



TECHNICAL DOCUMENTATION

DormDash



22 OCTOBER 2025

CMPG 323

Contents

Introduction	3
Purpose	3
Scope	3
System Setup.....	4
Docker Setup	8
Dockerfile Overview	8
Building and Running the Container	8
System Architecture	9
Overview.....	9
Presentation Layer (Frontend)	9
Application Layer (Backend)	16
Data Layer (Database)	22
Testing and Quality Assurance	22
Configuration and Tooling	26
Integration and Data Flow	30
Version Control	31
Overview.....	31
Repository Structure	32
Branching Strategy	34
Commit Guidelines	36
Collaboration Practices	38
Frontend Structures	39
Frontend Directory Structure:.....	39
Routing (React Router DOM):.....	40
API Communication:	41
File :api.php	42
UI/UX Design:.....	46
Backend Structure	50
Architectural Overview	50
Route Layer	50
Controller Layer	51
Model Layer	53
Middleware and Security	54
Testing and Quality Assurance	54
Testing Overview	54
Developer Testing.....	54

User Testing.....	55
Quality Assurance Process	56
Security Measures	57
Appendices	58
Appendix A: Environment Variables Reference	58
Appendix B: API Endpoint Catalogue	58
Appendix C: Typical Test Cases and Quality Assurance	60
Appendix D: Glossary of Terms	60
Appendix E: Directory Structure Snapshot	61

Introduction

Purpose

The purpose of Dorm Dash, the Resident Management Platform (RMP), is to streamline and digitize the management of student residences at the North-West University (NWU) across all campuses. The platform provides a centralized, secure, and role-based system that consolidates key residential management processes such as room and bed allocation, event management, communication, and reporting into one accessible web-based interface.

By transitioning from manual or fragmented processes to a unified digital solution, Dorm Dash aims to enhance efficiency, transparency, and accountability in residence administration. In addition to meeting NWU's operational requirements, the platform is designed with scalability and modularity in mind—allowing for future commercialization and potential adaptation by other universities, colleges, or residential institutions seeking similar digital transformation solutions.

Scope

Dorm Dash is being developed as a Minimum Viable Product (MVP) that fulfils NWU's immediate needs while laying the foundation for future expansion. The MVP will incorporate core functionality centred around resident safety, data security, and administrative efficiency, including:

- User Authentication and Role Management: Secure login and access control compliant with POPIA data protection standards.
- Room and Bed Allocation: Automated assignment and management of student accommodations.
- Event Management: Creation and coordination of residence and campus-related events.
- Voting System: Digital voting for residence leadership and other student affairs.
- Group Management and Chat Functionality: In-platform communication channels for residents and administrators.
- Notification System: Real-time alerts and announcements to keep users informed.
- Reporting and Analytics: Tools for administrators to generate insights into occupancy, events, and other key metrics.

The system architecture is designed to be scalable and modular, supporting integration with future components such as the NWU Student Information System (SIS) and advanced analytics dashboards. This approach ensures that Dorm Dash remains adaptable for long-term institutional needs and potential commercial deployment.

System Setup

Pre-requisites

Before deploying or running Dorm Dash, the following tools and dependencies must be installed and configured:

Category	Requirement	Description
Operating System	Ubuntu 20.04 or later / Windows 11 / macOS Monterey or later	Docker-compatible environment
Containerization	Docker Desktop $\geq$ 4.29	Provides Docker Engine and Docker Compose for container management
Package Managers	Composer $\geq$ 2.7, Node.js v22 + npm 10 + Yarn $\geq$ 1.22	Required for installing Laravel / React dependencies
Source Control	Git $\geq$ 2.40	Version-control system for cloning and managing source code
Browser	Chrome, Edge, or Firefox (latest)	For accessing the web interface
Database	MySQL 8 (local or remote)	Used by Laravel for persistence; configured in .env
Optional Utilities	Postman, MySQL Workbench	For API testing and database management

Environment Variables

Copy .env.example to .env and configure credentials for DB_DATABASE, DB_USERNAME, DB_PASSWORD, and APP_KEY before container build.

Run php artisan key:generate to register the application key.

Alternatively, find the .env file's text below

APP_NAME=DomrDash

APP_ENV=local

APP_KEY=base64:L1fdV15PzDFL89cHe4+JXMrZjdYVfAHfVX9Qz/4AF5M=

APP_DEBUG=true

APP_URL=http://localhost:8000

APP_LOCALE=en

APP_FALLBACK_LOCALE=en

APP_FAKER_LOCALE=en_US

APP_MAINTENANCE_DRIVER=file

APP_MAINTENANCE_STORE=database

PHP_CLI_SERVER_WORKERS=4

BCRYPT_ROUNDS=12

LOG_CHANNEL=stack

LOG_STACK=single

LOG_DEPRECATIONS_CHANNEL=null

LOG_LEVEL=debug

DB_CONNECTION=mysql

DB_HOST=mysql-8ce78f3-dormdash-web.d.aivencloud.com

DB_PORT=12246

DB_DATABASE=defaultdb

DB_USERNAME=avnadmin

DB_PASSWORD=AVNS_wdogBEGEMzmQ0-uA4BO

MYSQL_ATTR_SSL_CA=null

SESSION_DRIVER=database

SESSION_LIFETIME=120

SESSION_ENCRYPT=false

SESSION_PATH=

SESSION_DOMAIN=null

BROADCAST_CONNECTION=log

FILESYSTEM_DISK=local

QUEUE_CONNECTION=database

CACHE_STORE=database

CACHE_PREFIX=

MEMCACHED_HOST=127.0.0.1

REDIS_CLIENT=phpredis

REDIS_HOST=127.0.0.1

REDIS_PASSWORD=null

REDIS_PORT=6379

MAIL_MAILER=log

MAIL_SCHEME=null

MAIL_HOST=127.0.0.1

MAIL_PORT=2525

MAIL_USERNAME=null

MAIL_PASSWORD=null

MAIL_FROM_ADDRESS="hello@example.com"

MAIL_FROM_NAME="\${APP_NAME}"

AWS_ACCESS_KEY_ID=

AWS_SECRET_ACCESS_KEY=

AWS_DEFAULT_REGION=us-east-1

AWS_BUCKET=

AWS_USE_PATH_STYLE_ENDPOINT=false

VITE_APP_NAME="\${APP_NAME}"

BROADCAST_DRIVER=pusher

PUSHER_APP_ID=your_app_id

PUSHER_APP_KEY=your_app_key

PUSHER_APP_SECRET=your_app_secret

PUSHER_HOST=api.pusherapp.com

PUSHER_PORT=443

PUSHER_SCHEME=https

PUSHER_APP_CLUSTER=mt1

BROADCAST_DRIVER=pusher

PUSHER_APP_ID=2063787

PUSHER_APP_KEY=9ac25f1a718056eb2a5c

PUSHER_APP_SECRET=9333dc6c5ef238e4366e

PUSHER_APP_CLUSTER=mt1

VITE_PUSHER_APP_KEY="\${PUSHER_APP_KEY}"

VITE_PUSHER_HOST=api-\${PUSHER_APP_CLUSTER}.pusher.com

VITE_PUSHER_PORT=443

VITE_PUSHER_SCHEME=https

VITE_PUSHER_APP_CLUSTER="\${PUSHER_APP_CLUSTER}"

Docker Setup

Dorm Dash is containerized using a customized Ubuntu-based Laravel Sail image that bundles PHP 8.0, Node 22, MySQL/PostgreSQL clients, and supporting utilities.

Docker allows the project to run in a reproducible, isolated environment—ensuring identical behaviour across developer machines.

Dockerfile Overview

The provided Dockerfile:

- Uses Ubuntu 24.04 as the base image.
- Installs PHP 8.0 and extensions (MySQL, GD, XML, Redis, etc.).
- Adds Node 22, Yarn, and Composer for React + Laravel builds.
- Configures a non-root 'sail' user and exposes port 80.
- Launches the Laravel development server via Supervisor.

Building and Running the Container

1. Clone the repository

[DSVerster/dormdash-web](https://github.com/DSVerster/dormdash-web)

2. Copy and configure environment variables (See code above) In your IDE run the following command in the terminal

```
composer install  
npm install  
php artisan migrate:fresh  
php artisan db:seed  
npm run dev  
php artisan serve --port=8000
```

3. Access the application.
4. Navigate to `http://localhost:8000/` in your browser.

The Laravel + React SPA should load with Inertia-driven routing.

System Architecture

Overview

The Residence Management Platform (RMP), named *DormDash-Web*, was developed using a three-tier web architecture consisting of a presentation layer (frontend), an application layer (backend), and a data layer (database). This layered structure ensures scalability, maintainability, and clear separation of concerns, making it easier to extend the platform beyond the semester's MVP phase. The system runs within Docker containers, allowing developers from different campuses to maintain consistent environments during collaborative work. This architecture combines React.js, Laravel, and MySQL, forming a secure, fast, and modern web application framework suited for academic and production environments.

Presentation Layer (Frontend)

The frontend is built with React.js (version 18.3.1), integrated into Laravel using Inertia.js. This enables a smooth single-page application (SPA) experience where pages are updated dynamically without reloading the entire site.

The interface is mobile-responsive and designed for two main user roles: Students and Administrators.

Styling and interactivity are achieved using:

- TailwindCSS and Bootstrap for responsive design
- Radix UI and Headless UI for accessible components
- Framer Motion for animations
- Lucide-React for icons
- Axios for HTTP communication
- Laravel Echo + Pusher.js for real-time notifications

This layer offers dashboards for each role, allowing students to view rooms, events, and notifications while administrators manage allocations, events, and announcements.

- **Figures below illustrate the frontend structure and integration setup**

Figure 1: Public assets and layout components (auth, app, settings) in the React frontend.

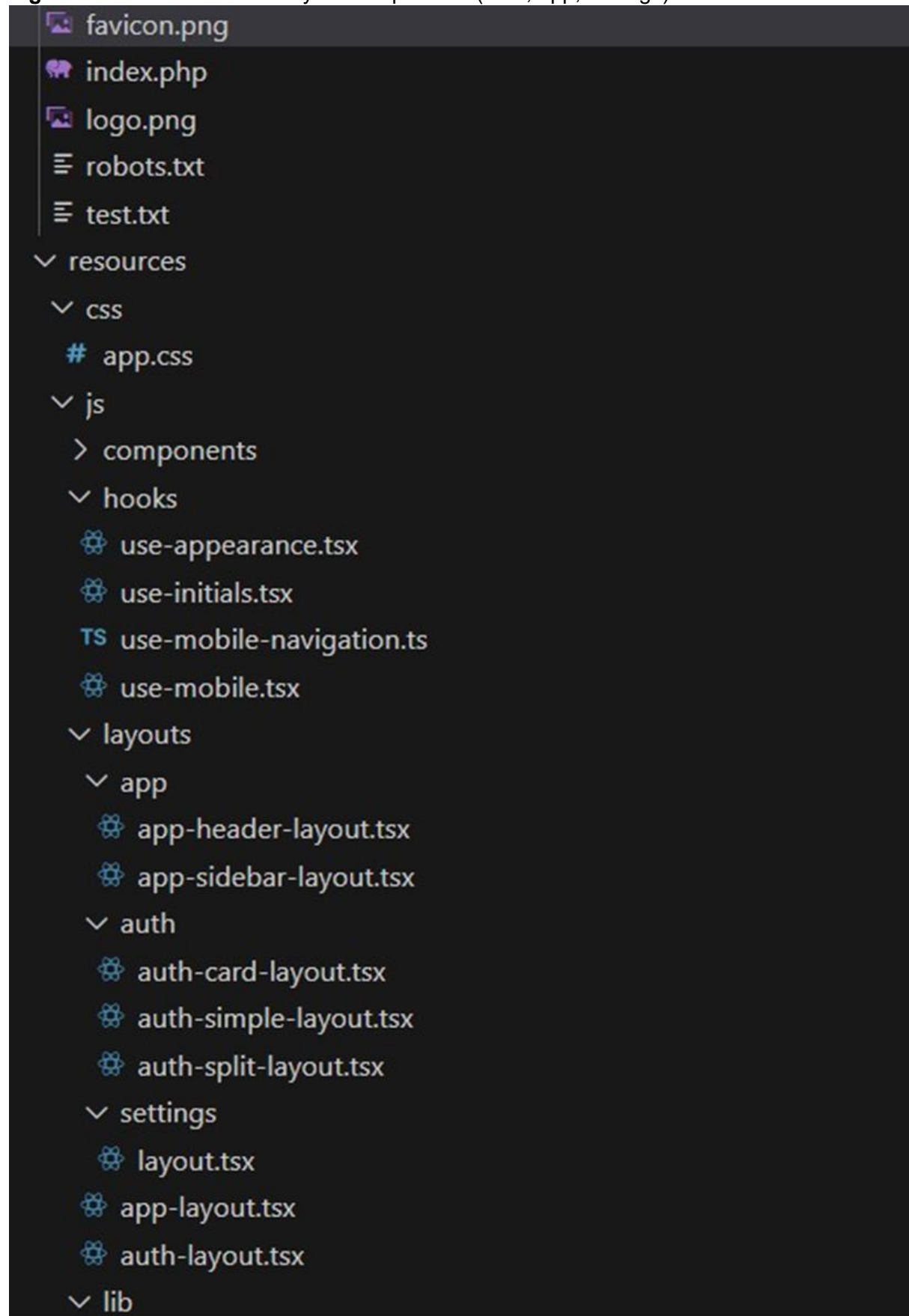


Figure 2: Public directory assets and server configuration files.

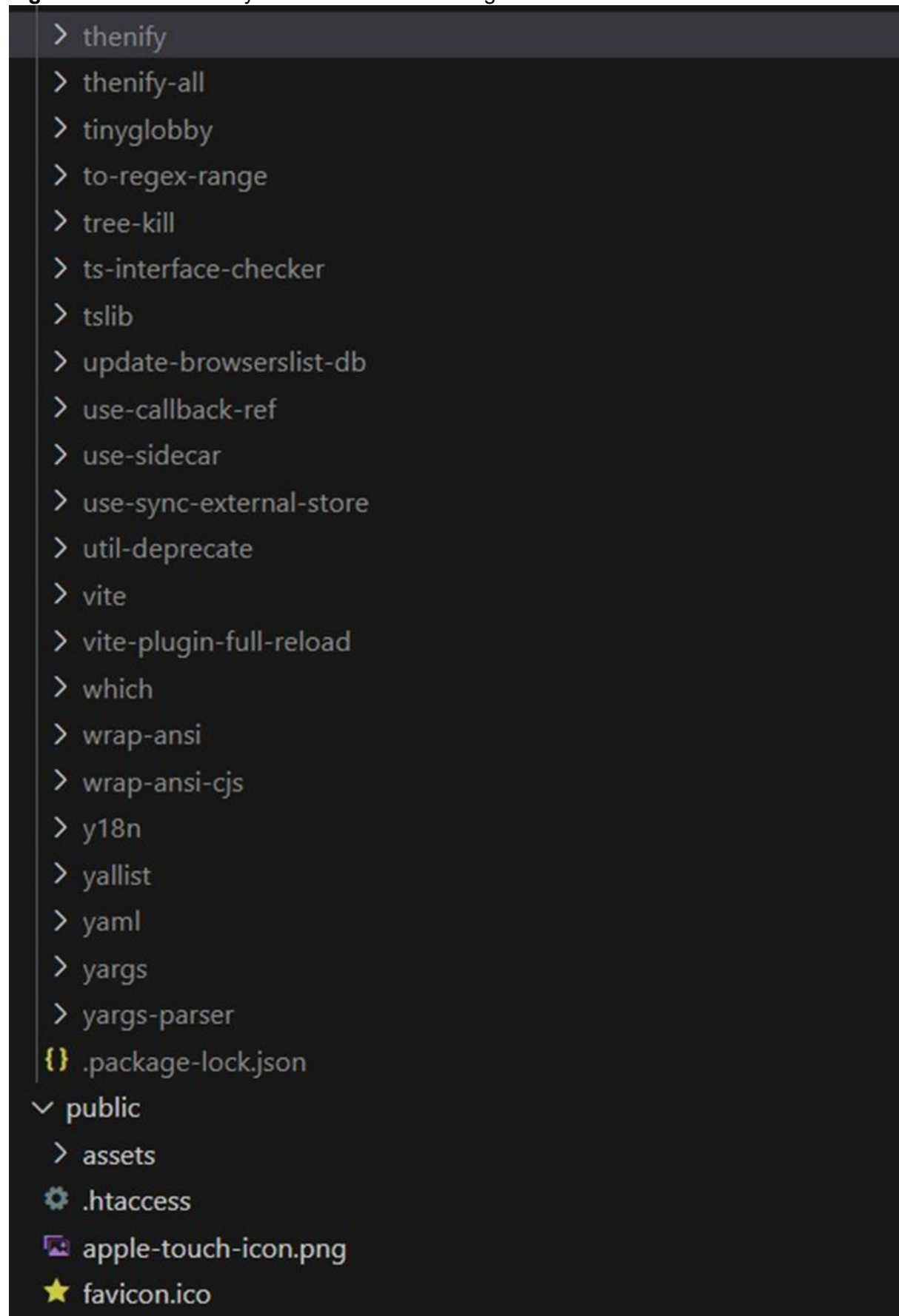


Figure 3: lib/ directory containing core React components for Pages, Auth, Maintenance, and Profile sections.

