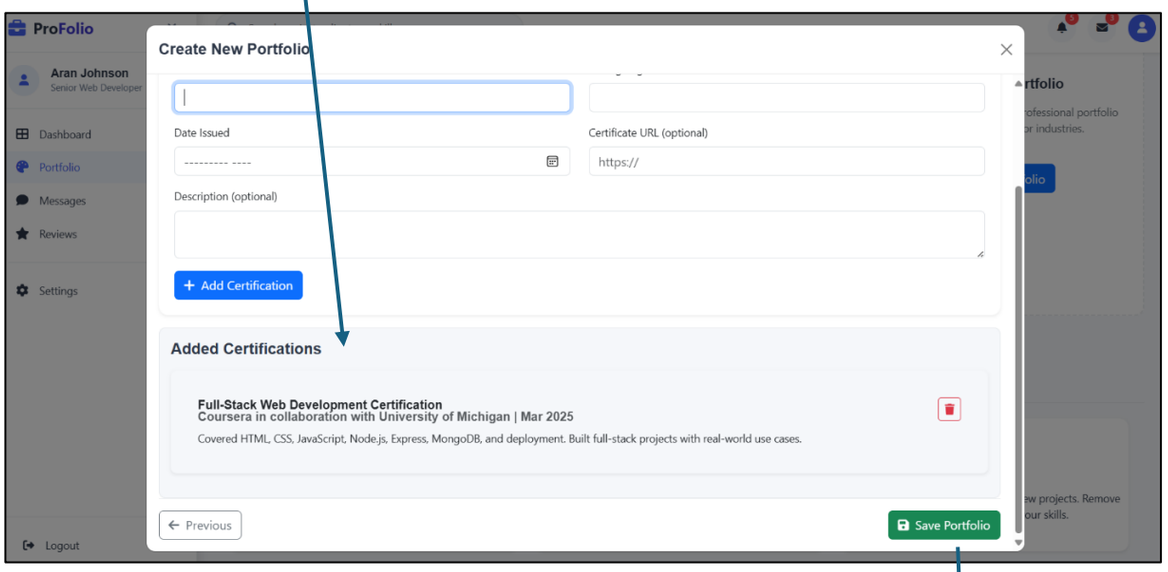
Once the user completes the professional experience sample form and clicks the **“Add Experience”** button, the entered experience is immediately added and displayed in the **“Added Work Samples”** section.

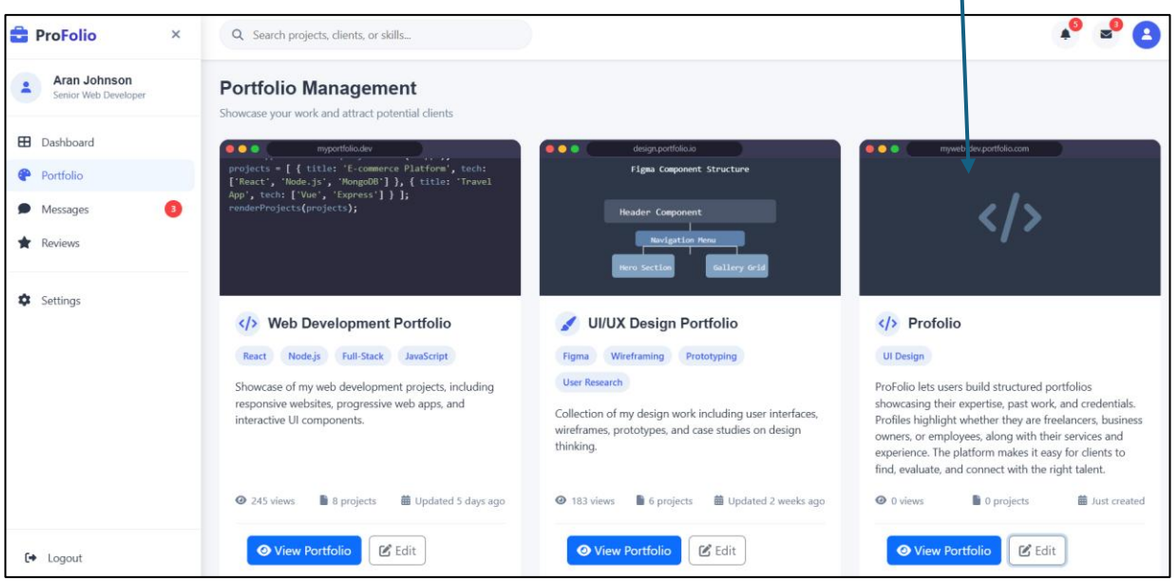


The system proceeds to the next step of the portfolio builder.



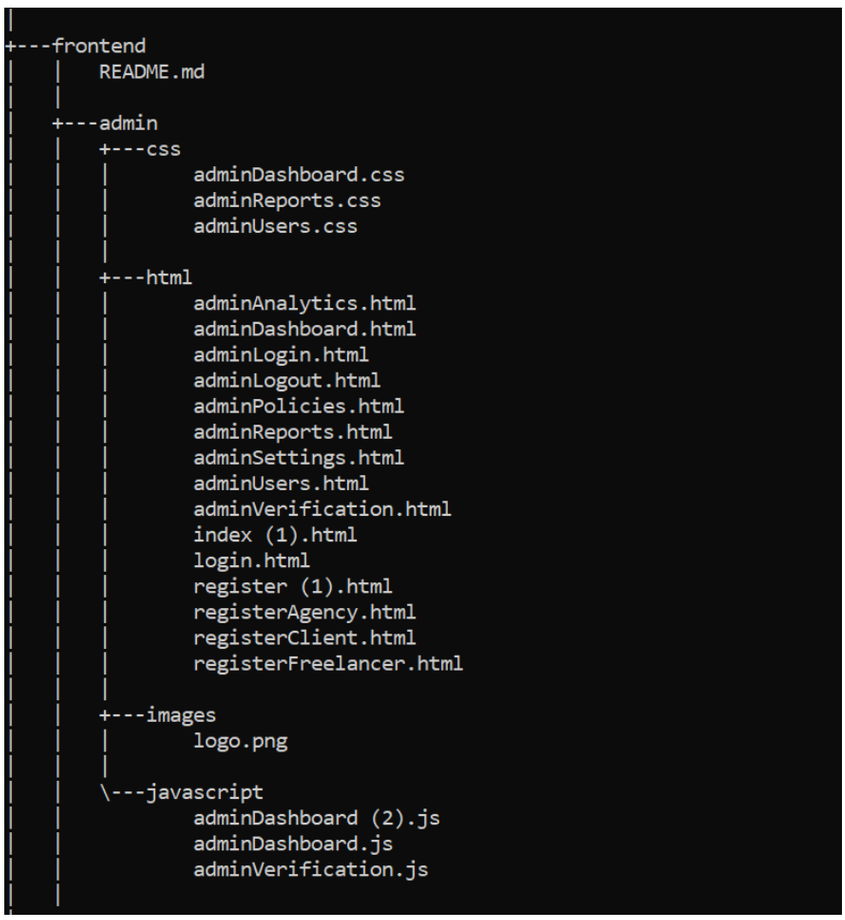
Once the user completes the form and clicks the **"Add Certification"** button, the entered certifications is immediately added and displayed in the **"Added Certifications"** section.

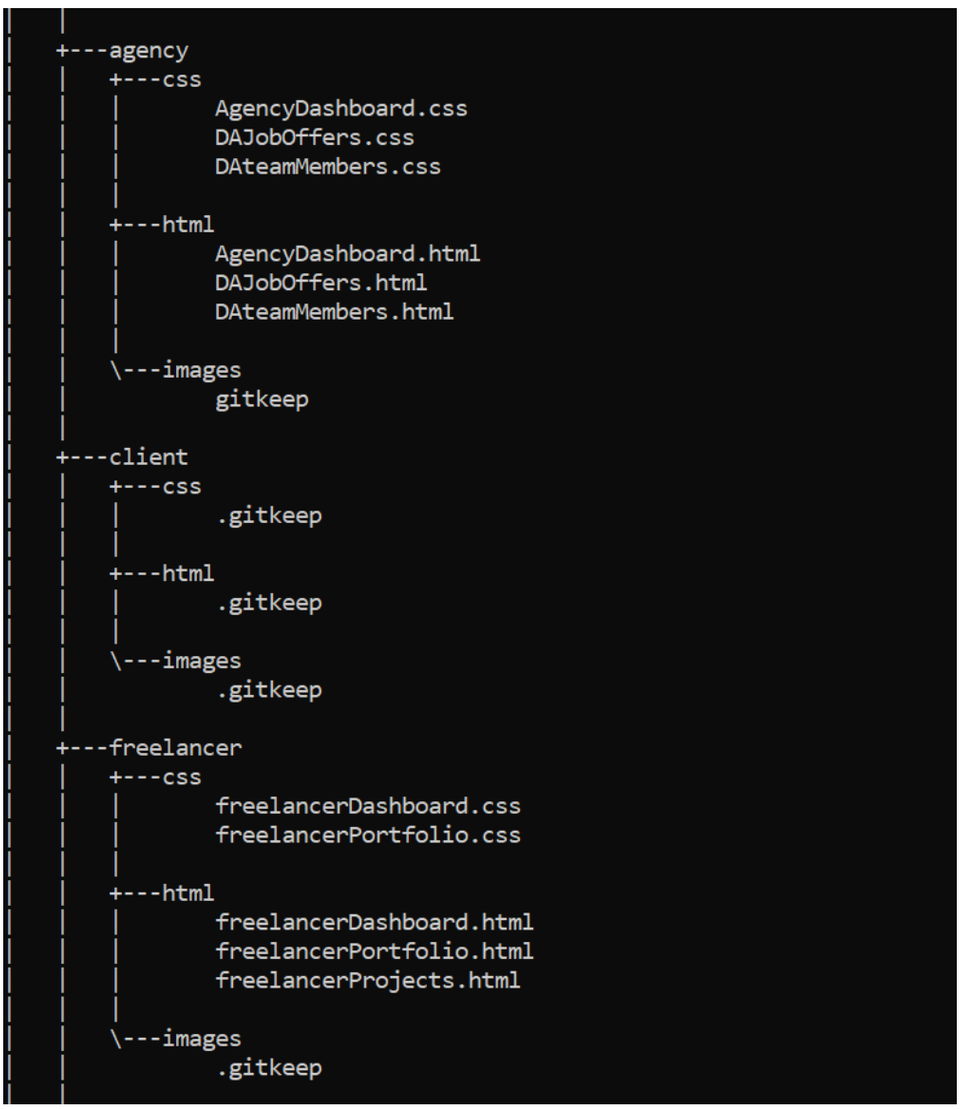




Once a portfolio item is saved, it will automatically appear in the Portfolio Management section with a "Just Created" label to indicate that it was recently

