# ADMIN, TEACHER, AND STUDENT DASHBOARDS

**Learning Objectives**

By the end of this laboratory exercise, students should be able to:

* Differentiate user roles and implement role-based access control (RBAC).
* Create distinct, role-specific dashboards within a single application.
* Develop dynamic navigation bars that change based on user role.
* Utilize CodeIgniter's Session library to manage user state and permissions across pages.
* Apply Bootstrap components and layout techniques to create informative and user-friendly dashboard interfaces.
* Implement authorization checks to restrict access to specific functionalities.

**Prerequisite student experiences and knowledge**

Before starting this exercise, students should have:

* Completed Laboratory Exercise 4 (User Authentication).
* A functioning login/registration system with a `users` table containing a `role` field.
* Understanding of CodeIgniter controllers, views, and session management.
* Basic proficiency in HTML, PHP, and Bootstrap grid system & components.
* Ability to write simple SQL queries and use the CodeIgniter Model.

**Background**

Most real-world applications serve different types of users, each with unique privileges and needs. A Learning Management System (LMS) is a prime example, typically involving Administrators (manage system, users, courses), Teachers (create content, manage grades), and Students (view courses, submit work).

This exercise focuses on building upon the authentication system from Lab 4. After a user logs in, they must be redirected to a dashboard tailored to their role. The application must also protect these dashboards, ensuring users cannot access areas reserved for other roles, a concept known as Role-Based Access Control (RBAC).

**Materials/Resources**

* Personal Computer with Internet Access
* XAMPP/WAMP/LAMP server installed
* CodeIgniter Framework (latest version)
* Visual Studio Code or any code editor
* Git and GitHub Account
* Web Browser (Chrome, Firefox, etc.)
* Pass the user's role and relevant data to the view.

**Step 4: Create a Unified Dashboard View with Conditional Content**

1. Create or modify the dashboard view at **app/Views/auth/dashboard.php**.
2. Use PHP conditional statements to display different content based on the user's role.

**Step 5: Create a Dynamic Navigation Bar**

1. Modify your header template (**app/Views/templates/header.php**) to include role-specific navigation items accessible from anywhere in the application.

**Step 6: Configure RoutesLaboratory Activity**

**Step 1: Project Setup**

1. Open your existing ITE311-LASTNAME CodeIgniter project.
2. Ensure your database has a **users** table with a **role** column: **admin, teacher, student**.
   * If not, create a new migration to alter the table.
3. Verify that the login process from Lab 4 correctly stores the user's **role** in the session data.
4. Open your previously created CodeIgniter project **ITE311-LASTNAME**.
5. Ensure your local server and database are running.
6. Open a terminal/command prompt in your project root.

**Step 2: Modify the Login Process for Unified Dashboard**

1. Navigate to your **Auth.php** controller.
2. Locate the **login()** method where user credentials are verified.
3. After a successful login, redirect everyone to a generic **dashboard** and implement a conditional check on the user's **role** from the session.

**Step 3: Enhance the Dashboard Method in the Auth Controller**

1. In your **Auth.php** controller, locate the **dashboard()** method.
2. Enhance this method to:
3. Perform authorization check (ensure user is logged in).
4. Fetch role-specific data from the database.
5. Ensure your **app/Config/Routes.php** has the correct route for the dashboard:
   * $routes->get('/dashboard', 'Auth::dashboard');

**Step 7: Test the Application Thoroughly**

1. Register or manually create users in your database with different roles (**admin, teacher, student**).
2. Log in with each user and verify:

* All users are redirected to the same **dashboard** URL.
* The dashboard displays different content based on the user's role.
* The navigation bar shows appropriate menu items for each role.
* Users can only see and access functionality intended for their role.

1. Test the logout functionality and access control.

**Step 8: Push to GitHub**

1. Commit your changes with a descriptive message.
   * At least five commits and it should be 4 days before submission are required to identify the progress of version control of the code or syntax.
   * Commit: "ROLE BASE Implementation"
2. Push the changes to your GitHub repository.

**Step 9: Vulnerable Checking**

1. Secure the **students** login and registration process so there is no vulnerability in the login procedures.

Output / Results

* Screenshot 1: The **user's table shows** users with different roles.
* Screenshot 2: When logged in as an admin, the dashboard view shows admin-specific content.
* Screenshot 3: When logged in as a teacher, the dashboard view shows teacher-specific content.
* Screenshot 4: When logged in as a student, the dashboard view shows student-specific content.
* Screenshot 5: The navigation bar displays different menu items for admin vs student users.
* Screenshot 6: The GitHub repository shows the latest commits.