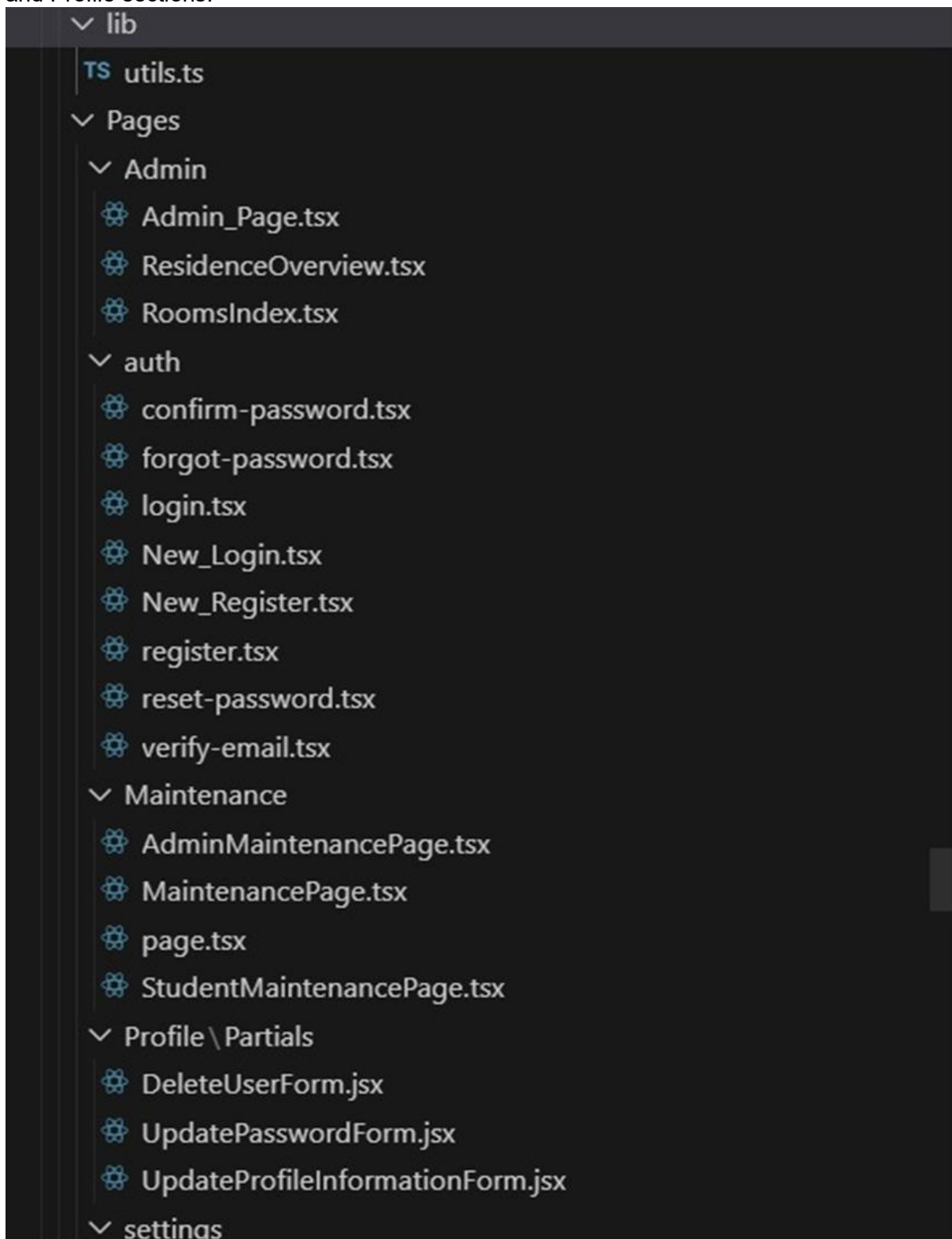


Figure 4: React–Laravel integration using Inertia.js (app.tsx, ssr.tsx, ziggy.js, and Blade templates).

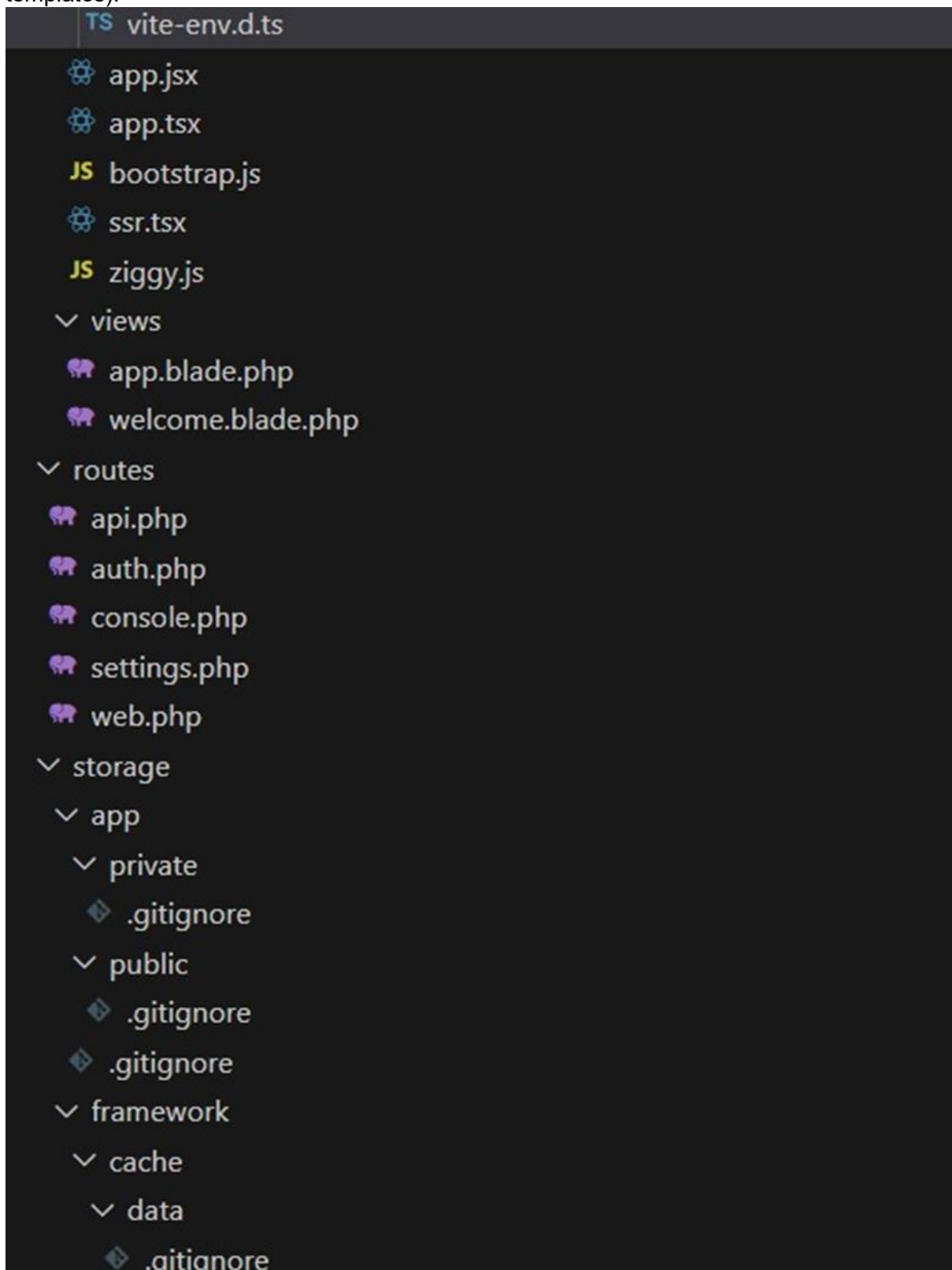


Figure 5: Student dashboard components for Events, Profile, Rooms, and Voting pages.

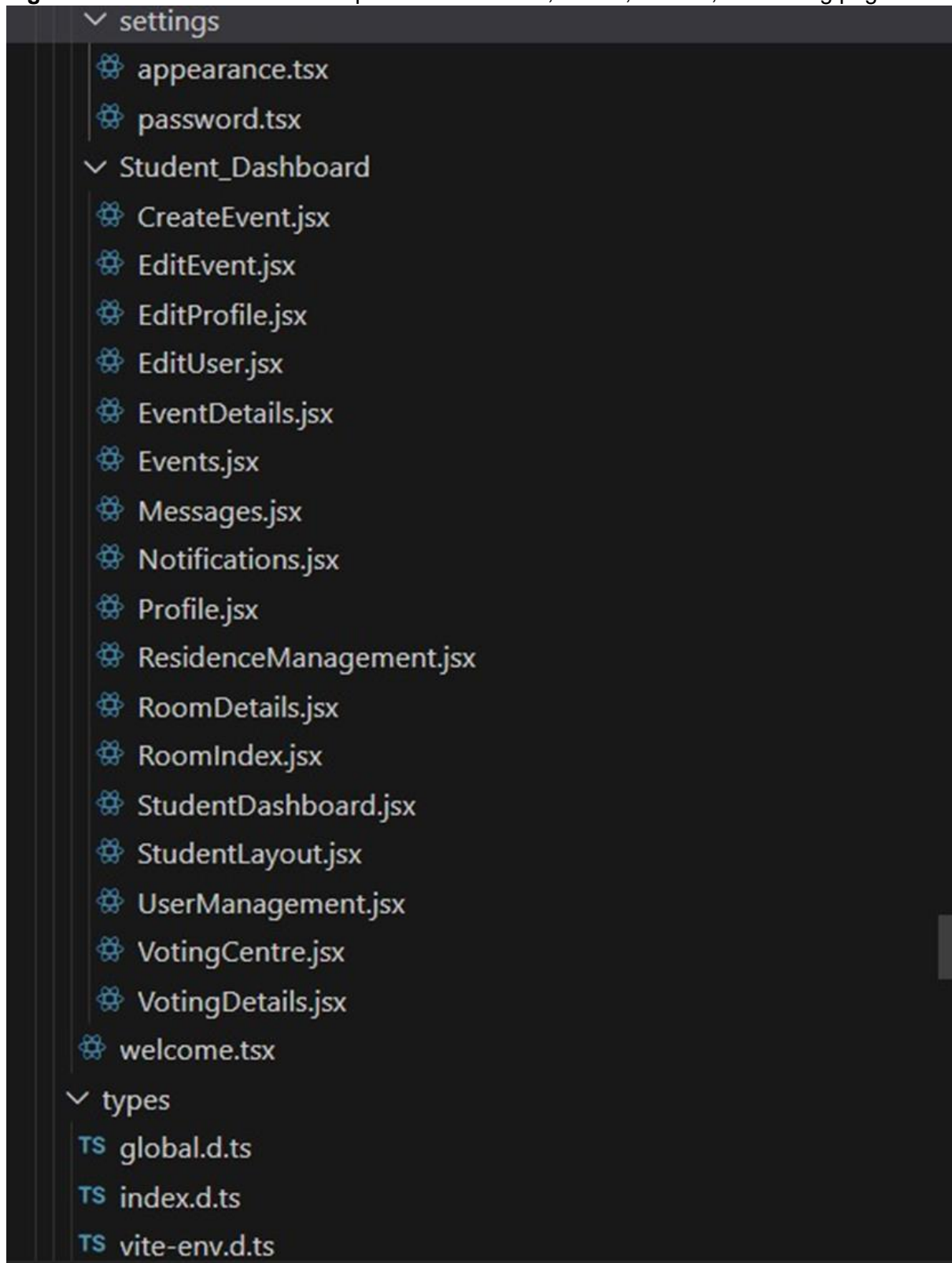
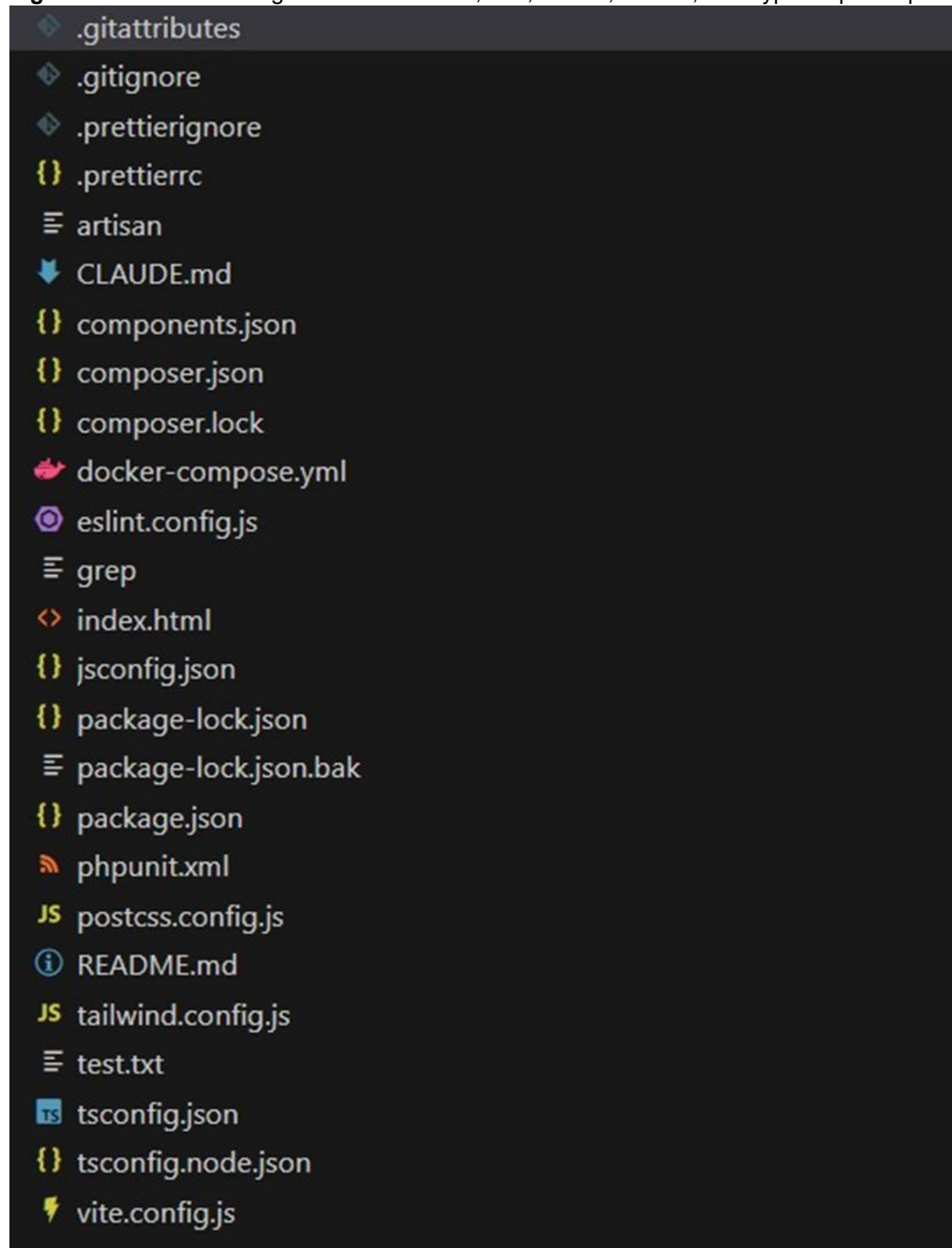


Figure 6: Root-level configuration for Tailwind, Vite, ESLint, Docker, and TypeScript setup.