Screenshot of GitHub folder structure





```
|
+---database
|       Table_user.pdf
|
+---docs
|   +---lab11
|   |       Admiin_Verfication.PNG
|   |       Admin_Dashboard.PNG
|   |       Admin_Settings.PNG
|   |       Admin_User.PNG
|   |       Agency_Dashboard.jpg
|   |       analytics.PNG
|   |       homepage.1.png
|   |       homepage.2.png
|   |       homepage.3.PNG
|   |       homepage.4.PNG
|   |       homepage.5.PNG
|   |       homepage.6.PNG
|   |       Job_Offers.jpg
|   |       Reported_Users.PNG
|   |       Team_Members.jpg
|   |
|   \---planning
|           Phase-1_Architecture-Diagram_ProFolio.pdf
|           Phase-1_ERD_ProFolio.pdf
|           Phase-1_Project-Proposal_ProFolio.pdf
|           Phase-1_Use-Case-Diagram_ProFolio.pdf
|
```

```
C:.
|   README.md
|
+---backend
|   |   server.js
|   |
|   +---connection
|   |       connection.php
|   |
|   +---CSS
|   |       agency.css
|   |       client.css
|   |       freelancer.css
|   |       index.css
|   |       login.css
|   |       register.css
|   |
|   +---models
|   |       connection.php
|   |
|   +---PHP
|   |       index.php
|   |       login.php
|   |       register.php
|   |       registerAgency.php
|   |       registerClient.php
|   |       registerFreelancer.php
|   |
|   \---routes
|           connection.php
|
```

Overview of Back-end Setup

The backend setup provides the foundation for managing server-side logic, user data, and secure operations in the project. It runs on a local server environment using XAMPP, which includes Apache for handling HTTP requests, PHP for backend scripting, and MySQL for managing the database. The system is designed with a clear project structure that separates logic for different user roles clients, freelancers, and agencies each with their own registration and login functionalities.

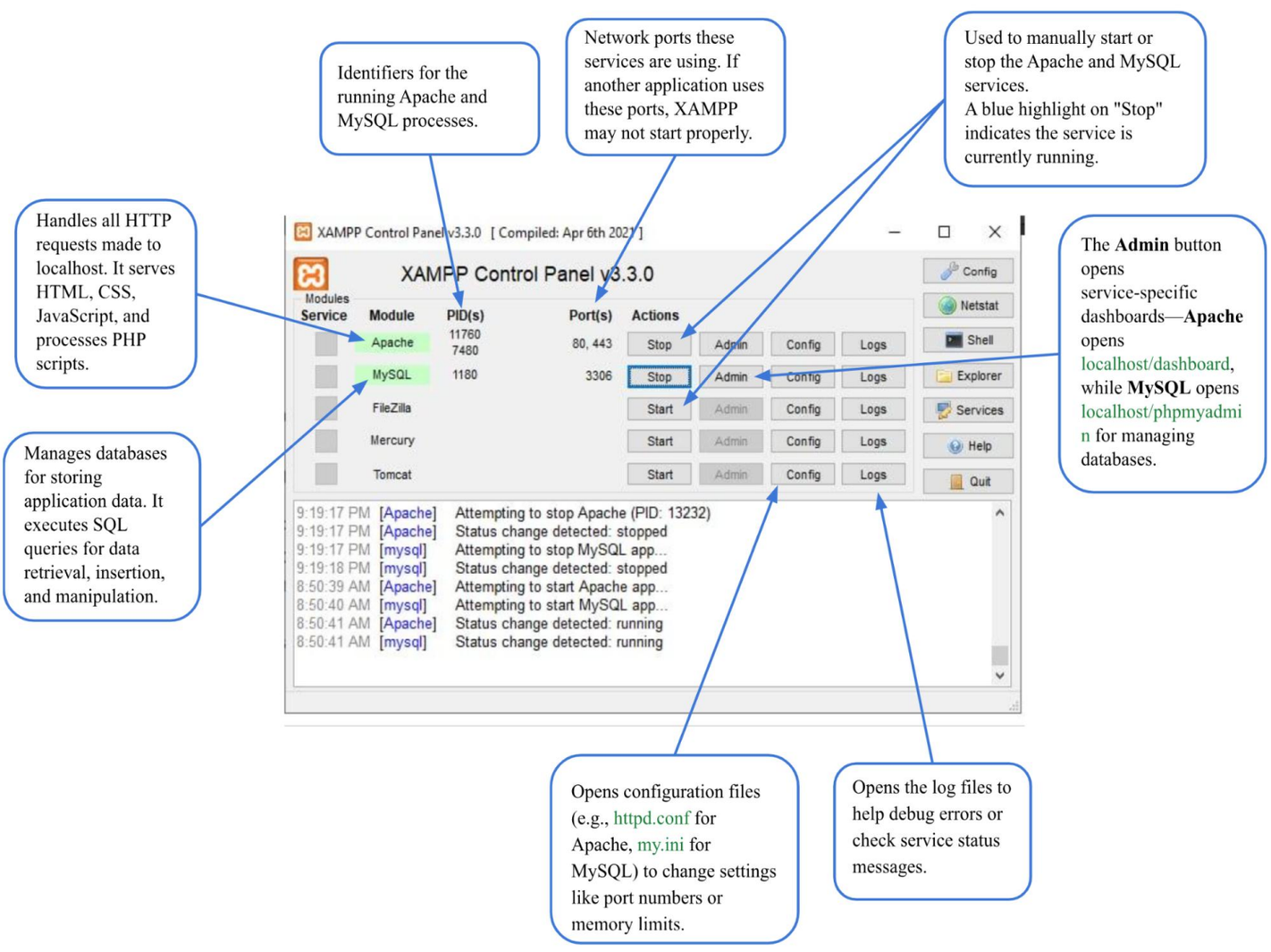
This setup also integrates Google APIs for enhanced features like address autofill or secure login, and includes a database schema to organize user information, job data, and messages. A flow diagram outlines how data moves through the system, from user input on the frontend to processing and storage on the backend, ensuring smooth interaction between all components.

Local Server Environment

We use XAMPP as our local server environment for development, which includes Apache as our web server, PHP as our scripting language, and MySQL as our database system. When we access our web pages through localhost, Apache serves the content and handles HTTP requests. PHP runs on the Apache server to process backend logic, while MySQL manages our data storage and retrieval.

XAMPP provides a user-friendly control panel that allows us to easily start, stop, and configure services. Apache handles all HTML, CSS, JavaScript, and PHP scripts, while MySQL executes queries for data manipulation. The Admin button opens dashboards like **localhost/dashboard** for Apache and **phpMyAdmin** for MySQL, enabling us to manage our server and database visually. Additionally, we can access configuration files and log files to troubleshoot issues or modify default settings such as port numbers and memory limits. This setup is essential for testing dynamic websites and applications before deployment.

ProFolio Structure



The backend architecture of the Profolio project follows a modular and organized structure designed to promote maintainability, scalability, and ease of collaboration. The root directory /PROFOLIO/ is divided into several key subdirectories and PHP scripts, each serving a specific purpose within the system.

```
/PROFOLIO/
├── connection/
│   └── connection.php         → central DB connection file
├── CSS/                       → styling files (e.g., main.css)
├── dashboard/                 → user dashboard interface
├── PHP/
│   ├── index.php              → landing page (entry point)
│   ├── login.php              → login logic (backend)
│   ├── register.php           → general registration handler
│   ├── registerAgency.php     → agency-specific registration
│   ├── registerClient.php     → client-specific registration
│   ├── registerFreelancer.php → freelancer-specific registration
├── uploads/                   → stores uploaded files
```

/connection/

- connection.php
  This file establishes a centralized MySQL database connection using the mysqli extension. By isolating database connectivity here, it promotes code reusability and ensures that changes to connection parameters (like host, username, or password) only need to be updated in one place.

/CSS/

- This directory is dedicated to frontend styling, including files such as main.css. It provides a central location for all design-related resources, ensuring that styling is separated from logic and functionality. This aligns with best practices for separation of concerns.

/dashboard/

- Contains files related to the user dashboard interface. Once a user successfully logs in or registers, they are directed to this area where they can interact with the platform (e.g., view messages, update profiles, track activity). This modular approach allows for future expansion such as separate dashboards for freelancers, clients, and agencies.

/PHP/

This folder contains the core backend logic, with each script fulfilling a specific function in the authentication and registration workflow:

- index.php
  Acts as the entry point or landing page of the application. It may contain redirection logic, basic navigation, or initial session checks.