**QUESTIONS:**

* + - 1. **Authorization vs. Authentication: Based on your implementation, explain the difference between authentication from Lab 4 and authorization from Lab 5. Where in your code did you implement authorization?**

In my system, authentication (Lab 4) is about checking who the user is. It happens in Auth::login() where the email and password are checked, and in Auth::setAuthSession() where the user info is saved into the session.

Authorization (Lab 5) is about what the user can see after login. This is done in Auth::dashboard() where it checks if session('userID') exists before showing the dashboard, and also in dashboard.php where the role decides what part of the dashboard will show.

1. **How does the dashboard view determine which content to display? Explain the role of the session variable in this process.**

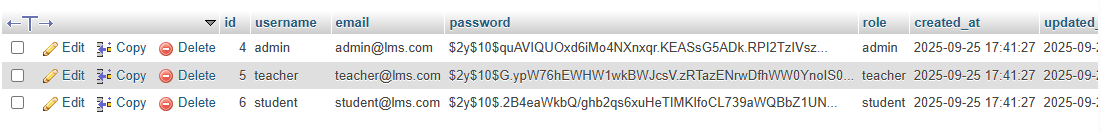
The dashboard uses the role from the session. When the user logs in, their role (like admin, teacher, or student) is stored in the session. In dashboard.php, there is a switch that checks that role and shows the right view (admin view, teacher view, or student view). That’s how it decides what content to display.

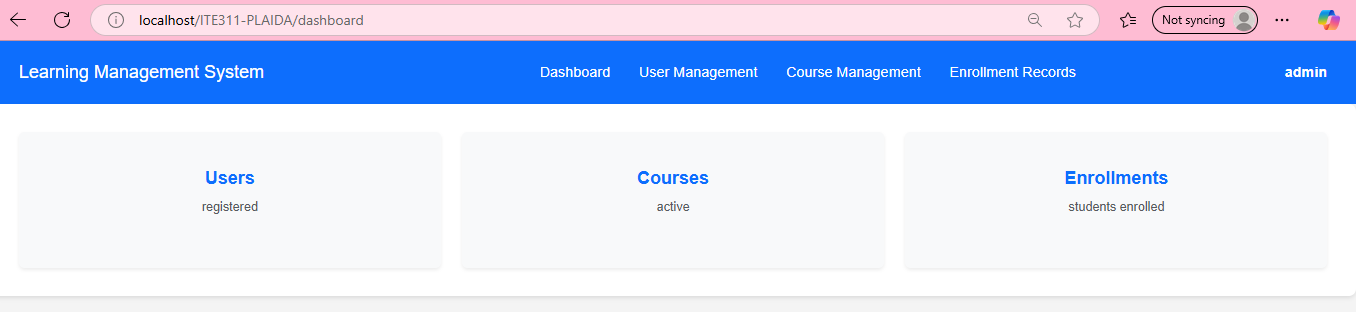
**3. If we wanted to add a new user role, what changes would be required in the current implementation to support this new role?**

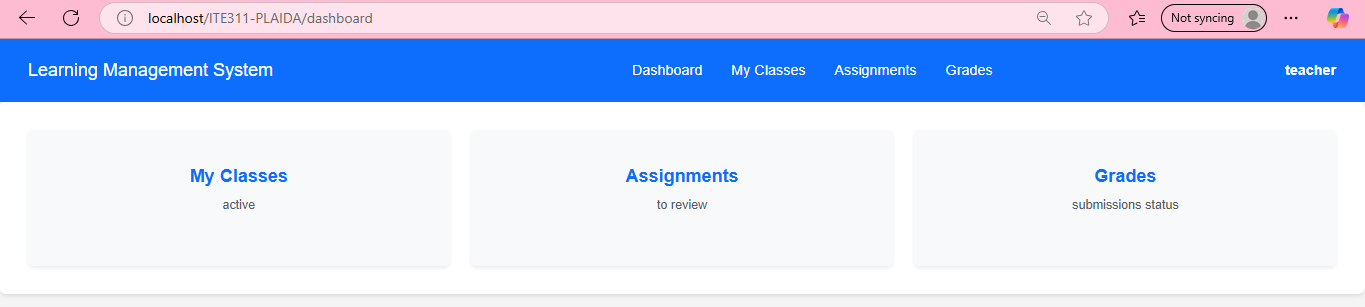
If I want to add another role, I need to:

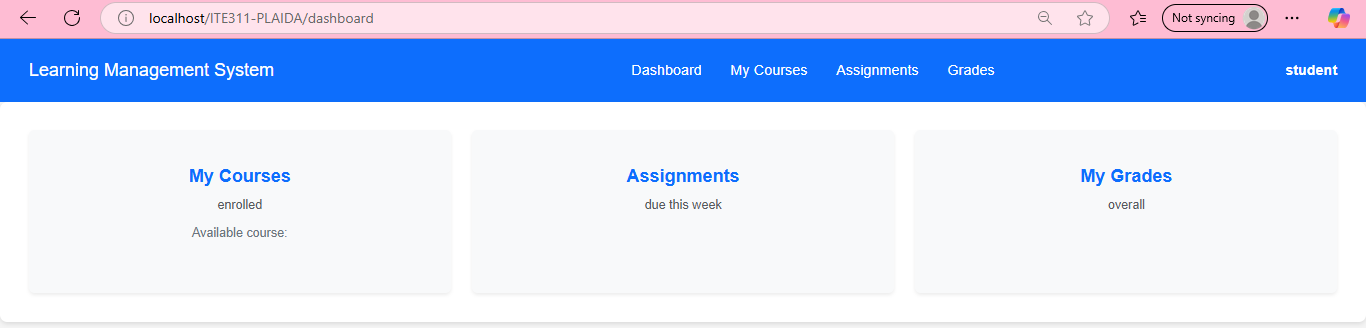
* Update the validation in Auth::getRegisterRules() to include the new role.
* Add the new role option in the registration form.
* Add a new case in the switch inside dashboard.php for the new role.
* Create a new view file for that role.

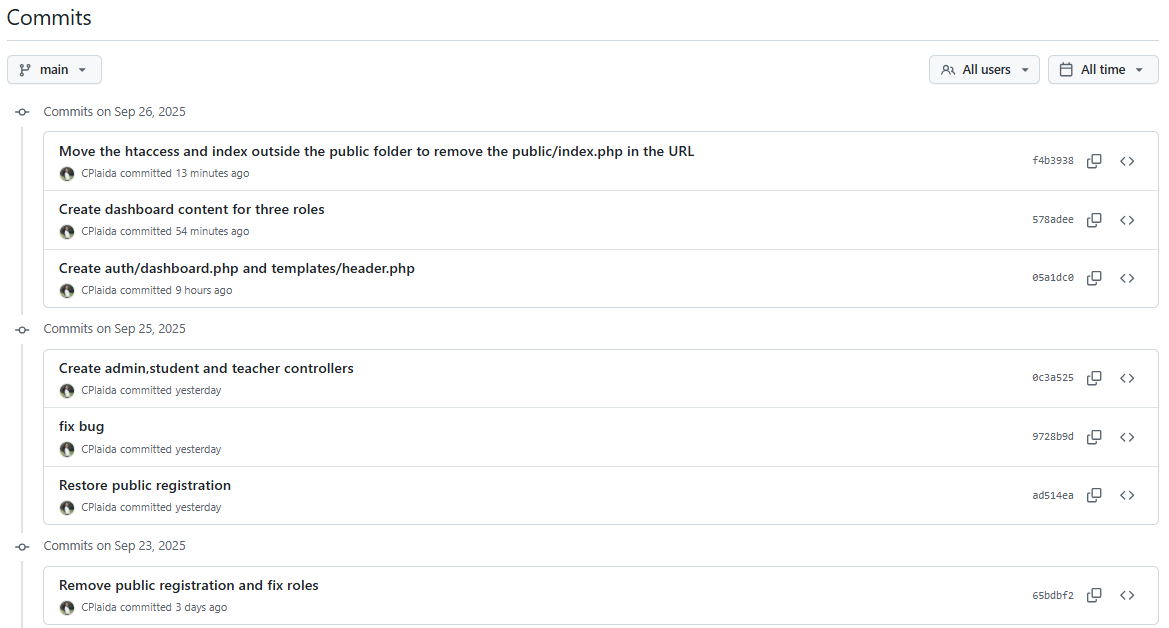
I don’t need to change the routes, since all roles already go to /dashboard. The reason I can use the same route is because the role saved in the session decides what content to show. Instead of creating different routes like /adminDashboard, /teacherDashboard, or /studentDashboard, my system uses only one /dashboard route and then checks the role of the logged-in user to show the correct dashboard view. This makes the code simpler, avoids repeating routes, and keeps all users under one unified entry point.

**Output / Results**









**Conclusion**

I learned that authentication is about checking the user’s identity, while authorization is about what the user can open after login. The dashboard looks at the session role to show the right content for admin, teacher, or student. If a new role is added, the code and views just need updates so the system can support it. This shows that roles are important because they control what each user can see and do.