Application Layer (Backend)

The backend was developed using Laravel 12.34.0, responsible for managing business logic, authentication, and data handling. All APIs were developed in-house and return JSON responses consumed by the React frontend. Authentication and security are handled using Laravel Sanctum, while custom middleware enforces role-based access and data validation.

Security implementations include:

- Password hashing (bcrypt)
- Strict input validation
- Foreign key constraints
- Mass assignment protection
- Encrypted connections (SSL)

The backend architecture follows a modular design pattern with dedicated controllers managing different functional areas, including events, users, residences, groups, chatrooms, and notifications.

Figure 7: Laravel Blade templates and routing files (web.php, auth.php).

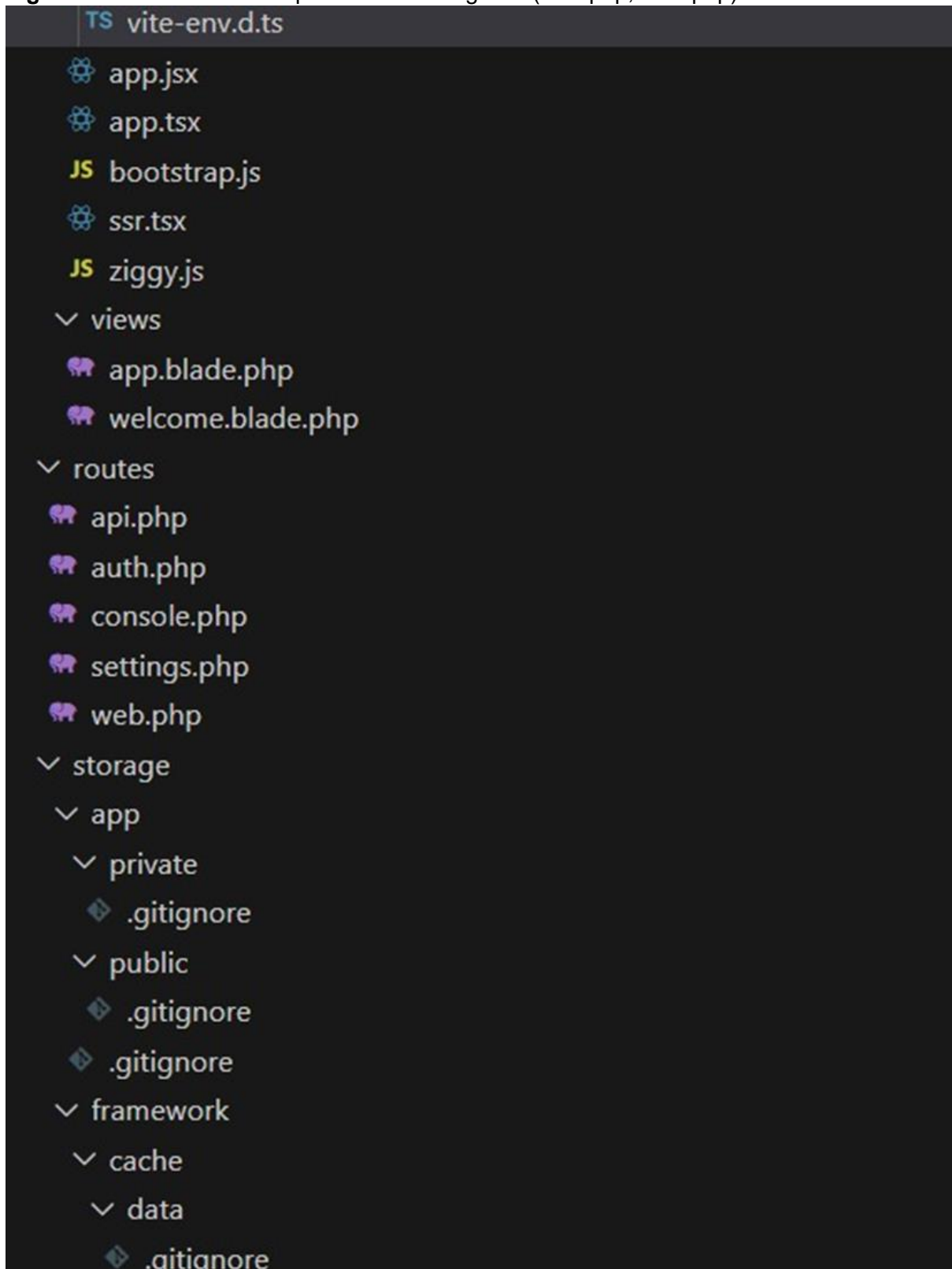


Figure 8: Unit test setup and vendor libraries ensuring backend code quality.

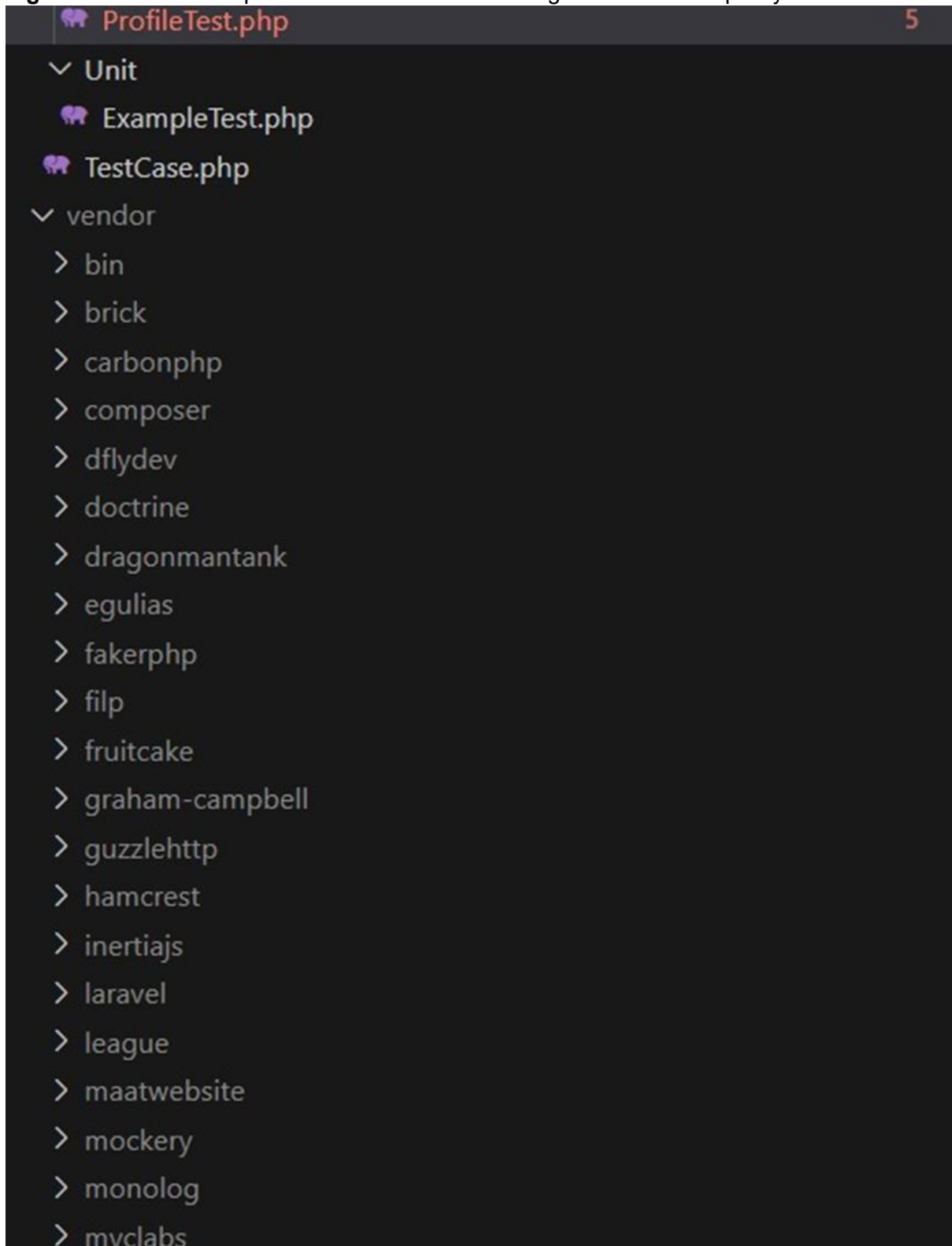


Figure 9: Composer-managed dependencies and Laravel environment setup.

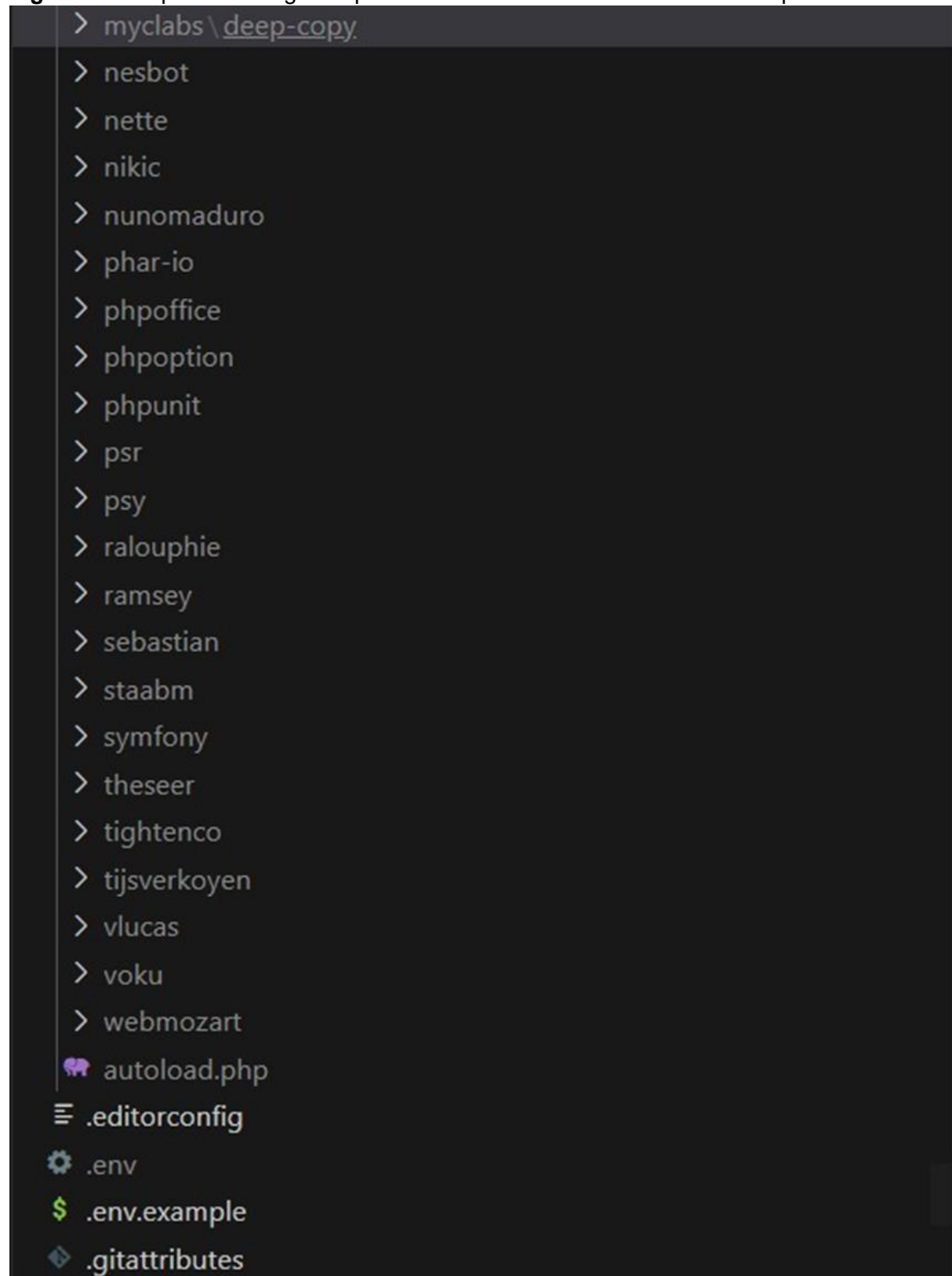


Figure 10: Laravel configuration files and vendor package structure.