- login.php
  Handles the login logic, including input validation, authentication against the database, session creation, and user role redirection.

- register.php
  Serves as a general registration handler, possibly including logic to route users based on the type they select (freelancer, client, or agency) and basic validation common to all types.

- registerAgency.php
  Responsible for agency-specific registration logic. This could include collecting and validating additional fields such as company name, registration documents, or team member details.

- registerClient.php
  Manages the registration process for clients, likely capturing information such as company or individual client details and project preferences.

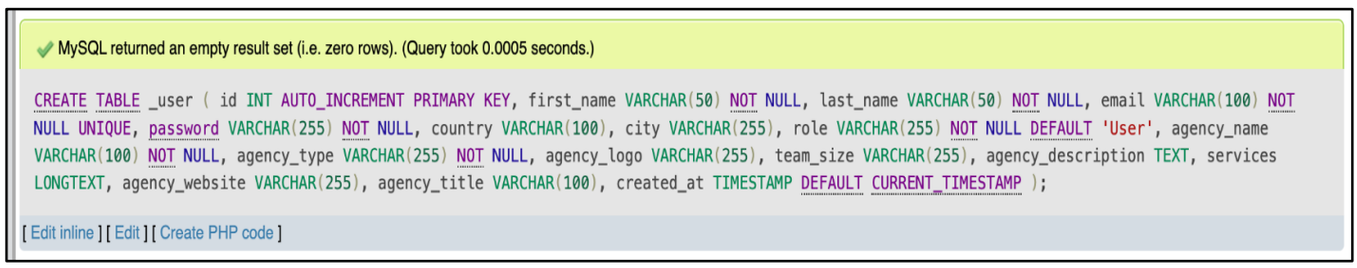
- registerFreelancer.php
  Dedicated to freelancer registration, possibly handling fields such as skills, experience, portfolio links, and availability.

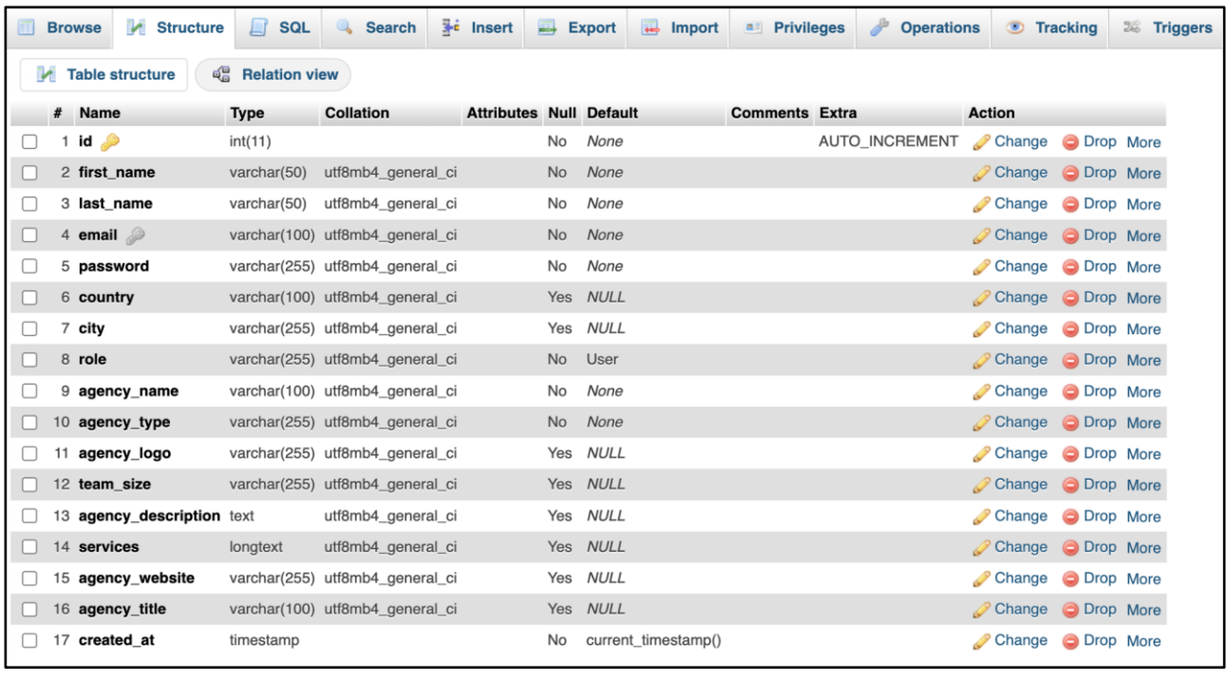
/uploads/

- This directory is used to store user-uploaded files such as:
  - Profile images
  - Resumes
  - Portfolio samples

Database

The Profolio database is designed as the backbone of the platform, efficiently managing all critical user and business data for a freelancing ecosystem that connects agencies, freelancers, and clients. It is built using MySQL, with a schema structured for scalability, flexibility, and performance. At its core, the database is focused on supporting a role-based user system, where individuals can register as freelancers or clients, and organizations can onboard as agencies. Each user regardless of their role is represented within a unified data model that ensures easy access control, extensibility, and secure authentication





Browse | Structure | SQL | Search | Insert | Export | Import | Privileges | Operations | Tracking | Triggers

Table structure | Relation view

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	id	int(11)			No	None		AUTO_INCREMENT	Change Drop More
2	first_name	varchar(50)	utf8mb4_general_ci		No	None			Change Drop More
3	last_name	varchar(50)	utf8mb4_general_ci		No	None			Change Drop More
4	email	varchar(100)	utf8mb4_general_ci		No	None			Change Drop More
5	password	varchar(255)	utf8mb4_general_ci		No	None			Change Drop More
6	country	varchar(100)	utf8mb4_general_ci		Yes	NULL			Change Drop More
7	city	varchar(255)	utf8mb4_general_ci		Yes	NULL			Change Drop More
8	role	varchar(255)	utf8mb4_general_ci		No	User			Change Drop More
9	agency_name	varchar(100)	utf8mb4_general_ci		No	None			Change Drop More
10	agency_type	varchar(255)	utf8mb4_general_ci		No	None			Change Drop More
11	agency_logo	varchar(255)	utf8mb4_general_ci		Yes	NULL			Change Drop More
12	team_size	varchar(255)	utf8mb4_general_ci		Yes	NULL			Change Drop More
13	agency_description	text	utf8mb4_general_ci		Yes	NULL			Change Drop More
14	services	longtext	utf8mb4_general_ci		Yes	NULL			Change Drop More
15	agency_website	varchar(255)	utf8mb4_general_ci		Yes	NULL			Change Drop More
16	agency_title	varchar(100)	utf8mb4_general_ci		Yes	NULL			Change Drop More
17	created_at	timestamp			No	current_timestamp()			Change Drop More

As the first step in setting up the Profolio database, we created the _user table. It is the core table that powers all user interactions on the platform. This table serves as the backbone for managing user data, supporting various roles such as freelancers, clients, and agencies within a unified structure. It captures essential user information including names, email, password, and location data, while also accommodating role-based distinctions through the role field. For agency accounts, additional fields such as agency_name, agency_type, services, and team_size enable detailed profiles that help users better showcase their brand and capabilities.

Attributes that make up the _user table:

1. Primary Identification

- id (INT, AUTO_INCREMENT, PRIMARY KEY):
  A unique numeric identifier for each user. It auto-increments with every new record insertion, ensuring consistent and unambiguous user identification across the platform.

2. Personal & Authentication Details

- first_name (VARCHAR(50), NOT NULL):
  Stores the user's first name. Required for personalization and account identification.

- last_name (VARCHAR(50), NOT NULL):
  Stores the user's last name. Also required.

- email (VARCHAR(100), NOT NULL, UNIQUE):
  Functions as the unique login identifier. The UNIQUE constraint ensures no two users can register with the same email.

- password (VARCHAR(255), NOT NULL):
  Stores the user's hashed password (usually hashed using algorithms like bcrypt). The extended length supports modern hashing formats.

3. Location-Based Information

- country (VARCHAR(100), NULLABLE):
  Stores the country associated with the user. Useful for localizing content, filtering users by region, and platform analytics.

- city (VARCHAR(255), NULLABLE):
  Optional city field for more granular geographic categorization.

4. User Role & Access Control

- role (VARCHAR(255), NOT NULL, DEFAULT 'User'):
  Defines the user's function or access level within the platform. Supported values may include:
    - Freelancer
    - Agency Owner
    - Client
    - Admin

5. Agency-Specific Fields

- agency_name (VARCHAR(255), NULLABLE):
  Name of the agency the user represents or manages.

- agency_type (VARCHAR(255), NULLABLE):
  Categorizes the agency (e.g., Design, Development, Marketing). Enables filtering and discovery by type.

- agency_logo (VARCHAR(255), NULLABLE):
  Path or URL to the uploaded agency logo image file.

- team_size (VARCHAR(255), NULLABLE):
  Describes the size of the team. Helps clients understand the scale of the agency.

- agency_description (TEXT, NULLABLE):
  A rich-text field for agencies to describe their background, mission, expertise, or values.

- services (LONGTEXT, NULLABLE):
  Lists the services offered by the agency. Stored as long text to accommodate detailed descriptions or structured data.

- agency_website (VARCHAR(255), NULLABLE):
  Stores a URL to the agency's website for external verification or portfolio browsing.

- agency_title (VARCHAR(100), NULLABLE):
  A custom title displayed on profiles or dashboards for added personalization.