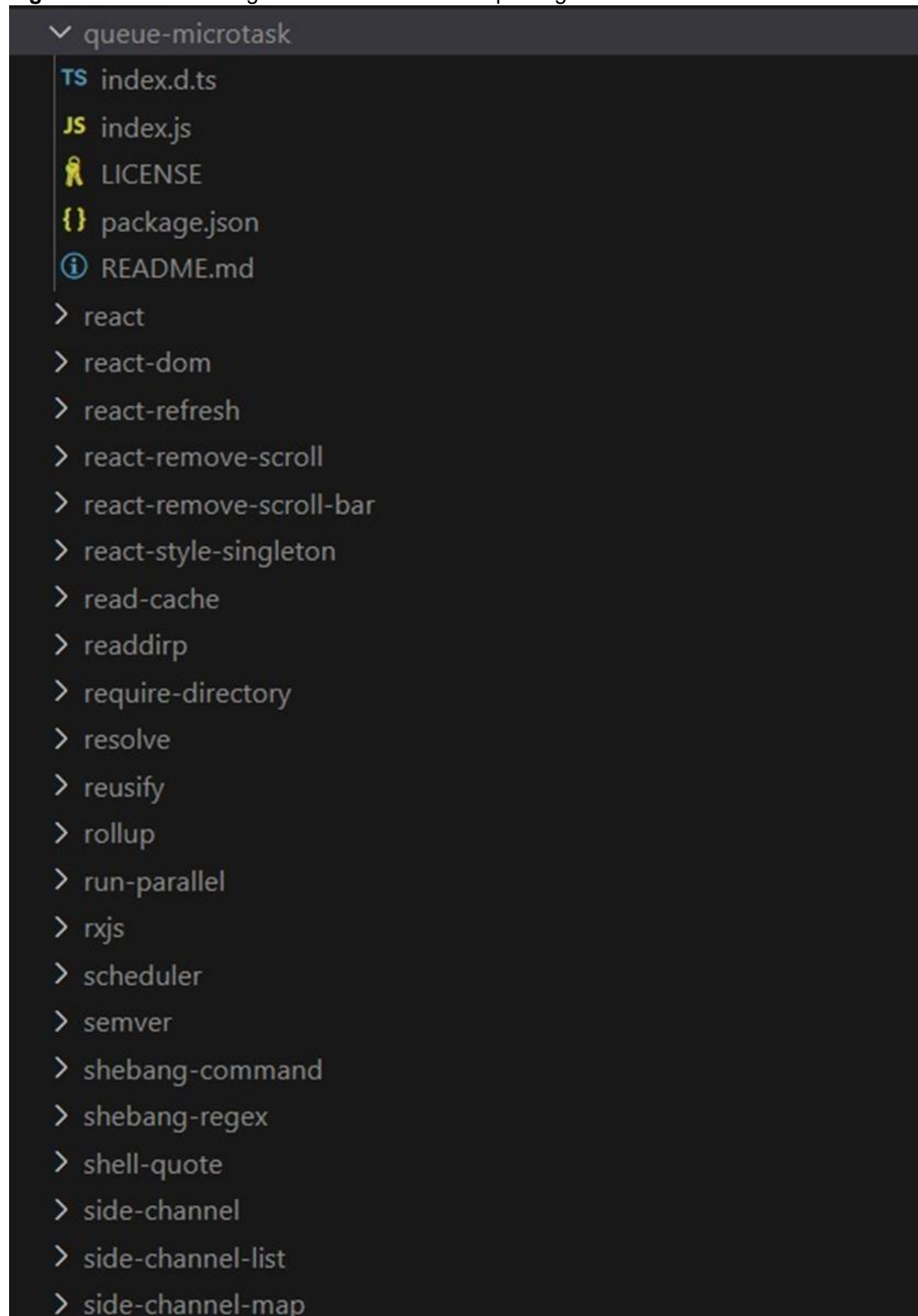
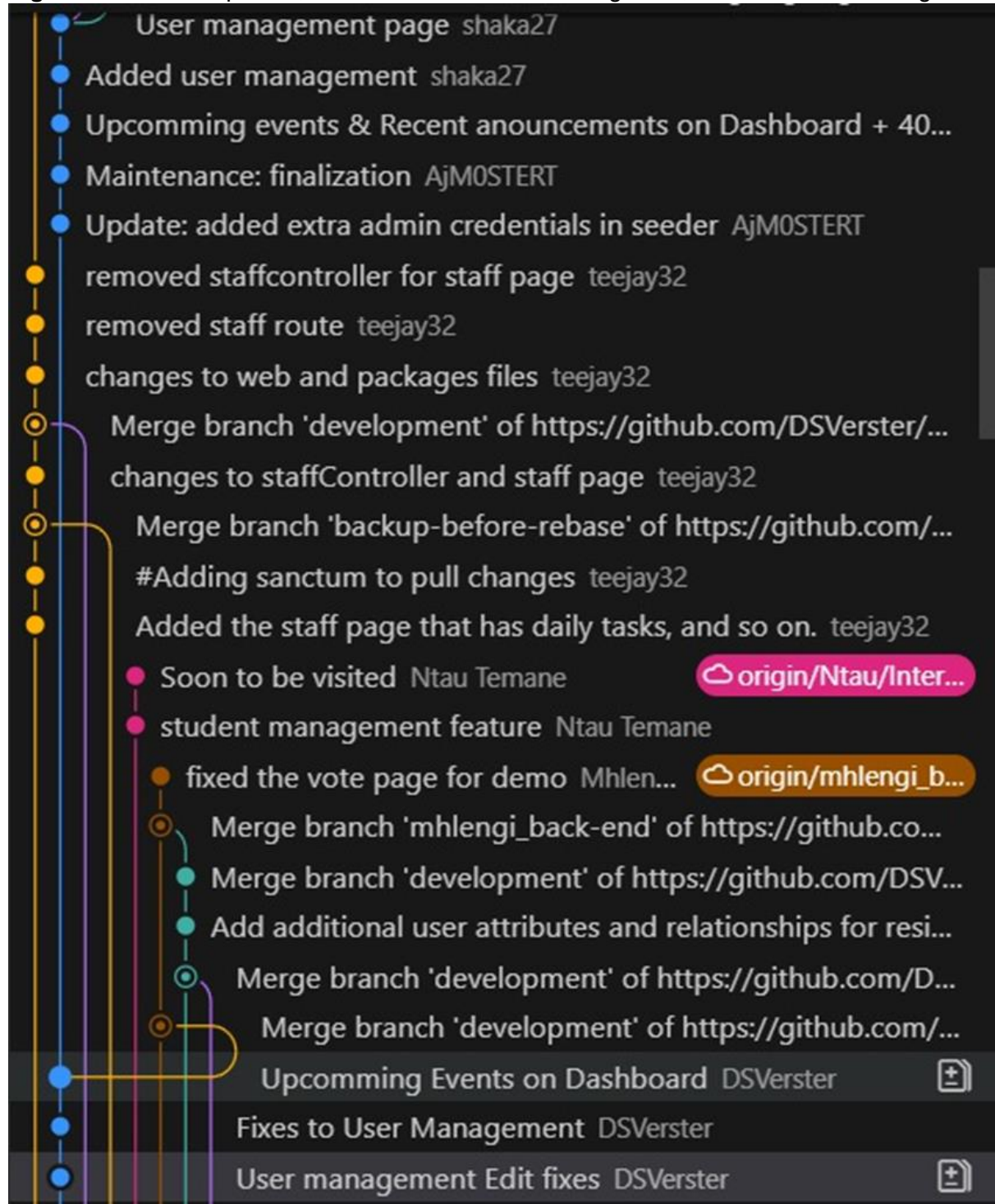


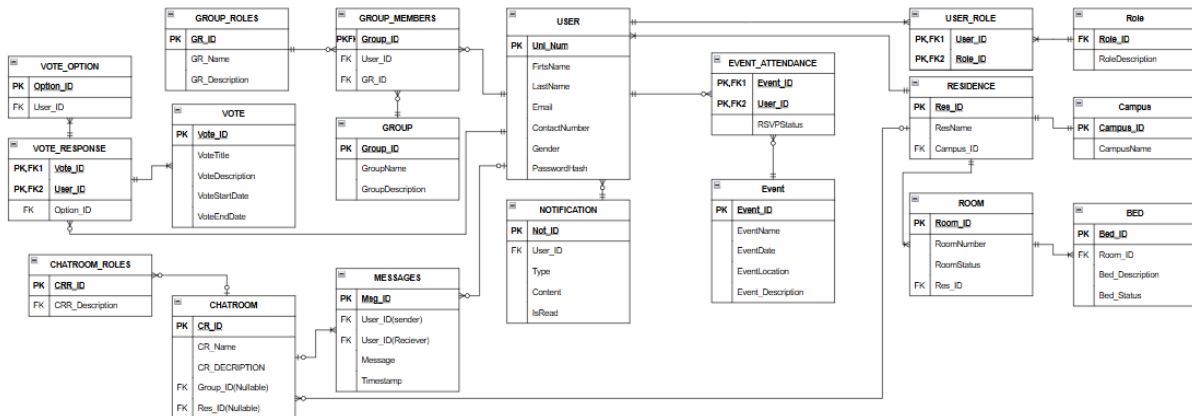
Figure 11: Seeder updates, Sanctum authentication integration, and staff controller logic.



Data Layer (Database)

The RMP uses a local MySQL database. The database stores user profiles, room allocations, event details, group memberships, and voting records. Data security and integrity are maintained through password hashing, validation rules, and relational constraints. Laravel's ORM (Eloquent) simplifies database interactions, ensuring consistency between the backend and the data layer.

Below is an ERD used:



Testing and Quality Assurance

Testing was carried out using PHPUnit, Laravel's native testing framework. Both feature tests and unit tests were implemented to verify authentication, dashboard access, and profile management. Automated testing ensures that critical components function as expected and that new updates do not introduce regressions.

Figure 12: Feature tests implemented for authentication, profile, and dashboard functionality.

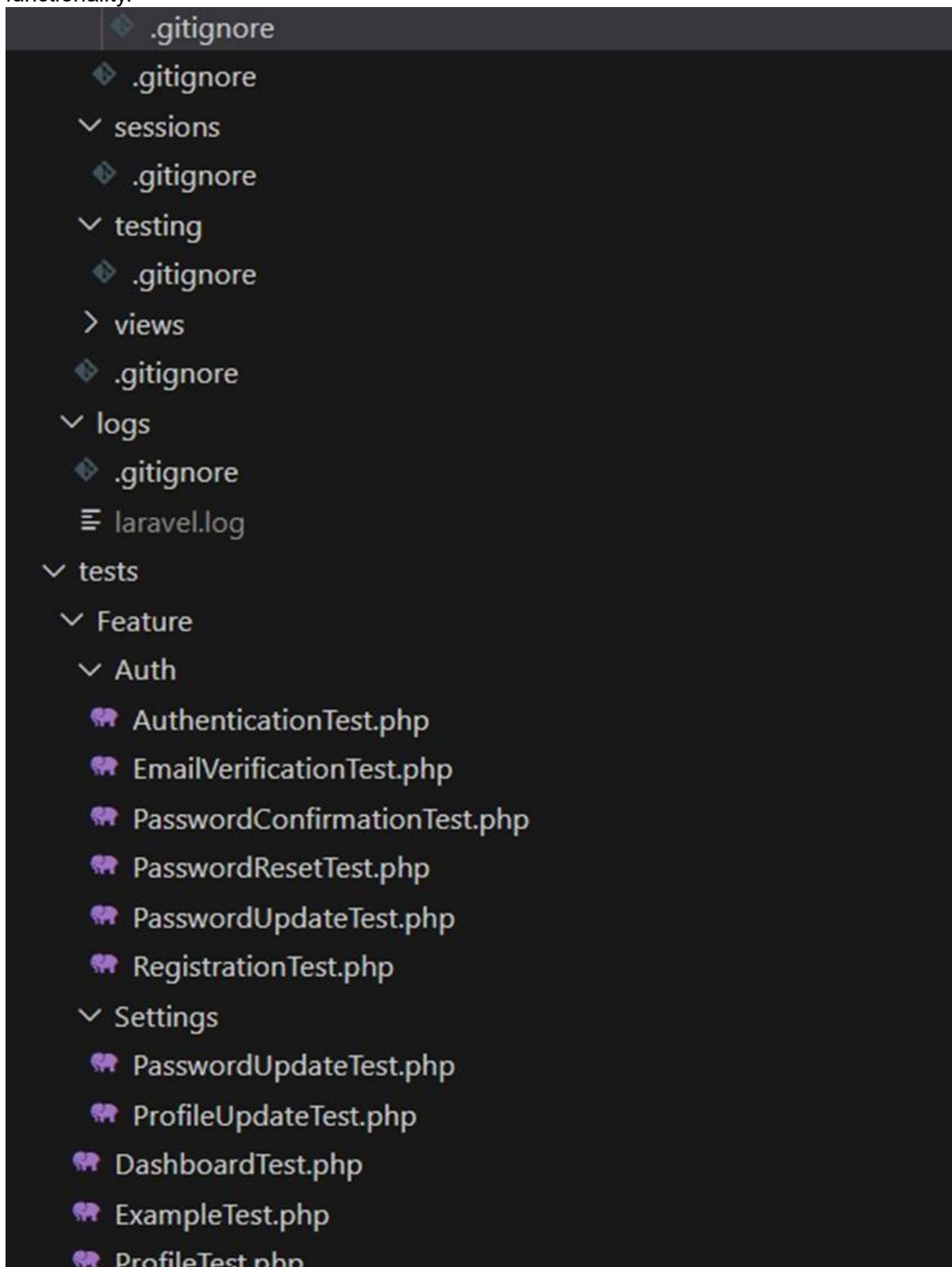


Figure 13: Unit testing suite including TestCase.php and ProfileTest.php

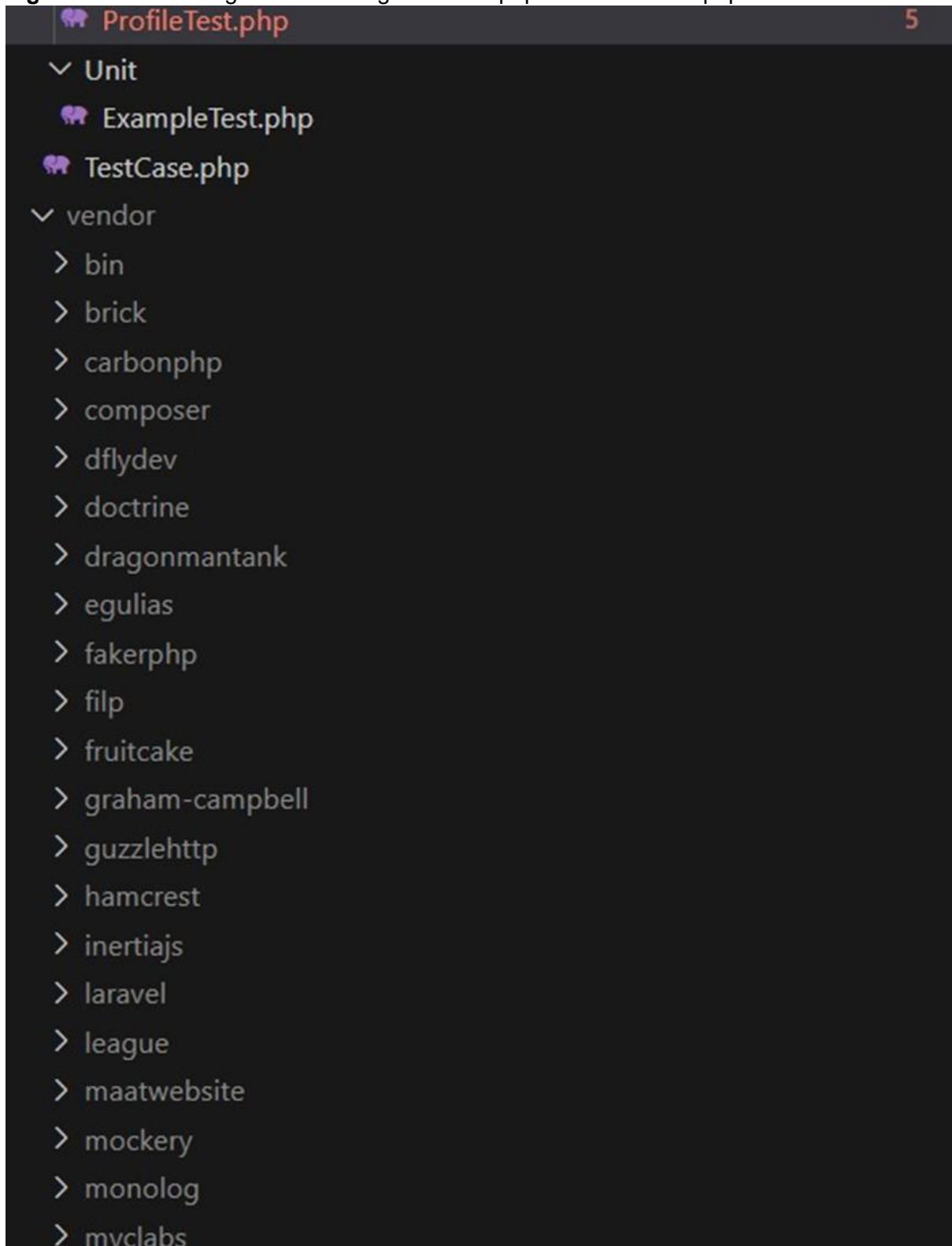
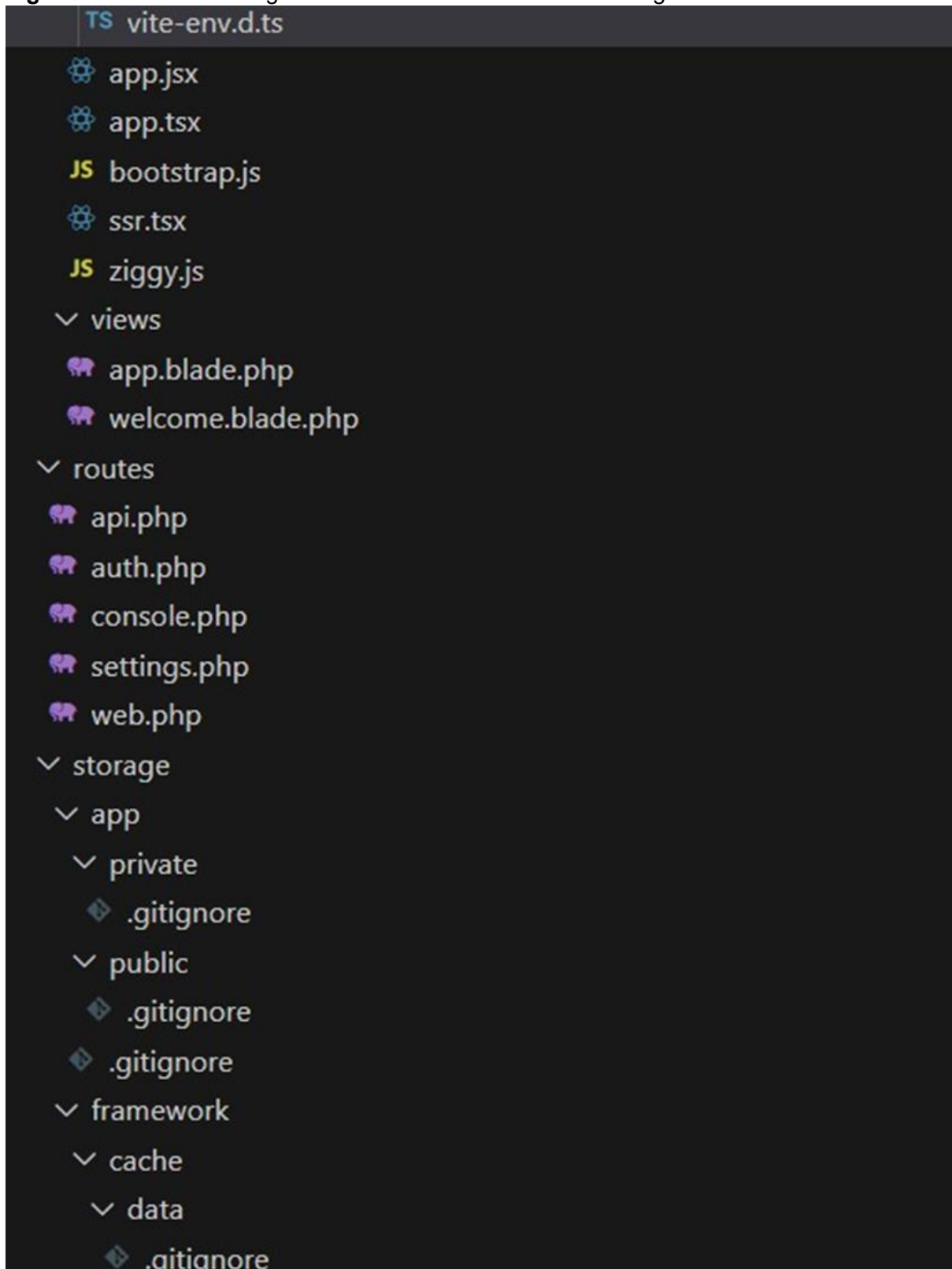


Figure 14: PHPUnit configuration for automated backend testing.



Configuration and Tooling

The project includes extensive configuration for build tools and environment management.

- Vite and Laravel-Vite Plugin are used for fast development and optimized builds.
- Prettier, ESLint, and Docker ensure consistent coding standards and reproducible environments.
- The .env file manages environment-specific variables such as database credentials and API keys.

Figure 15: Project configuration files (.gitignore, .prettierrc, docker-compose.yml, and README.md).

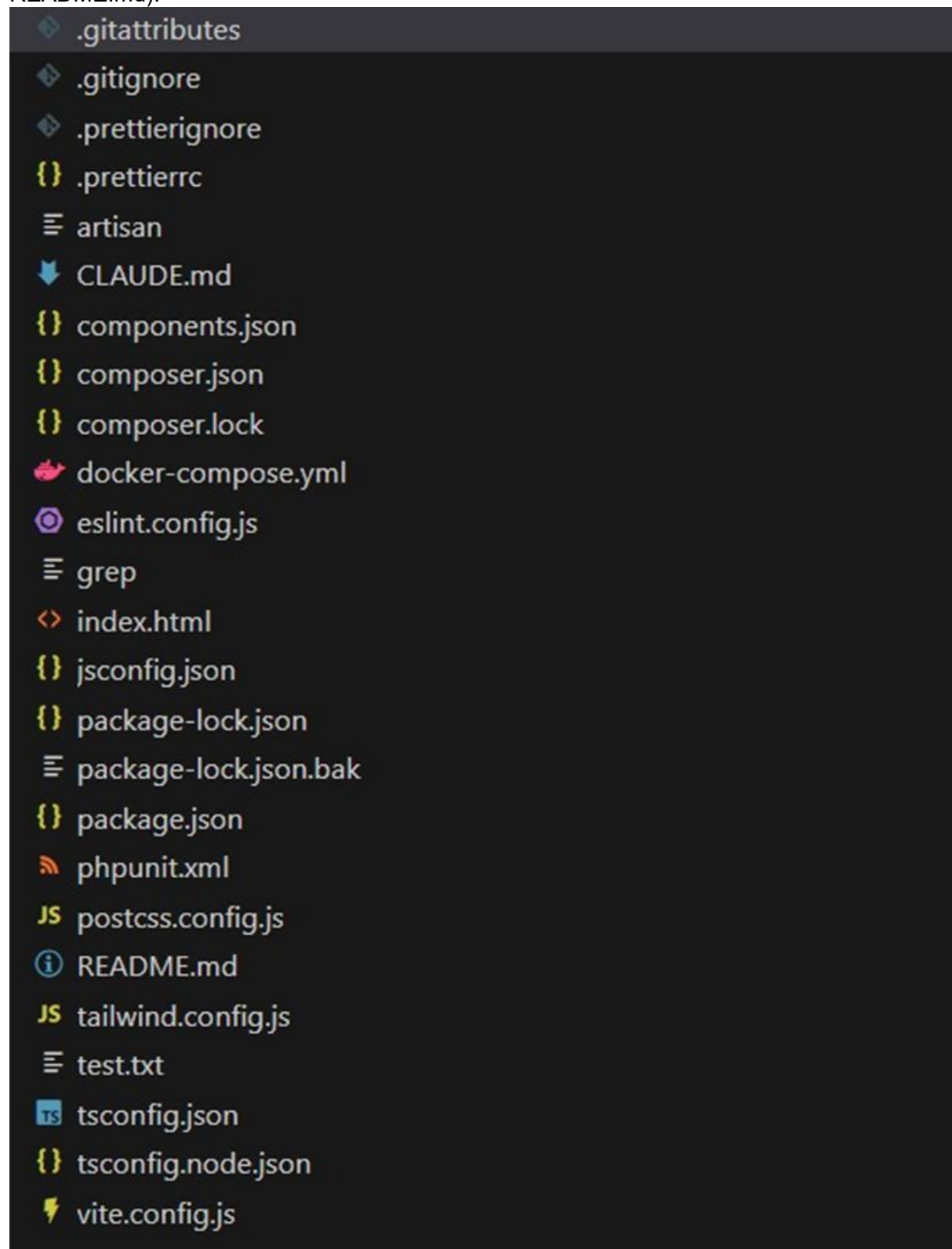


Figure 16: Environment and Composer setup (.env, .editorconfig, autoload.php).

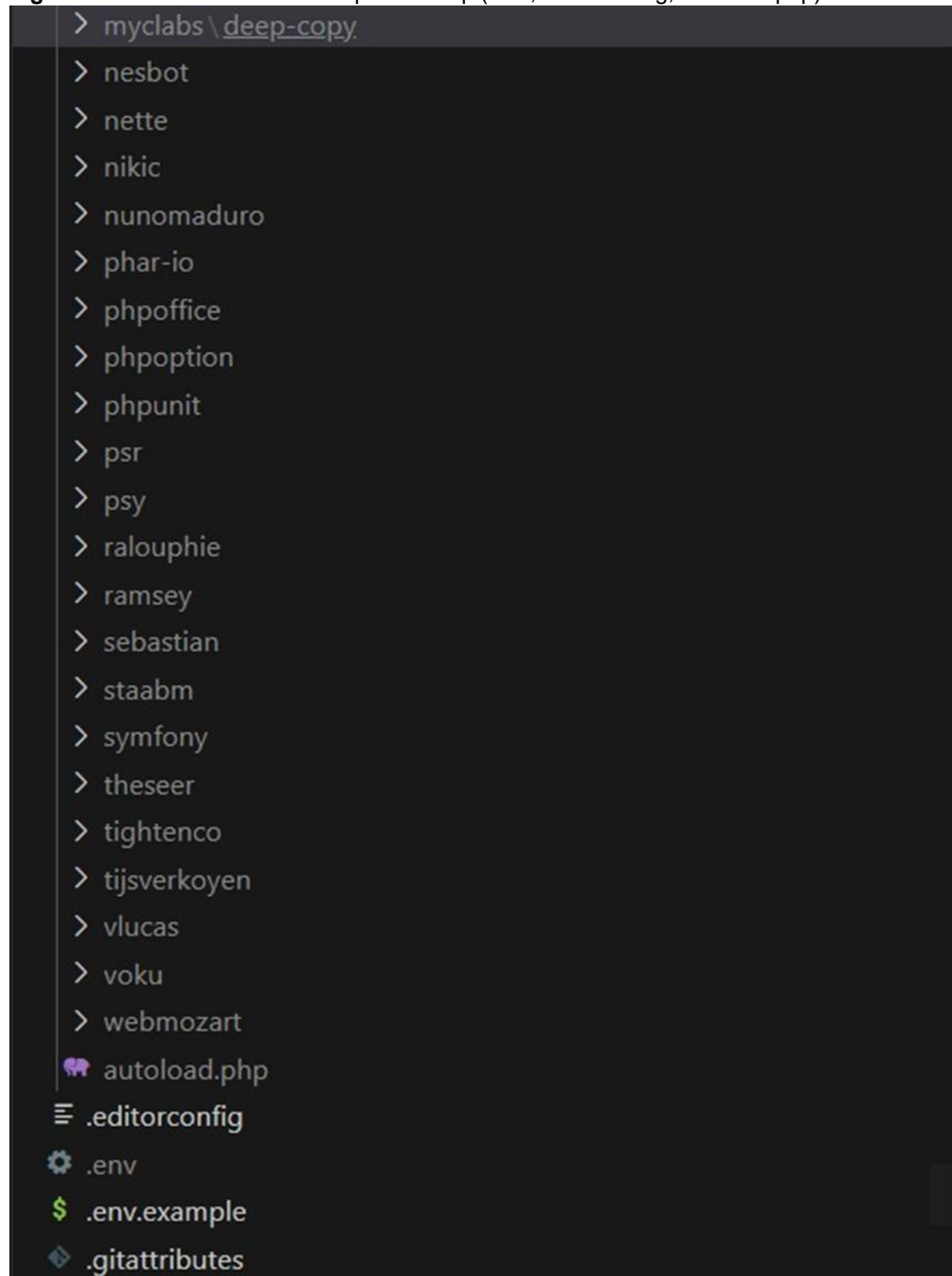
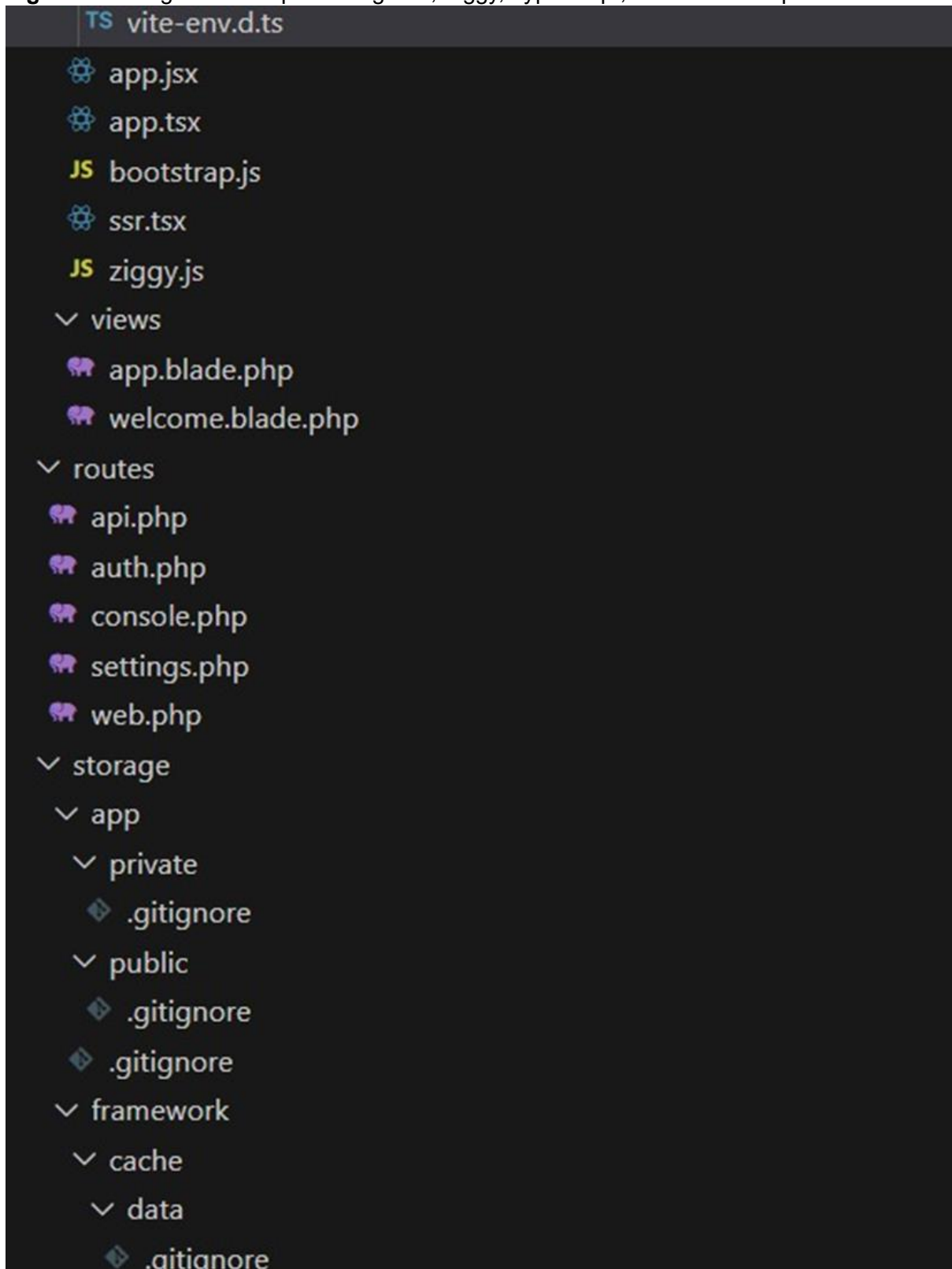


Figure 17: Integration setup showing Vite, Ziggy, TypeScript, and Blade templates



Integration and Data Flow

Data moves seamlessly between the three system layers:

1. Users interact with the React frontend, triggering requests (e.g., booking rooms or submitting feedback).
2. These requests are sent via Axios to the Laravel backend, where business logic is executed.
3. Laravel communicates with MySQL, updates data, and returns JSON responses.
4. The frontend dynamically updates through Inertia.js without reloading pages.
5. Real-time updates such as notifications or chat messages are handled via Pusher.js and Laravel Echo.