Registration Page

Our registration page serves as the primary entry point for new users to join the Profolio platform. It is designed to onboard users efficiently and securely, whether they are freelancers, clients, or agencies. When a user submits the registration form, our backend PHP script takes over to handle the logic required to validate, process, and store the new account data.

This process begins with capturing all submitted data such as the user's **first name, last name, email, password, country**, and predefined **role** (client, freelancer, or agency_owner). To ensure security, each of these inputs is sanitized using mysqli_real_escape_string() to guard against SQL injection attacks.

This PHP code manages user registration. It first checks if a user's email already exists in the database. If not, it inserts the user's data into the database and redirects the user to the login page. If the email already exists, it alerts the user. However, the original code is vulnerable to SQL injection and should be rewritten using parameterized queries for security.

```php
    // Check if the email already exists
    $checkEmailSql = "SELECT * FROM _user WHERE email = '$email'";
    $result = $con->query($checkEmailSql);

    if ($result->num_rows > 0) {
        echo "<script>alert('Email already exists. Please use a different email.');</script>";
    } else {
        // Insert the data if email doesn't exist
        $sql = "INSERT INTO _user (first_name, last_name, email, password, country, role)
                VALUES ('$firstName', '$lastName', '$email', '$password', '$country', '$role')";

        if ($con->query($sql)) {
            echo "<script>alert('Registration successful!'); window.location.href='login.php';</script>";
        } else {
            echo "Error: " . $con->error;
        }
    }
}
?>
```

- $checkEmailsql = "SELECT * FROM _user WHERE email = '$email'";
  - This line constructs an SQL query to select all columns (SELECT *) from the _user table where the email column matches the value stored in the $email variable. This variable presumably comes from the user's registration form submission.

- $result = $con->query($checkEmailsql);
  - This line executes the SQL query using the database connection object $con (which is assumed to be established elsewhere in the code). The result of the query (either a rowset or an error) is stored in the $result variable.

- if ($result->num_rows > 0) { ... }
  - This condition checks if the query returned any rows. If $result->num_rows is greater than 0, it means an email address matching the user's input already exists in the database.

- echo "<script>alert('Email already exists. Please use a different email.');</script>";
  - If the email already exists, this line uses JavaScript to display an alert message to the user, prompting them to use a different email address.

- } else { ... }
  - This block executes only if the email address is unique (no rows were returned from the previous query).

- $sql = "INSERT INTO _user (first_name, last_name, email, password, country, role) VALUES ('$firstName', '$lastName', '$email', '$password', '$country', '$role')";
    - This line constructs an SQL query to insert a new user record into the _user table. It inserts values for first_name, last_name, email, password, country, and role, all taken from variables presumably populated from the user's registration form. Important Security Note: This code is vulnerable to SQL injection. Using parameterized queries is crucial to prevent this vulnerability.

- if ($con->query($sql)) { ... }
    - This line executes the insertion query.

- echo "<script>alert('Registration successful!'); window.location.href='login.php';</script>";
    - If the insertion was successful, this line displays a success message using JavaScript and redirects the user to the login.php page.

- } else { echo "Error: " . $con->error; }
    - If the insertion fails, this line displays an error message including the database error message from $con->error.

This PHP code handles user registration via Google Sign-In. It checks if the user's email already exists; if not, it adds the user to the database (leaving the password field blank as Google handles authentication). If the email exists, it redirects the user to the login page. Crucially, the original code needs improvement to prevent SQL injection vulnerabilities. The improved version uses prepared statements for enhanced security.

```php
// Handle Google Sign-Up

if ($_SERVER["REQUEST_METHOD"] == "POST" && isset($_POST['google_signup'])) {
  $firstName = isset($_POST['first_name']) ? mysqli_real_escape_string($con, $_POST['first_name']) : '';
  $lastName = isset($_POST['last_name']) ? mysqli_real_escape_string($con, $_POST['last_name']) : '';
  $email = isset($_POST['email']) ? mysqli_real_escape_string($con, $_POST['email']) : '';
  $role = 'client';

  if (!empty($email)) {
      $checkUserSql = "SELECT * FROM _user WHERE email = '$email'";
      $result = $con->query($checkUserSql);

      if ($result->num_rows === 0) {
        // No user yet - proceed with Google sign-up
        $sql = "INSERT INTO _user (first_name, last_name, email, password, country, role)
                VALUES ('$firstName', '$lastName', '$email', '', '', '$role')";
        $con->query($sql);
        echo json_encode(['status' => 'success', 'message' => 'Signed in with Google!']);
        exit;
    } else {
        // Email already exists - do NOT allow Google sign-up again
        echo json_encode([
          'status' => 'redirect',
          'message' => 'Email already registered. Redirecting to login...',
          'redirect_url' => 'login.php'
      ]);

        exit;
    }

  }

  // Final fallback to ensure this block does NOT fall through to regular registration
  exit;
}
```

- if ($_SERVER["REQUEST_METHOD"] == "POST" && isset($_POST['google_signup'])) { ... }
    - This checks if the request method is POST and if the google_signup key exists in the $_POST array (Google sign-up form was submitted).

- $firstName = isset($_POST['first_name']) ? mysqli_real_escape_string($con, $_POST['first_name']) : '';
- $lastName = isset($_POST['last_name']) ? mysqli_real_escape_string($con, $_POST['last_name']) : '';
- $email = isset($_POST['email']) ? mysqli_real_escape_string($con, $_POST['email']) : '';
    - These lines retrieve values from the POST data ($_POST), representing user information (first name, last name, email), and sanitize them using mysqli_real_escape_string($con, ...) to prevent SQL injection. $con is assumed to be a database connection object. Important: While this sanitization helps, it's still not as robust as prepared statements.

- if (empty($email)) { ... }
    - This checks if the email is empty. If so, it skips the database check (although this is unusual behavior and might indicate a flaw in the input handling).

- $checkUserSql = "SELECT * FROM _user WHERE email = '$email'";
    - This constructs an SQL query to check if a user with the given email already exists.

- $result = $con->query($checkUserSql);
    - This executes the query.

- if ($result->num_rows == 0) { ... }
    - This checks if no user with the email was found.

- $sql = "INSERT INTO _user (first_name, last_name, email, password, country, role) VALUES ('$firstName', '$lastName', '$email', '', '$country', '$role')";
    - This inserts a new user into the _user table. Note that the password field is left empty, which is appropriate for Google sign-up since authentication happens through Google.

- $con->query($sql);
    - This executes the insertion query.

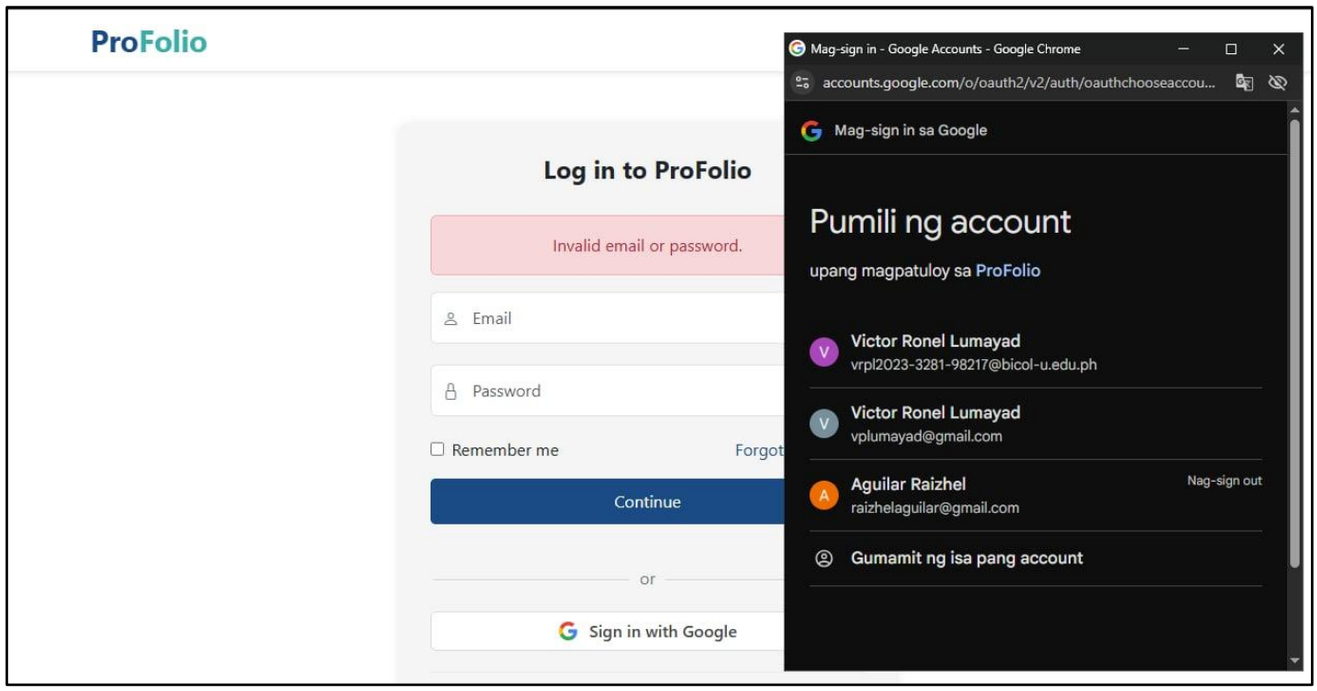
- echo json_encode(['status' => 'success', 'message' => 'Signed in with Google!']);
    - This sends a JSON response indicating successful sign-up.
- exit;
    - This terminates the script execution.

- } else { ... }
    - This block executes if a user with the given email already exists.