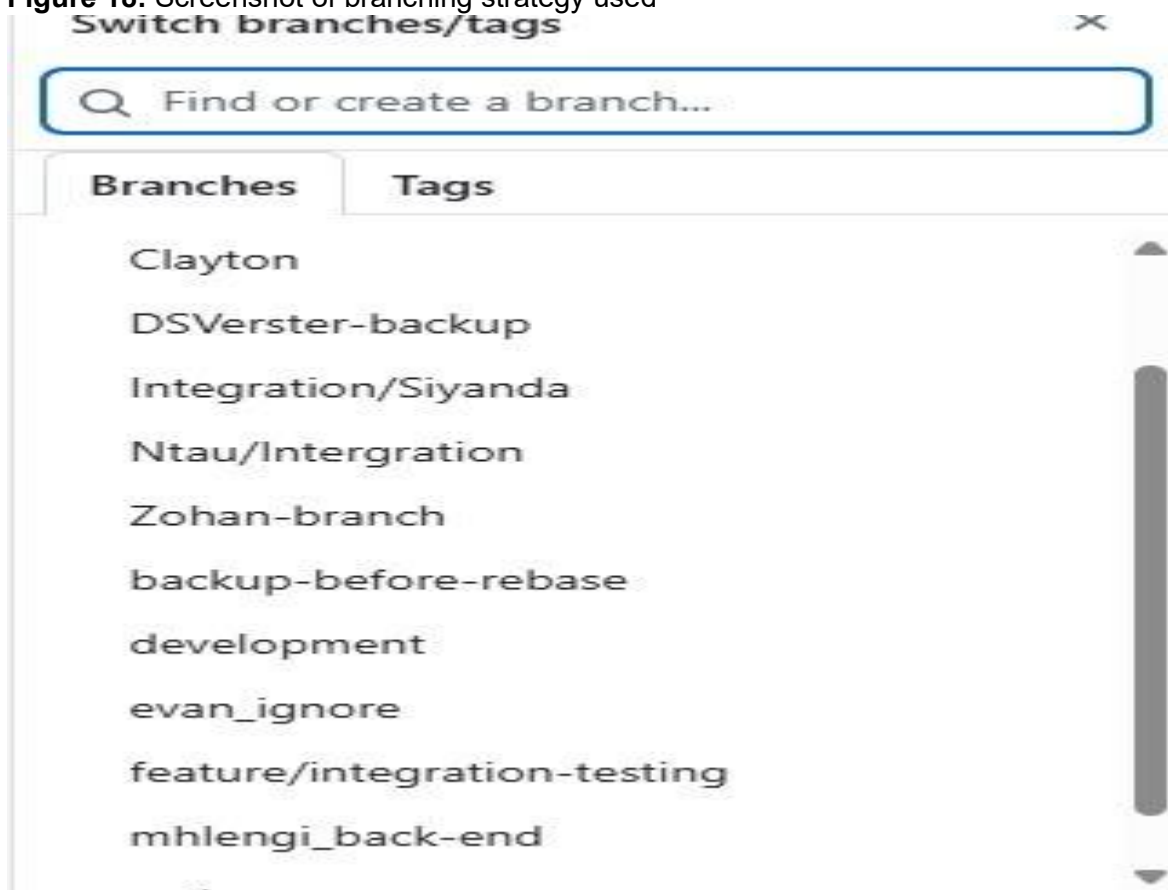
Automated background processes manage email verification and event notifications, improving user engagement and reliability.

Version Control

Overview

Version control was managed using **Git** and **GitHub**, ensuring collaboration and source integrity across a team of 13 developers working virtually across three campuses. All members used feature-based branches and Dockerized environments for consistent development and integration.

Figure 18: Screenshot of branching strategy used



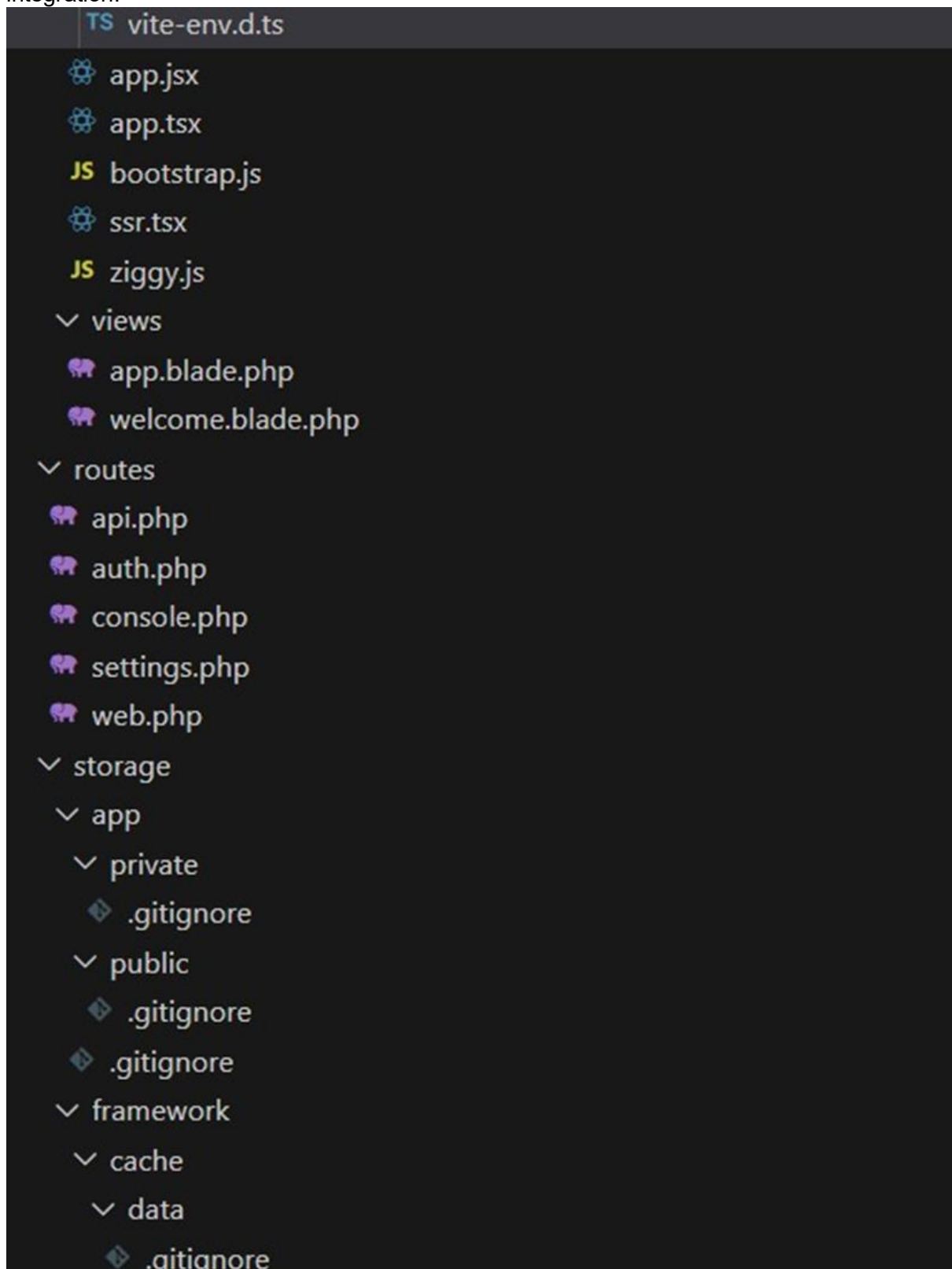
Repository Structure

The project repository, titled **dormdash-web**, contains well-organized directories for the Laravel backend, React frontend, database migrations, and configuration files.

/DormDash-Web

- |— app/ # Laravel application logic
- | |— Http/Controllers # Controllers for system features
- | |— Events/ # Event handling scripts
- |— bootstrap/ # Startup configuration
- |— config/ # Environment and app settings
- |— database/ # Migrations and seeders
- |— public/ # Static assets and entry point
- |— resources/ # React + Inertia frontend
- |— routes/ # Web and API routes
- |— ...

Figure 19: GitHub repository structure showing Laravel backend and React frontend integration.



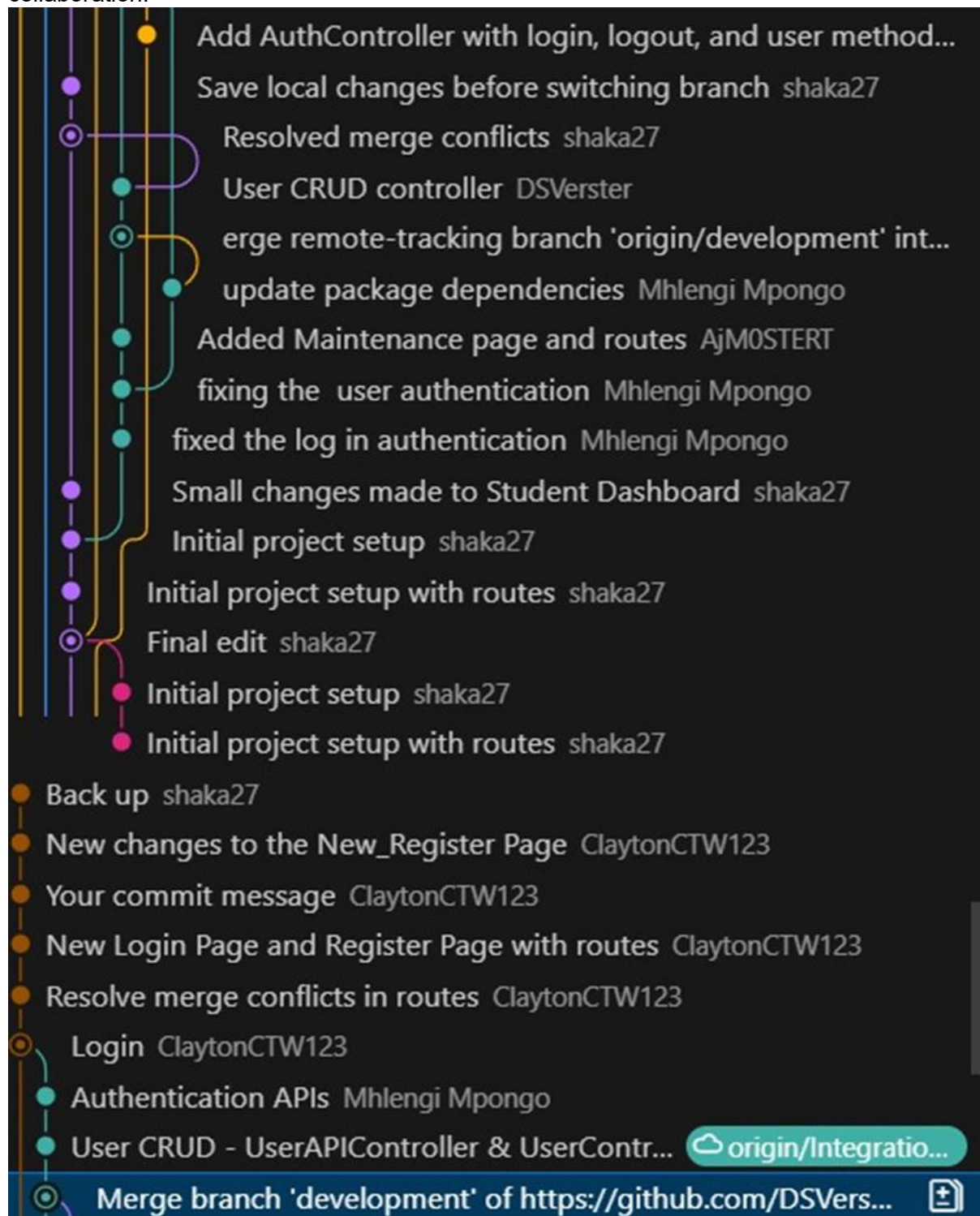
Branching Strategy

The team followed a **feature-branch workflow**, where each developer worked on their own branch and merged through reviewed pull requests.

Branch Example

main	→ <i>Stable production-ready branch</i>
dev	→ <i>Integration and testing branch</i>
feature/room-booking	→ <i>Room booking feature development</i>
feature/event-management	→ <i>Event creation and scheduling</i>
fix/auth-login	→ <i>Bug fixes in authentication</i>
docs/technical	→ <i>Documentation updates</i>

Figure 20: GitHub branches showing feature-based workflow for development and collaboration.



Commit Guidelines

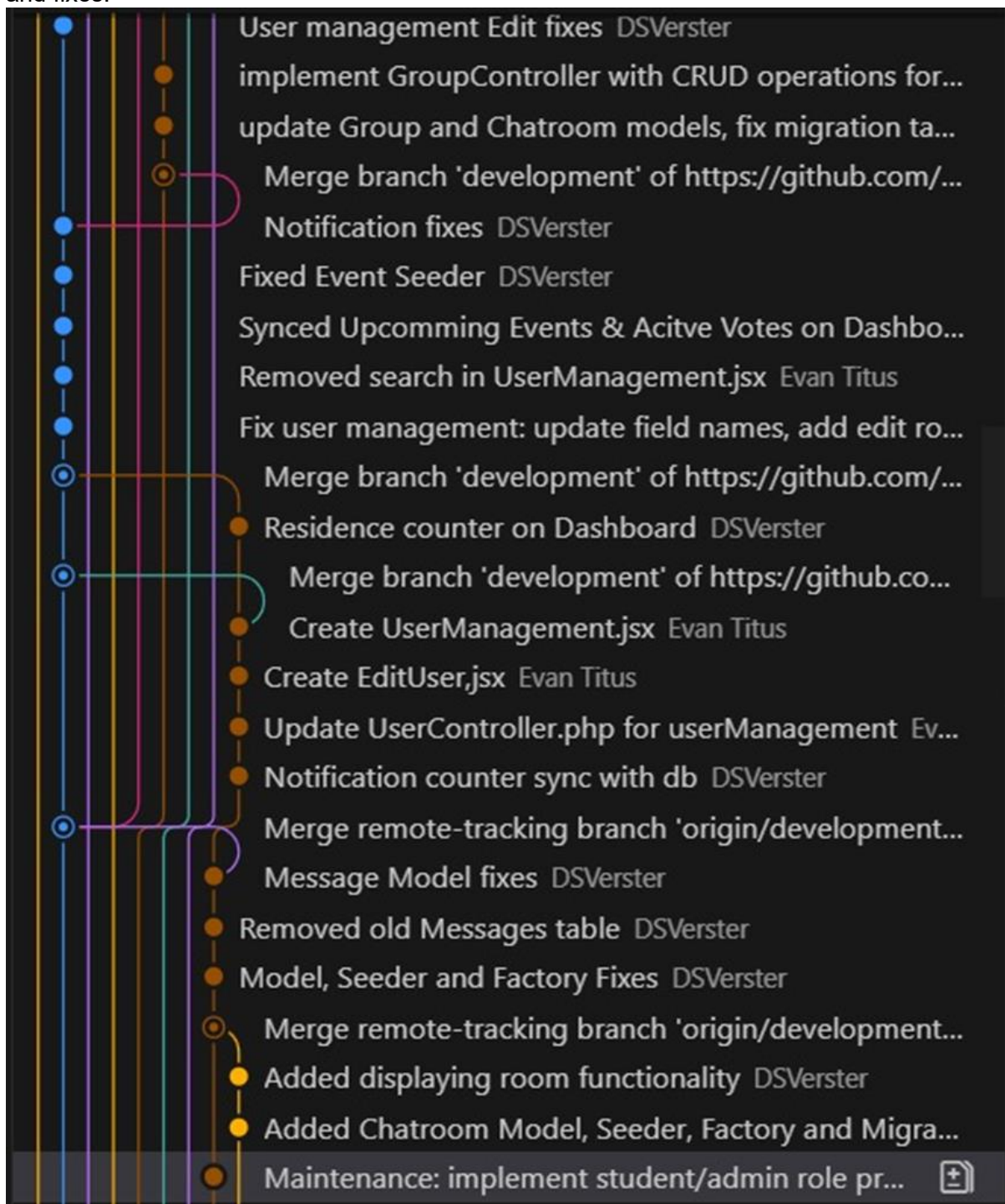
Commit messages were standardized for clarity and traceability. Each commit represented a small, testable change and followed a clear message format:

[Feature] Added EventController and route logic

[Fix] Resolved login validation bug

[Docs] Updated architecture section with new diagrams

Figure 21: Example commit history highlighting continuous feature development, merges, and fixes.