- echo json_encode(['status' => 'redirect', 'message' => 'Email already registered. Redirecting to login...', 'redirect_url' => 'login.php']);
    - This sends a JSON response indicating that the user is already registered and should be redirected to the login page.
- exit;
    - This terminates the script execution.

Login Page

The ProFolio login page features a clean and modern design with two main authentication options: traditional email/password login and Google Sign-In. Users can enter their credentials manually or opt to sign in quickly using their Google account. The "Sign in with Google" button triggers a Google account selection popup, making the login process faster and more convenient. This integration enhances user experience by providing a seamless and secure way to access the platform without the need to remember additional login details.



Code:

PHP > login.php

```php
<?php
session_start();
include_once("../connection/connection.php");
$con = connection();

if ($_SERVER["REQUEST_METHOD"] === "POST" && !isset($_POST['google_signup'])) {
    $email = $_POST['email'];
    $password = $_POST['password'];

    $stmt = $con->prepare("SELECT * FROM _user WHERE email = ?");
    $stmt->bind_param("s", $email);
    $stmt->execute();
    $result = $stmt->get_result();

    if ($result && $result->num_rows > 0) {
        $user = $result->fetch_assoc();

        // Verify the hashed password
        if (password_verify($password, $user['password'])) {
            $_SESSION['email'] = $user['email'];
            $_SESSION['role'] = $user['role'];

            $redirect = match ($user['role']) {
                'freelancer' => 'freelancer_dashboard.php',
                'client' => 'client_dashboard.php',
                'agency' => 'agency_dashboard.php',
                default => 'login.php'
            };
```

PHP > login.php

```php
    }
    if ($_SERVER["REQUEST_METHOD"] === "POST" && isset($_POST['google_signup'])) {
        header('Content-Type: application/json');

        $email = $_POST['email'];
        $firstName = $_POST['first_name'];
        $lastName = $_POST['last_name'];

        // Check if user exists
        $stmt = $con->prepare("SELECT * FROM _user WHERE email = ?");
        $stmt->bind_param("s", $email);
        $stmt->execute();
        $result = $stmt->get_result();

        if ($result->num_rows > 0) {
            $user = $result->fetch_assoc();
            $_SESSION['email'] = $user['email'];
            $_SESSION['role'] = $user['role'];

            // Redirect based on role
            $redirect = match ($user['role']) {
                'freelancer' => 'freelancer_dashboard.php',
                'client' => 'client_dashboard.php',
                'agency' => 'agency_dashboard.php',
                default => 'login.php'
            };
```

```php
    if ($_SERVER["REQUEST_METHOD"] === "POST" && isset($_POST['google_signup'])) {
        if ($result->num_rows > 0) {
            };

            echo json_encode([
                'status' => 'redirect',
                'message' => 'Welcome back!',
                'redirect_url' => $redirect
            ]);
            exit();
        } else {
            // New user, redirect to registration
            $_SESSION['google_email'] = $email;
            $_SESSION['google_first_name'] = $firstName;
            $_SESSION['google_last_name'] = $lastName;

            echo json_encode([
                'status' => 'redirect',
                'message' => 'Complete your registration.',
                'redirect_url' => 'register.php'
            ]);
            exit();
        }
    }

    ?>
```

```php
if ($_SERVER["REQUEST_METHOD"] === "POST" && !isset($_POST['google_signup'])) {
    if ($result && $result->num_rows > 0) {
        if (password_verify($password, $user['password'])) {
            };

            header("Location: $redirect");
            exit();
        } else {
            $_SESSION['error'] = "Invalid email or password.";
        }
    } else {
        $_SESSION['error'] = "Invalid email or password.";
    }

    header("Location: login.php");
    exit();
}
```

This login system is built using PHP and integrates both traditional email/password authentication and Google Sign-In for user flexibility and ease of access. It starts with session_start() to initialize a session, allowing user data to persist across pages during their session. The system connects to the database using a custom connection() function from an included file connection.php.

When a user submits the login form via POST, the system retrieves the entered email and password. It uses a prepared SQL statement to securely fetch the user's details from the _user table based on the email. To prevent SQL injection, the email is bound to the statement using bind_param(). After executing, it checks if the user exists, and if so, verifies the submitted password against the hashed password in the database using password_verify().

If the login is successful:

- The user's email and role are stored in the session.
- A match statement determines where to redirect the user based on their role.
- A header is used for redirection.

If the credentials are invalid:

- An error message is stored in the session ($_SESSION['error']), which is later displayed on the login page.

When the Google Sign-In button is clicked, JavaScript initializes Google's OAuth2 client using google.accounts.oauth2.initTokenClient(). It requests the user's profile and email and retrieves an access token. With this token, it fetches the user's information from Google's Userinfo API.

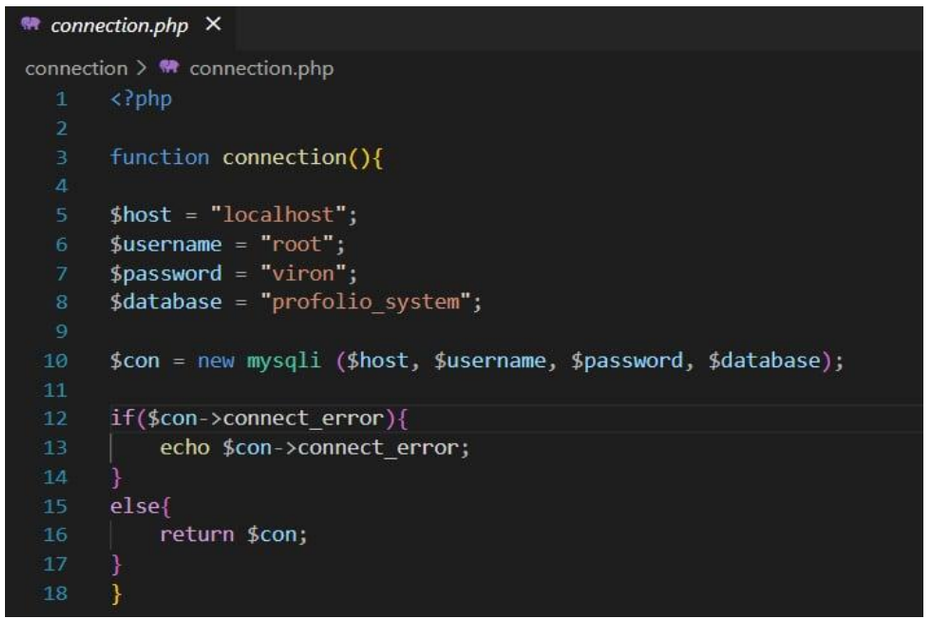
This data is then sent to the backend using an XMLHttpRequest. The PHP backend checks if the email already exists:

- If it does, it logs the user in by storing their email and role in the session and redirects them.
- If it doesn't, the user is assumed to be new. Their email and name are stored in the session temporarily, and they are redirected to a registration page to complete their account setup.

Code snippets of Server.js, Sample model, and Routes

Database Connection (connection.php)

The connection.php file is a centralized script that handles the connection to our MySQL database using PHP's mysqli extension. It defines a function that sets the database host, username, password, and database name, and then attempts to establish a connection using these details. If the connection fails, it outputs an error message using mysqli_connect_error(), which helps quickly identify issues during development. This file is included in all backend-enabled pages to ensure a consistent and reusable method of connecting to the database. Using a single connection file improves maintainability by allowing changes to be made in one place, keeps the code cleaner by avoiding repetition, and enhances organization by keeping all connection logic in one location. It also supports better security by storing sensitive database credentials in a single file, making them easier to manage and protect. Overall, connection.php plays a key role in structuring the backend of the project and helps maintain a more efficient and reliable development workflow.



- <?php
  - This line starts the PHP code block.

- function connection(){
  - This line defines a function named connection. Functions in PHP are blocks of reusable code.

- These lines define variables that hold the database connection details:
  - $host = "localhost";
    - The database server hostname (typically "localhost" for a local server).
  - $username = "root";
    - The MySQL username. Important Security Note: Using "root" in a production environment is extremely risky. Create a dedicated user with limited privileges.
  - $password = "viron";
    - The MySQL password. Important Security Note: Hardcoding passwords directly in code is a major security vulnerability. Consider using environment variables or a more secure method for storing credentials.
  - $database = "profolio_system";
    - The name of the database to connect to.

- $con = new mysqli($host, $username, $password, $database);
  - This line attempts to create a new MySQLi connection object using the credentials defined in the previous lines. mysqli is the MySQLi extension for PHP.
- This if statement checks for connection errors:
  - if ($con->connect_error) { ... }
    - This condition checks the connect_error property of the $con object. If it's not empty, it means there was an error connecting to the database.
  - echo $con->connect_error;
    - If there was an error, this line prints the error message to the output.

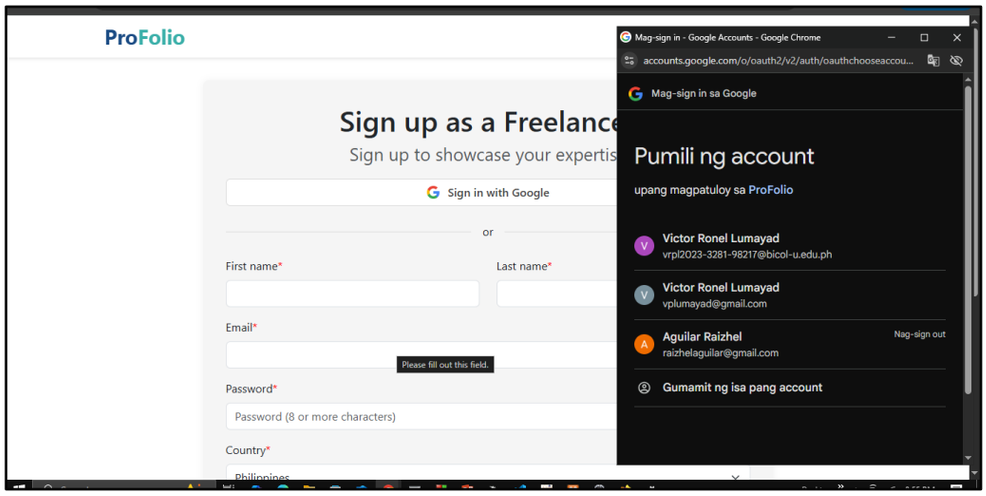
- else { return $con; }
  - If the connection was successful, the function returns the $con object, which can then be used to interact with the database.