Collaboration Practices

- All members contributed to GitHub through assigned feature branches.
- Pull Requests and peer reviews were used to maintain code quality.
- Discord and WhatsApp facilitated real-time coordination.
- GitHub Issues and milestones were used for progress tracking.

This version control approach fostered discipline, transparency, and alignment across all contributors, closely mirroring real-world software engineering practices.

Frontend Structures

DormDash makes use of React.js alongside a multitude of tools that are necessary for the system to be able to interact with the both the student user and the administrator user simultaneously and provide them with an interactive user interface that receives information and informs the user of it in live time.

Frontend Directory Structure:

/ DormDash-Web

```
|— app/                # Laravel application logic
| |— Http/Controllers  # Controllers for system features
| |— Events/           # Event handling scripts
|— bootstrap/         # Startup configuration
|— config/            # Environment and app settings
|— database/          # Migrations and seeders
|— public/            # Static assets and entry point
|— resources/         # React + Inertia frontend
| |— css/
| |   └— app.css      # Main CSS file
| |— js/
| |   └— components/  # Reusable UI components
| |   └— hooks/       # Custom React hooks
| |   └— layouts/     # Layout components (e.g., navigation)
| |   └— lib/         # Library utilities or wrappers
| |   └— pages/       # Page-level components/routes
| |   └— types/       # TypeScript type definitions
| |   └— app.tsx      # Main application entry point
| |   └— ssr.tsx      # Server-side rendering entry (if applicable)
|— views/            # Blade or frontend view templates
|— routes/           # Web and API route definitions
└— ...
```

The files that are pertinent to this structure are fall under the resources line item.

File	Purpose
css/	This is used to store the application's Css file
components/	Reusable UI components shared across the app.
libs/	Wrappers for third-party libraries used in the project.
pages/	Page components corresponding to routes such as the appearance
layouts/	Layout components like navigation bars or sidebars used across pages.
types/	Used for the listing of what data types are used for users, navigation, and shared info
hooks/	Used to contain custom React hooks for their respective functions
app.tsx	Used to set up the base of the entire user interface

Routing (React Router DOM):

The system makes use of the file *web.php* to handle browser-based routes. Each route defined in *web.php* corresponds to a React page or component, i.e. the login page. Being that both students and staff need to have access to their respective parts of the system when they login

Figure 22: Code from routing file for user login purposes

```
use App\Http\Controllers\ProfileController;
use App\Http\Controllers\MaintenanceRequestController;
use Illuminate\Foundation\Application;
use Illuminate\Support\Facades\Route;
use App\Http\Controllers\AuthController;
use App\Http\Controllers\UserController;
use Illuminate\Support\Facades\Auth;
use Inertia\Inertia;

Route::middleware('guest')->group(function () {
    Route::get('/login', fn() => Inertia::render('auth/New_Login'))->name('login');
    Route::post('/login', [AuthController::class, 'login']);
});

// Make the login page route as the main entry page
Route::get('/', function () {
    return redirect()->route('login');
});

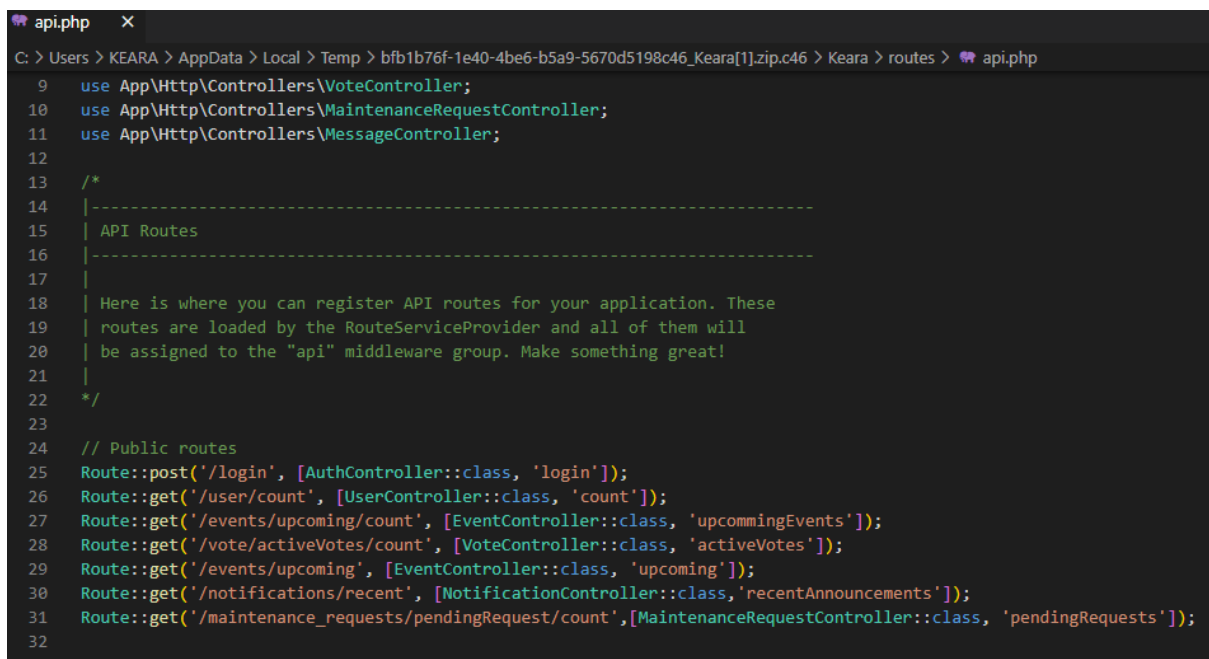
// Login/Register pages
Route::get('/login', fn () => Inertia::render('auth/New_Login'))->name('login');
Route::post('/login', [AuthController::class, 'login']);
Route::get('/register', fn () => Inertia::render('auth/New_Register'))->name('register');
Route::post('/register', [UserController::class, 'store']);
```

API Communication:

The system makes use of a custom-made API that consists of a number of .php files to facilitate the communication of the user interface and pertinent components of the system that is necessary for the system to operate.

Below is an example of one of the files used to create the API. This file organizes and controls how the backend API listens and responds to various frontend requests for what the program calls for.

Figure 23: Example code from the API file



```
api.php
C: > Users > KEARA > AppData > Local > Temp > bfb1b76f-1e40-4be6-b5a9-5670d5198c46_Keara[1].zip.c46 > Keara > routes > api.php
9  use App\Http\Controllers\VoteController;
10 use App\Http\Controllers\MaintenanceRequestController;
11 use App\Http\Controllers\MessageController;
12
13 /*
14 |-----
15 | API Routes
16 |-----
17 |
18 | Here is where you can register API routes for your application. These
19 | routes are loaded by the RouteServiceProvider and all of them will
20 | be assigned to the "api" middleware group. Make something great!
21 |
22 */
23
24 // Public routes
25 Route::post('/login', [AuthController::class, 'login']);
26 Route::get('/user/count', [UserController::class, 'count']);
27 Route::get('/events/upcoming/count', [EventController::class, 'upcomingEvents']);
28 Route::get('/vote/activeVotes/count', [VoteController::class, 'activeVotes']);
29 Route::get('/events/upcoming', [EventController::class, 'upcoming']);
30 Route::get('/notifications/recent', [NotificationController::class, 'recentAnnouncements']);
31 Route::get('/maintenance_requests/pendingRequest/count', [MaintenanceRequestController::class, 'pendingRequests']);
32
33
```

File :api.php

<?php

```
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Route;
use App\Http\Controllers\AuthController;
use App\Http\Controllers\UserController;
use App\Http\Controllers\NotificationController;
use App\Http\Controllers\EventController;
use App\Http\Controllers\VoteController;
use App\Http\Controllers\MaintenanceRequestController;
use App\Http\Controllers\MessageController;

/*
|-----
| API Routes
|-----
|
| Here is where you can register API routes for your application. These
| routes are loaded by the RouteServiceProvider and all of them will
| be assigned to the "api" middleware group. Make something great!
|
*/

// Public routes
Route::post('/login', [AuthController::class, 'login']);
Route::get('/user/count', [UserController::class, 'count']);
Route::get('/events/upcoming/count', [EventController::class, 'upcommingEvents']);
Route::get('/vote/activeVotes/count', [VoteController::class, 'activeVotes']);
```

```

Route::get('/events/upcoming', [EventController::class, 'upcoming']);
Route::get('/notifications/recent', [NotificationController::class, 'recentAnnouncements']);
Route::get('/maintenance_requests/pendingRequest/count', [MaintenanceRequestController::class, 'pendingRequests']);

```

// Protected routes (require authentication)

```

Route::middleware('auth:sanctum')->group(function () {
    Route::post('/logout', [AuthController::class, 'logout']);
    Route::get('/user', [AuthController::class, 'user']);

```

// EVENT API routes

```

Route::post('/events', [App\Http\Controllers\EventController::class, 'store']);
Route::put('/events/{id}', [App\Http\Controllers\EventController::class, 'update']);
Route::delete('/events/{id}', [App\Http\Controllers\EventController::class, 'destroy']);

```

// RESIDENCE protected routes here

```

Route::get('/residences', [App\Http\Controllers\ResidenceController::class, 'index']);
Route::post('/residences', [App\Http\Controllers\ResidenceController::class, 'store']);
Route::get('/residences/{id}', [App\Http\Controllers\ResidenceController::class, 'show']); //Display single specific residence
Route::put('/residences/{id}', [App\Http\Controllers\ResidenceController::class, 'update']);
Route::delete('/residences/{id}', [App\Http\Controllers\ResidenceController::class, 'destroy']);

```

//ROOM protected routes here

```

Route::get('/rooms', [App\Http\Controllers\RoomController::class, 'index']);
Route::post('/rooms', [App\Http\Controllers\RoomController::class, 'store']);
Route::get('/rooms/{id}', [App\Http\Controllers\RoomController::class, 'show']); //Display single specific room
Route::put('/rooms/{id}', [App\Http\Controllers\RoomController::class, 'update']);
Route::delete('/rooms/{id}', [App\Http\Controllers\RoomController::class, 'destroy']);

```

//NOTIFICATION protected routes here

```

Route::get('/notifications', [App\Http\Controllers\NotificationController::class, 'index']);

```

```

Route::post('/notifications', [App\Http\Controllers\NotificationController::class, 'store']);

Route::get('/notifications/{id}', [App\Http\Controllers\NotificationController::class,
'show']); //Display single specific notification

Route::put('/notifications/{id}',
[App\Http\Controllers\NotificationController::class, 'update']);

Route::delete('/notifications/{id}', [App\Http\Controllers\NotificationController::class,
'destroy']);

//Route::get('/notifications/count', [NotificationController::class, 'unreadCount']);

//Route::get('/notifications/recent', [NotificationController::class, 'recentAnnouncements']);


// CHATROOM protected routes

Route::get('/chatrooms', [App\Http\Controllers\ChatroomController::class, 'index']);
Route::post('/chatrooms', [App\Http\Controllers\ChatroomController::class, 'store']);
Route::get('/chatrooms/{id}', [App\Http\Controllers\ChatroomController::class, 'show']);
Route::put('/chatrooms/{id}', [App\Http\Controllers\ChatroomController::class, 'update']);
Route::delete('/chatrooms/{id}', [App\Http\Controllers\ChatroomController::class,
'destroy']);


// MESSAGES protected routes

Route::get('/messages', [App\Http\Controllers\MessageController::class, 'index']);

Route::get('/messages/residence-users', [App\Http\Controllers\MessageController::class,
'getResidenceUsers']);

Route::get('/messages/direct/{userId}', [App\Http\Controllers\MessageController::class,
'getDirectMessages']);

Route::get('/chatrooms/{chatroomId}/messages',
[App\Http\Controllers\MessageController::class, 'getChatroomMessages']);

Route::post('/messages', [App\Http\Controllers\MessageController::class, 'store']);
Route::get('/messages/{id}', [App\Http\Controllers\MessageController::class, 'show']);
Route::put('/messages/{id}', [App\Http\Controllers\MessageController::class, 'update']);
Route::delete('/messages/{id}', [App\Http\Controllers\MessageController::class, 'destroy']);
Route::get('/messages/count', [MessageController::class, 'unreadCount']);


// VOTING protected routes

Route::get('/votes', [App\Http\Controllers\VoteController::class, 'index']);
Route::post('/votes', [App\Http\Controllers\VoteController::class, 'store']);

```

```

Route::get('/votes/{id}', [App\Http\Controllers\VoteController::class, 'show']);
Route::put('/votes/{id}', [App\Http\Controllers\VoteController::class, 'update']);
Route::delete('/votes/{id}', [App\Http\Controllers\VoteController::class, 'destroy']);
Route::post('/votes/submit-response', [App\Http\Controllers\VoteController::class,
'submitResponse']);

});

```

// CRUD for Users

```

Route::middleware('auth:sanctum')->group(function () {
    Route::get('/user', [UserController::class, 'index']);
    Route::post('/user', [UserController::class, 'store']);
    Route::put('/user/{id}', [UserController::class, 'update']);
    Route::delete('/user/{id}', [UserController::class, 'destroy']);

});

```

//testing the CORS setup

```

Route::get('/test-cors', function () {
    return response()->json([
        'message' => 'CORS is working!',
        'status' => 'success',
        'timestamp' => now()
    ]);
});

```

UI/UX Design:

Each user type, student or administrator, has their own dashboard that they will see once they login to the system. All the pages, alerts and functionalities for the respective tabs in their dashboard as well as the dashboard themselves were created using the aforementioned items:

- TailwindCSS
- Bootstrap
- Radix UI and Headless UI
- Framer Motion
- Lucide-React
- Laravel Echo and Pusher.js

The design for the Login and Sign-up pages follow a different design approach, they are mainly composed of a series of textboxes where they are to enter their necessary details as well as the button to register their new account or to sign in to their existing account.

Both dashboards for the administrator user and the student user, while have provide the respective users with details tailored to their role, follow the same design to display the pages and notifications.