- }
  - This closing brace ends the function definition.

API Testing Screenshots

registerFreelancer.php

The Google Sign-In integration was implemented on the freelancer registration form to enhance user experience by streamlining the sign-up process. During API testing, we verified the full OAuth 2.0 flow from initiating the sign-in via the "Sign in with Google" button to handling the authentication response on the server side.



- The process begins when a user clicks the "Sign in with Google" button on the freelancer registration form. This triggers a client-side redirect to Google's OAuth 2.0 authorization server.
- A Google authentication popup window appears, prompting the user to select their Google account and grant permission for the application to access specific user data (defined in the OAuth 2.0 scope). This ensures secure authentication without requiring the application to handle user credentials directly.
- Upon successful authentication, Google redirects the user back to a pre-configured redirect URI. This redirect includes an authorization code.
- The registerFreelancer.php script receives the authorization code. It then uses this code to exchange it for an access token via a request to Google's OAuth 2.0 token endpoint. This exchange requires the application's client ID and client secret (kept confidential).
- Using the access token, the script makes a request to Google's People API (or a similar Google API) to retrieve user information, including the user's name, email address, and potentially a profile picture URL.

- **User Registration/Login Logic:**
    - New User
        - If the retrieved email address does not exist in the application's database, a new freelancer record is created, populating the fields with the data received from Google. A new session is created for the user.

    - Existing User
        - If the email address already exists in the database, the system verifies if the user's Google account matches the existing record. If a match is found, the user is logged in directly, creating or updating their session. No password is required in this scenario.

REST API Functionality - Google Registration for Freelancer

METHOD	ENDPOINT	DESCRIPTION	REQUEST DATA	RESPONSE
POST	registerFreelancer.php	Handles the registration of a freelancer via Google Sign-In. It accepts a tokenId from the Google authentication response, verifies it with Google, and checks if the email already exists in the database. If the email is new, it registers the user with basic info extracted from the token or submitted alongside it.	**JSON:** { "tokenId": "ya29.A0AVA9y1tExampleGoogleToken" }	✅**Success**: Signed in with Google successfully. ⚠️**Email Exists:** Email already registered. Redirecting to login.

Code:

```php
// Check if the email already exists
$checkEmailSql = "SELECT * FROM _user WHERE email = '$email'";
$result = $con->query($checkEmailSql);

if ($result->num_rows > 0) {
    echo "<script>alert('Email already exists. Please use a different email.');</script>";
} else {
    // Insert the data if email doesn't exist
    $sql = "INSERT INTO _user (first_name, last_name, email, password, country, role)
            VALUES ('$firstName', '$lastName', '$email', '$password', '$country', '$role')";

    if ($con->query($sql)) {
        echo "<script>alert('Registration successful!'); window.location.href='login.php';</script>";
    } else {
        echo "Error: " . $con->error;
    }
}
```

- $checkEmailSql = "SELECT * FROM _user WHERE email = '$email'";
  - This line creates an SQL query to check if an email address already exists in the _user table. The $email variable presumably comes from user input.

- $result = $con->query($checkEmailSql);
  - This executes the query.

- if ($result->num_rows > 0) { ... }
  - This checks if the query returned any rows (meaning the email already exists). If so, a JavaScript alert is displayed.

- } else { ... }
  - This block executes if the email is unique.

- $sql = "INSERT INTO _user (first_name, last_name, email, password, country, role) VALUES ('$firstName', '$lastName', '$email', '$password', '$country', '$role')";
  - This creates an SQL query to insert new user data into the _user table. The values are taken from variables ($firstName, $lastName, etc.) presumably obtained from a user registration form.

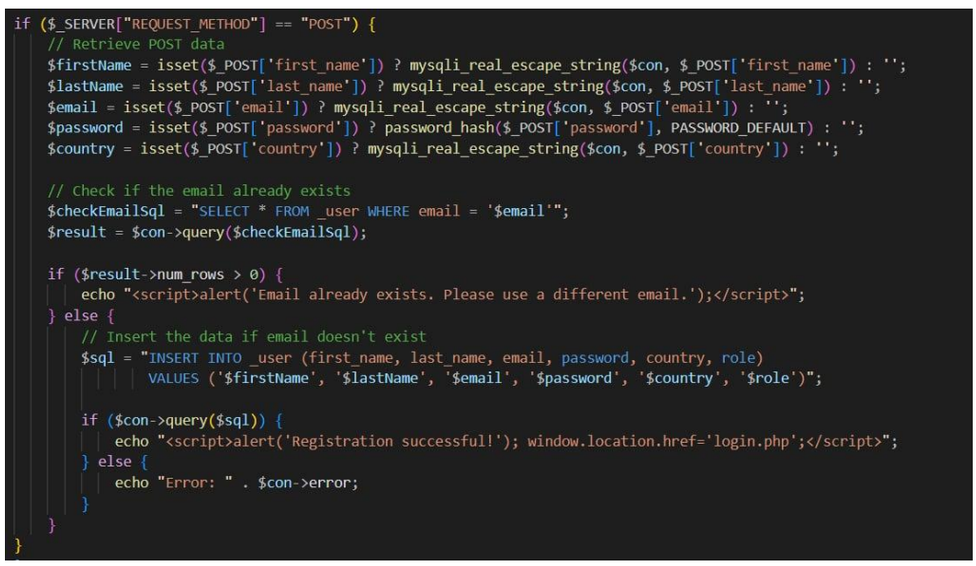
- if ($con->query($sql)) { ... }
  - This executes the insertion query. If successful, a JavaScript alert and redirect to login.php occurs.

- } else { echo "Error: " . $con->error; }
  - If the insertion fails, an error message is displayed.

REST API Functionality - Manual Registration for Freelancer

METHOD	ENDPOINT	DESCRIPTION	REQUEST DATA	RESPONSE
POST	registerFreelancer.php	Handles the manual registration of a freelancer by accepting user-submitted form data. It receives the first name, last name, email, password, country, and role. The backend checks whether the email already exists in the database. If not, it inserts the new user data into the database and returns a success response.	🟡**Manual Registration** { "firstName": "John", "lastName": "Doe", "email": "john@example.com", "password": "hashed_password", "country": "Philippines"}	✅**Success**: Registration successful ⚠️ **Email Exists:** Email already exists. Please use a different email.

Code:



```php
if ($_SERVER["REQUEST_METHOD"] == "POST") {
    // Retrieve POST data
    $firstName = isset($_POST['first_name']) ? mysqli_real_escape_string($con, $_POST['first_name']) : '';
    $lastName = isset($_POST['last_name']) ? mysqli_real_escape_string($con, $_POST['last_name']) : '';
    $email = isset($_POST['email']) ? mysqli_real_escape_string($con, $_POST['email']) : '';
    $password = isset($_POST['password']) ? password_hash($_POST['password'], PASSWORD_DEFAULT) : '';
    $country = isset($_POST['country']) ? mysqli_real_escape_string($con, $_POST['country']) : '';

    // Check if the email already exists
    $checkEmailSql = "SELECT * FROM _user WHERE email = '$email'";
    $result = $con->query($checkEmailSql);

    if ($result->num_rows > 0) {
        echo "<script>alert('Email already exists. Please use a different email.');</script>";
    } else {
        // Insert the data if email doesn't exist
        $sql = "INSERT INTO _user (first_name, last_name, email, password, country, role)
                VALUES ('$firstName', '$lastName', '$email', '$password', '$country', '$role')";

        if ($con->query($sql)) {
            echo "<script>alert('Registration successful!'); window.location.href='login.php';</script>";
        } else {
            echo "Error: " . $con->error;
        }
    }
}
```

- (if ($_SERVER["REQUEST_METHOD"] == "POST"))
    - The code begins by checking if the HTTP request method is . This ensures the code only executes when a form is submitted.

- It then retrieves user data from the $_POST superglobal array, which contains data submitted via a form. The data includes first_name, last_name, email, password, and country.

- The isset() function checks if each element exists in the $_POST array. If an element is missing, its corresponding variable is assigned an empty string. This is a basic form of input validation.

- mysqli_real_escape_string($con, ...)
    - this is applied to each field except the password. This function escapes special characters in strings to prevent some SQL injection attacks. However, this method is not sufficient to prevent all SQL injection attacks and is considered outdated and insecure compared to prepared statements. It's crucial to understand that this only provides partial protection and is not a replacement for proper parameterized queries.

- $checkEmailSql = "SELECT * FROM user WHERE email = '$email'";
    - This line constructs an SQL query to check if a user with the given email already exists in the user table. The $email variable, which comes directly from user input, is directly concatenated into the SQL string. This is the primary source of the SQL injection vulnerability.

- $result = $con->query($checkEmailSql);
  This executes the SQL query.

- if ($result->num_rows > 0) { ... }
  - This conditional statement checks the number of rows returned by the query. If the number of rows is greater than 0, it means an email with that address already exists in the database. In this case, a JavaScript alert is displayed to inform the user.

- $sql = "INSERT INTO user (first_name, last_name, email, password, country, role) VALUES ('$firstName', '$lastName', '$email', '$password', '$country', '$role')";
  - This line constructs the SQL INSERT query to add the new user to the database. The values for the fields are taken directly from the user input variables. This is another point where SQL injection is possible because user-supplied data is directly inserted into the query.

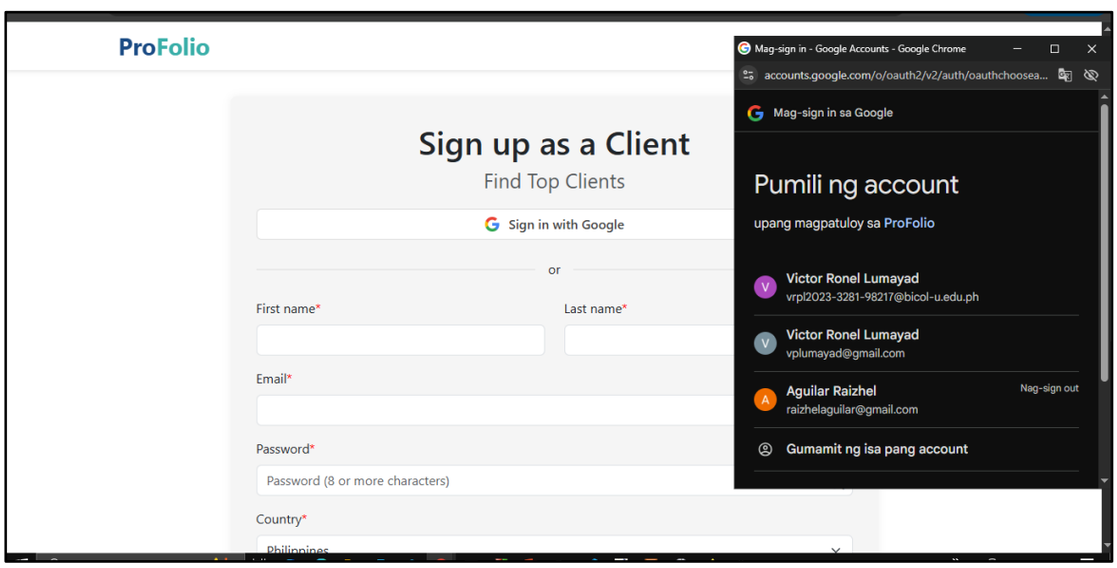
- password_hash($POST['password'], PASSWORD_DEFAULT)
  - This line uses the password_hash() function to securely hash the user's password before storing it in the database. This is a crucial security step that protects against plain-text password storage.

- The INSERT query is then executed using $con->query($sql).

- The conditional statement checks for successful execution: If the query is successful, a JavaScript alert is displayed, and the user is redirected to the login page (login.php). If there's an error, the database error message is displayed.

registerClient.php

The Google Sign-In feature is integrated into the Client Registration Form to simplify and speed up the account creation process, while also enhancing security. By using this feature, clients can bypass manual data entry for fields like name and email. When a client clicks the “Sign in with Google” button at the top of the form, it triggers Google’s OAuth 2.0 login flow. A popup window appears, prompting the user to choose a Google account, “Pumili ng account upang magpatuloy sa ProFolio”. Once the user selects an account, Google authenticates their identity and requests permission to share basic profile information such as name and email. After successful authentication, this data is securely sent to the backend script (registerClient.php). From there, the system can auto-fill the registration form, register the user instantly, or log them in automatically if their account already exists. The integration ensures a faster, smoother user experience with minimal friction during sign-up.



REST API Functionality - Google Sign-In for Client

METHOD	ENDPOINT	DESCRIPTION	REQUEST DATA	RESPONSE
POST	registerClient.php	Handles the registration of a client using Google Sign-In. The frontend sends a tokenId from Google, which is then verified by the backend to authenticate the user. Upon successful verification, Google provides basic profile information such as first name, last name, and email. The backend checks whether the email is already registered. If it exists, the user is redirected to log in.	**JSON:** {"tokenId": "ya29.A0AVA9y1tExampleGoogleToken"}	✅ **Success**: Signed in with Google successfully. ⚠️ **Email Exists:** Email is already registered. Redirecting to login.

Code:

```php
// Retrieve the role from the URL parameter
$role = isset($_GET['role']) ? $_GET['role'] : 'client'; // Default to 'client' if not provided

// Handle Google Sign-Up

if ($_SERVER["REQUEST_METHOD"] == "POST" && isset($_POST['google_signup'])) {
  $firstName = isset($_POST['first_name']) ? mysqli_real_escape_string($con, $_POST['first_name']) : '';
  $lastName = isset($_POST['last_name']) ? mysqli_real_escape_string($con, $_POST['last_name']) : '';
  $email = isset($_POST['email']) ? mysqli_real_escape_string($con, $_POST['email']) : '';
  $role = 'client';

  if (!empty($email)) {
    $checkUserSql = "SELECT * FROM _user WHERE email = '$email'";
    $result = $con->query($checkUserSql);

    if ($result->num_rows === 0) {
      // No user yet - proceed with Google sign-up
      $sql = "INSERT INTO _user (first_name, last_name, email, password, country, role)
              VALUES ('$firstName', '$lastName', '$email', '', '', '$role')";
      $con->query($sql);
      echo json_encode(['status' => 'success', 'message' => 'Signed in with Google!']);
      exit;
    } else {
      // Email already exists - do NOT allow Google sign-up again
      echo json_encode([
        'status' => 'redirect',
        'message' => 'Email already registered. Redirecting to login...',
        'redirect_url' => 'login.php'
      ]);

      exit;
    }
```

- <?php
  - This line starts the PHP code block.

- if ($_SERVER["REQUEST_METHOD"] == "POST" && isset($_POST['google_signup'])) {
  This line checks two conditions:
  - $_SERVER["REQUEST_METHOD"] == "POST"
    - Verifies that the HTTP request method is POST (meaning data was submitted from a form).
  - isset($_POST['google_signup'])
    - Checks if the google_signup key exists in the $_POST array. This likely indicates that the user initiated Google Sign-In. The entire block within the if statement only executes if both conditions are true.
- These lines 4-8 retrieve values from the $_POST array, which contains data submitted from the Google Sign-In form. isset() checks if each key exists; if not, an empty string is used as a default. No sanitization is needed here because prepared statements handle data security.

- $role = isset($_POST['role']) ? $_POST['role'] : 'client';
  - This line retrieves the user's role from the $_POST array. If the role key is not set, it defaults to 'client'.

- $stmt = $con->prepare("SELECT 1 FROM _user WHERE email = ?");
  - This line prepares an SQL statement to check for an existing email address. $con is assumed to be a database connection object established elsewhere. SELECT 1 is an optimized query; it only retrieves a single column (1) if a match is found, making it more efficient than SELECT *. The ? is a placeholder for the email address.

- $stmt->bind_param("s", $email);
  - This line binds the $email variable to the placeholder in the prepared statement. "s" specifies that the variable is a string. This is crucial for security; it prevents SQL injection.

- $stmt->execute();
  - This line executes the prepared statement.

- $result = $stmt->get_result();
  - This line gets the result set from the executed query.

- if ($result->num_rows > 0) { ... }
  - This line checks if the query returned any rows (meaning the email already exists).

- echo json_encode(['status' => 'redirect', ...]);
  - If the email already exists, this line sends a JSON response indicating a redirect to the login page.

- } else { ... }
  - This block executes if the email is unique (not found in the database).

- $stmt = $con->prepare("INSERT INTO _user ... VALUES (?, ?, ?, ?, ?, ?)");
  - This line prepares an SQL INSERT statement to add a new user to the database. The ? placeholders represent the values for the respective columns.

- $stmt->bind_param("ssssss", $firstName, $lastName, $email, "", $country, $role);
  - This line binds the user data to the placeholders in the prepared statement. "ssssss" specifies that all variables are strings. The password is set to an empty string ("") because Google handles authentication; the application doesn't need to store a password in this scenario.

- if ($stmt->execute()) { ... }
  - This line executes the prepared INSERT statement.

- echo json_encode(['status' => 'success', ...]);
  - If the insertion was successful, this line sends a JSON response indicating success.

- } else { echo json_encode(['status' => 'error', 'message' => $stmt->error]); }
  - If the insertion failed, this line sends a JSON response with an error message from the database.

- $stmt->close();
  - This line closes the prepared statement to release database resources.

- $stmt->close();
  - This line closes the prepared statement used for the email check.

- }
  - This closing brace ends the if statement.

- ?>
  - This line closes the PHP code block.

REST API Functionality - Manual Registration for Client

METHOD	ENDPOINT	DESCRIPTION	REQUEST DATA	RESPONSE
POST	registerClient.php	Handles the manual registration of a client. This endpoint receives form data directly from the user. It checks if the provided email already exists in the database. If the email is unique, the client is registered with the provided information.	🟡 **Manual Registration** { "firstName": "John", "lastName": "Doe", "email": "john@example.com", "password": "hashed_password", "country": "Philippines"}	✅ **Success**: Registration successful ⚠️ **Email Exists:** Email is already exists. Please use a different email.

Code:

```php
if ($_SERVER["REQUEST_METHOD"] == "POST") {
    // Retrieve POST data
    $firstName = isset($_POST['first_name']) ? mysqli_real_escape_string($con, $_POST['first_name']) : '';
    $lastName = isset($_POST['last_name']) ? mysqli_real_escape_string($con, $_POST['last_name']) : '';
    $email = isset($_POST['email']) ? mysqli_real_escape_string($con, $_POST['email']) : '';
    $password = isset($_POST['password']) ? password_hash($_POST['password'], PASSWORD_DEFAULT) : '';
    $country = isset($_POST['country']) ? mysqli_real_escape_string($con, $_POST['country']) : '';

    // Check if the email already exists
    $checkEmailSql = "SELECT * FROM _user WHERE email = '$email'";
    $result = $con->query($checkEmailSql);

    if ($result->num_rows > 0) {
        echo "<script>alert('Email already exists. Please use a different email.');</script>";
    } else {
        // Insert the data if email doesn't exist
        $sql = "INSERT INTO _user (first_name, last_name, email, password, country, role)
                VALUES ('$firstName', '$lastName', '$email', '$password', '$country', '$role')";

        if ($con->query($sql)) {
            echo "<script>alert('Registration successful!'); window.location.href='login.php';</script>";
        } else {
            echo "Error: " . $con->error;
        }
    }
}
?>
```

- if ($_SERVER["REQUEST_METHOD"] == "POST") { ... }
  - This condition ensures the code executes only if the request is a POST request. POST requests are typically used to submit data from a form to a server.

- $firstName = isset($_POST['first_name']) ? mysqli_real_escape_string($con, $_POST['first_name']) : '';
  - This line retrieves the first_name from the POST data. isset() checks if the variable is set. mysqli_real_escape_string($con, $_POST['first_name']) sanitizes the input to prevent SQL injection vulnerabilities. If first_name is not set, it defaults to an empty string. Similar lines handle lastName, email, password, and country.

- $password = isset($_POST['password']) ? password_hash($_POST['password'], PASSWORD_DEFAULT) : '';
  - Crucially, the password is not stored in plain text. password_hash() securely hashes the password using the PASSWORD_DEFAULT algorithm (recommended for security).

- $checkEmailsql = "SELECT * FROM _user WHERE email = '$email'";
  - This SQL query checks if an email address already exists in the _user table.

- $result = $con->query($checkEmailSql);
  - The query is executed, and the result is stored in $result.

- if ($result->num_rows > 0) { ... }
  - If the query returns one or more rows (meaning the email exists), an alert message is displayed informing the user.

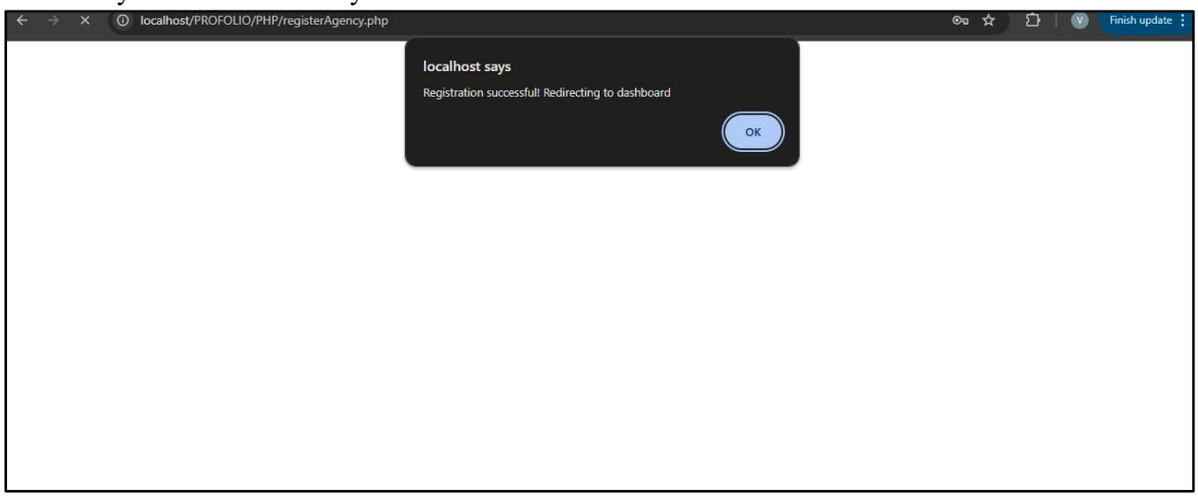
- $sql = "INSERT INTO _user (first_name, last_name, email, password, country, role) VALUES ('$firstName', '$lastName', '$email', '$password', '$country', '$role')";
    - If the email is unique, this SQL query inserts the user's data into the _user table. Note that $role is used, suggesting different user roles might exist in the system. The values are inserted using placeholders, which are vulnerable to SQL injection if not properly sanitized (as done earlier).

- if ($con->query($sql)) { ... }
    - If the insertion is successful, a success message is displayed, and the user is redirected to login.php.

- else { echo "Error: " . $con->error; }
    - If an error occurs during insertion, an error message is displayed.

registerAgency.php

Manual registration requires agencies to fill out a form with details such as agency name, logo, type, team size, description, services, website (optional), and admin account information (name, job title, email, password, country, city). The server-side code then validates, sanitizes, and securely hashes the password before storing the data in the database. Google Sign-In streamlines the process by allowing agencies to click the "Sign in with Google" button, authenticate with their Google account, and grant permission to share their name and email. The server retrieves this data, automatically populates the form fields, and either creates a new agency account or logs the user in directly if the email already exists.



REST API Functionality - Manual Registration for Agency

METHOD	ENDPOINT	DESCRIPTION	REQUEST DATA	RESPONSE
POST	registerAgency.php	Handles the manual registration of an agency. This endpoint receives form data directly from the user. It checks if the provided email already exists in the database. If the email is unique, the client is registered with the provided information; otherwise, an appropriate error message is returned. The endpoint also validates the input data to ensure data integrity and prevent vulnerabilities. Upon successful registration, a confirmation message or response is sent to the client.	**Manual Registration:** {"agencyName": "Example Agency","agencyLogo": "", // Base64 encoded logo (optional)"agencyType": "Design Agency","teamSize": "2-5","agencyDescription": "A creative design agency specializing in branding and web development.","services": "Web Design, Web Development, UI/UX Design", // Comma-separated list"agencyWebsite": "https://exampleagency.com","firstName": "John","lastName": "Doe","jobTitle": "Creative Director","email": "john.doe@exampleagency.com","password": "securepassword","country": "Philippines","city": "Manila")	✅ **Success**: Registration successful! Redirecting to dashboard' ⚠ **Email Exists:** The email address is already registered!

Code:

```php
PHP > registerAgency.php
  8   if ($_SERVER['REQUEST_METHOD'] == 'POST') {
 13       if (isset($_FILES['agencyLogo']) && $_FILES['agencyLogo']['error'] == 0) {
 40           }
 41       } else {
 42           $agencyLogo = ''; // If no file uploaded
 43       }
 44
 45       $agencyType = $_POST['agencyType'];
 46       $teamSize = $_POST['teamSize'];
 47       $agencyDescription = $_POST['agencyDescription'];
 48
 49       // Handle services selection
 50       $services = isset($_POST['services']) && is_array($_POST['services']) ? implode(',', $_POST['services'
 51
 52       $agencyWebsite = $_POST['agencyWebsite'];
 53       $firstName = $_POST['firstName'];
 54       $lastName = $_POST['lastName'];
 55       $jobTitle = $_POST['jobTitle'];
 56       $email = $_POST['email'];
 57       $password = password_hash($_POST['password'], PASSWORD_DEFAULT); // Hash password
 58       $country = $_POST['country'];
 59       $city = $_POST['city'];
 60
 61       // Check if the email already exists
 62       $emailCheck = $con->prepare("SELECT email FROM _user WHERE email = ?");
 63       $emailCheck->bind_param("s", $email);
 64       $emailCheck->execute();
 65       $emailCheck->store_result();
 66
 67       if ($emailCheck->num_rows > 0) {
```

```php
<?php
include_once("../connection/connection.php");
$con = connection();

$role = isset($_GET['role']) ? $_GET['role'] : 'agency'; // Default to 'Agency' if not provided

if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    // Get form data
    $agencyName = $_POST['agencyName'];

    // Handle file upload for logo
    if (isset($_FILES['agencyLogo']) && $_FILES['agencyLogo']['error'] == 0) {
        $logoTmp = $_FILES['agencyLogo']['tmp_name'];
        $logoName = $_FILES['agencyLogo']['name'];
        $logoExtension = strtolower(pathinfo($logoName, PATHINFO_EXTENSION));

        // Allowed file types
        $allowedExtensions = ['jpg', 'jpeg', 'png', 'gif'];

        // Check if the file is of a valid type
        if (in_array($logoExtension, $allowedExtensions)) {
            $logoDestination = '../uploads/' . $logoName;

            // Check if file already exists
            if (file_exists($logoDestination)) {
                echo "Error: File already exists.";
                exit();
            } else {
                if (move_uploaded_file($logoTmp, $logoDestination)) {
```

```php
 8   if ($_SERVER['REQUEST_METHOD'] == 'POST') {
67       if ($emailCheck->num_rows > 0) {
72       }
73       $emailCheck->close();
74
75       // Insert into database (without the 'id' field as it's auto-increment)
76       $stmt = $con->prepare("INSERT INTO _user (first_name, last_name, email, password, country, city, agenc
77       VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?)");
78
79       $stmt->bind_param("sssssssssssss", $firstName, $lastName, $email, $password, $country, $city, $agenc
80
81       if ($stmt->execute()) {
82           echo "<script>alert('Registration successful! Redirecting to dashboard'); window.location.href='lo
83       } else {
84           echo "Error: " . $stmt->error;
85       }
86       $stmt->close();
87       $con->close();
88   }
89   ?>
90
```