All dashboards have the following pages in common:

- Dashboards
- Notifications
- Messages
- Profile
- Maintenance
- Voting
- Rooms

The administrators have reserved access to two other pages that all students will not have access to :

- User management
- Residence management

Example Pages

Figure 24: Sign in page

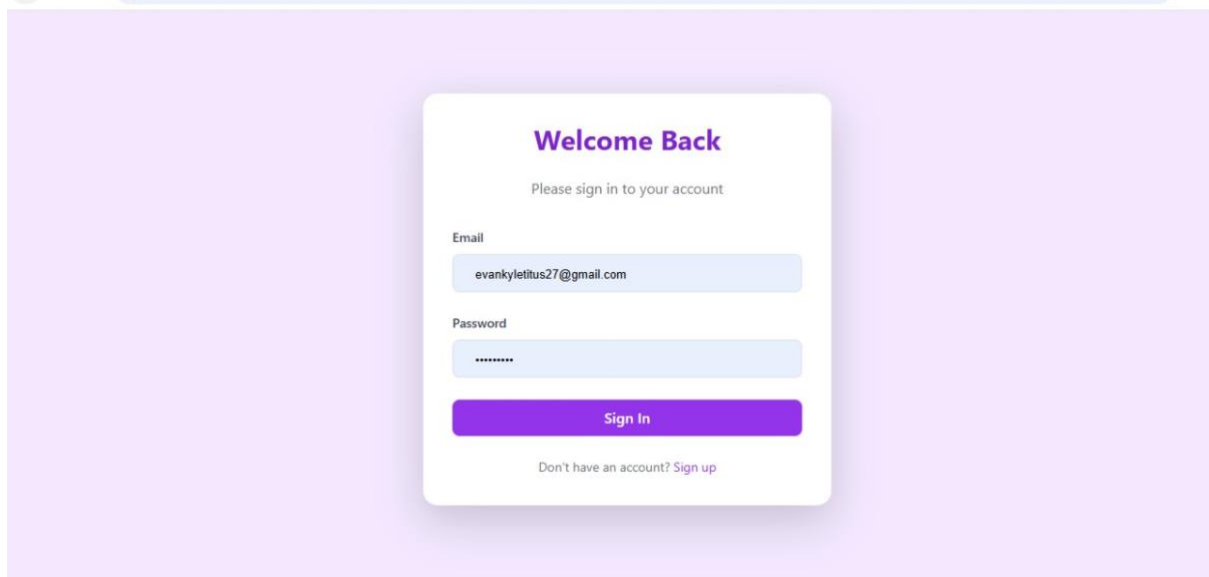


Figure 25: The Events Management Page

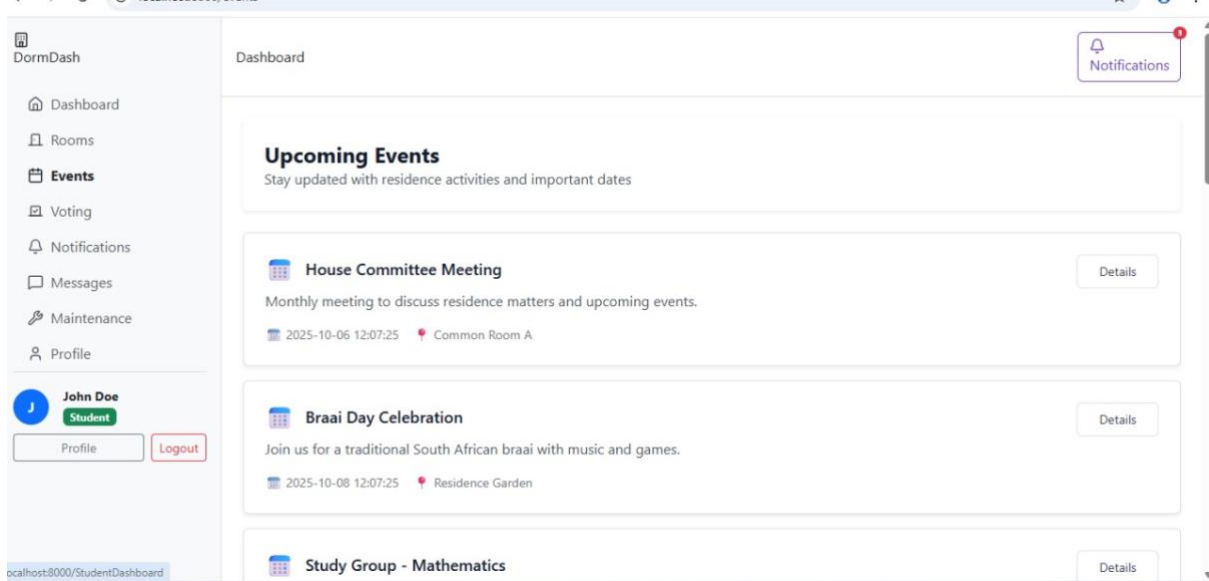


Figure 26: Profile Page

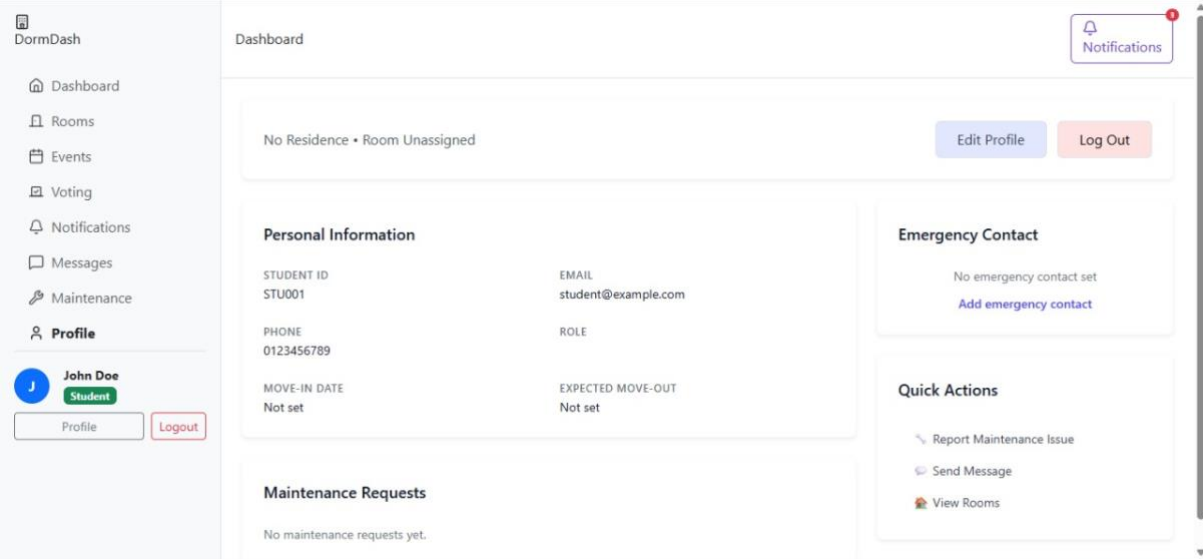


Figure 27: The Residence Management Page

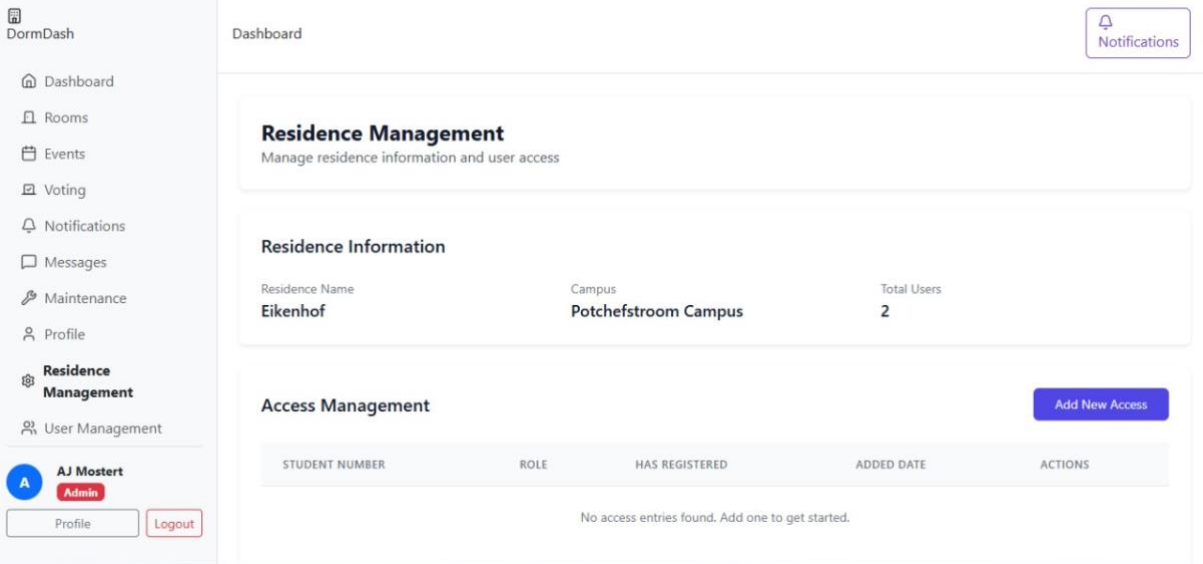
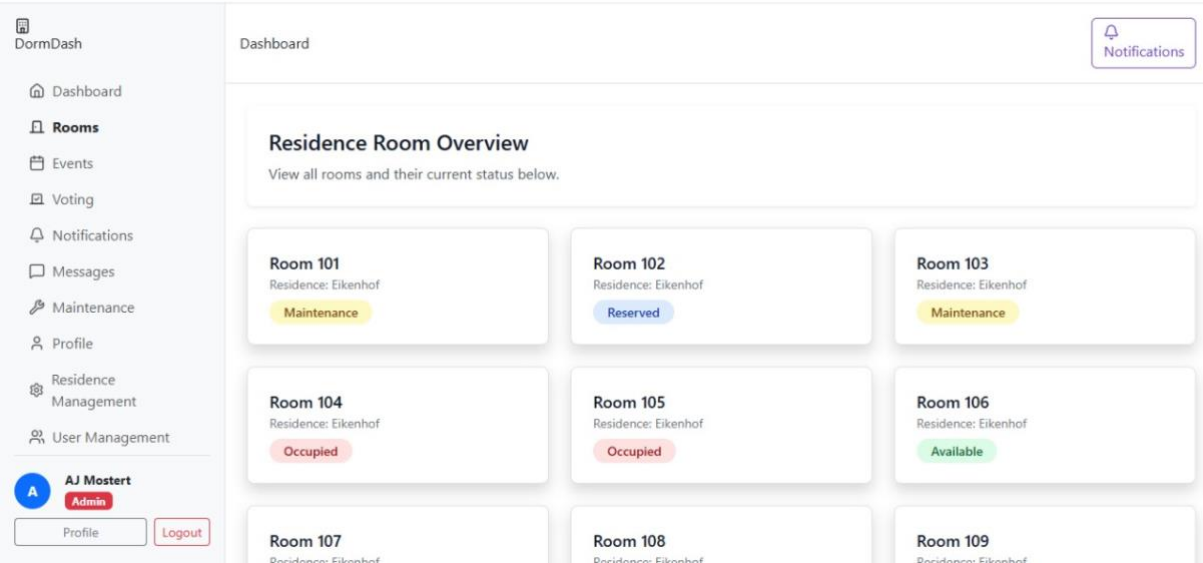


Figure 28: Room Management Page



Backend Structure

Architectural Overview

The backend of the DormDash system is built according to the Model–View–Controller (MVC) architectural pattern design pattern, which organizes application logic into three separate layers: models (for data and business rules), views (for presentation), and controllers (for request handling). The separation of concerns enables modularity, scalability, and maintainability, so that multiple developers can work on different parts of the system simultaneously without running into issues of overlap or dependency.

The MVC architecture lets code be reuse and improves testability two key aspects in the design of DormDash's backend. The selected backend framework, Laravel (PHP) enables a lot of productivity in the code, as it comes packaged with an interactive routing, ORM (Eloquent), authentication, and validation. All of these provide a level of operational sophistication with the database, user management, error management, all while being compliant with best practices in developing modern web applications.

From a systems engineering perspective, MVC promotes separation of responsibilities and information hiding, two principles that are emphasized in structured programming theory. These principles enable backend logic to be encapsulated, data manipulation to be abstracted, and changes to the user interface to occur without impacting the core logic of the system.

Route Layer

Route files serve as the main point of entry into the DormDash backend. Each route is an HTTP endpoint that maps to a controller function that will handle the client's request. The routes in Laravel are defined in the `web.php` or `api.php` file, depending on whether the request is for a web-based request or a RESTful API request.

In DormDash, the API route structure follows a RESTful convention, providing predictable and logically structured URLs. Common route groups include:

- **Authentication Routes:** Handle user login, registration, password resets, and token validation.
- **User Routes:** Manage CRUD operations for user profiles, roles, and permissions.
- **Residence and Room Routes:** Manage the accounting of rooms and tracking of residents.
- **Event and Voting Routes:** Enable engagement through event creation, voting polls, and response tracking.
- **Notification Routes:** Deliver real-time announcements and system alerts.
- **Maintenance Routes:** Facilitate maintenance request submissions and status monitoring.

Figure 29: Example of code from router file

```
// Authentication Routes
Route::post('/register', [AuthController::class, 'register']);
Route::post('/login', [AuthController::class, 'login']);

// Protected Routes (require authentication)
Route::middleware(['auth:sanctum'])->group(function () {
    Route::get('/user', [UserController::class, 'profile']);
    Route::post('/logout', [AuthController::class, 'logout']);
});

// Residence and Room Management
Route::get('/residences', [ResidenceController::class, 'index']);
Route::post('/rooms/allocate', [RoomController::class, 'allocateRoom']);

// Events and Voting
Route::post('/events', [EventController::class, 'create']);
Route::post('/votes', [VoteController::class, 'submitVote']);
});
```

To guarantee that only authorised users can access protected endpoints, these route definitions are modularly grouped and secured using middleware like AUTH: sanctum.

Controller Layer

Controllers represent the business logic backbone of the application. Each DormDash controller is scoped to a function area that follows the single-responsibility principle. Controllers process HTTP requests from the client, use models when appropriate, and return a JSON response back to the client.

Key controllers include:

- **AuthController:** Manages registration, login, logout, and session tokens. Incorporates hashing for password encryption and token-based authentication using Laravel Sanctum.
- **UserController:** Handles CRUD operations for users, role assignment, and access control.
- **ResidenceManagementController:** Coordinates room allocations, validates occupancy limits, and ensures data consistency.
- **EventController and VoteController:** Enable creation of residence events and voting polls, ensuring one-vote-per-user integrity through backend validation.
- **NotificationController:** Sends residence-wide messages and event notifications via Laravel's built-in broadcasting tools.
- **MaintenanceRequestController:** Allows users to log maintenance issues and track resolution progress.
- **AdminController:** Provides higher-level administrative functions, including system analytics and data oversight.

Figure 30: Example of code from the controller file

```
public function register(Request $request)
{
    $validated = $request->validate([
        'name' => 'required|string|max:255',
        'email' => 'required|string|email|unique:users',
        'password' => 'required|string|min:8',
    ]);

    $user = User::create([
        'name' => $validated['name'],
        'email' => $validated['email'],
        'password' => Hash::make($validated['password']),
    ]);

    $token = $user->createToken('auth_token')->plainTextToken;

    return response()->json([
        'user' => $user,
        'access_token' => $token,
        'token_type' => 'Bearer',
    ], 201);
}
```

To make sure that no erroneous data enters the system, controllers also use Laravel's Request classes for input validation. This greatly lowers the possibility of injection vulnerabilities or data corruption.

Model Layer

The model layer simplifies and oversees the data interactions that take place between the application and the database. The models for DormDash utilize Laravel's Eloquent ORM, which is a wrapper around a database that enables complex SQL to be mapped to object-oriented code that can be read and verified with more ease.

Each model corresponds to a table in the relational database, representing a core entity within the DormDash ecosystem.

These include:

- **User:** Stores user credentials, roles, and associated authentication tokens. Implements HasRoles and HasApiTokentraits.
- **Residence:** Represents a residence building, its name, and configuration.
- **Room:** Defines room attributes, capacity, and current occupancy.
- **Event & Vote:** Manage student activities, event creation, and voting logic.
- **Notification:** Stores and queues system announcements and messages.
- **MaintenanceRequest:** Tracks maintenance submissions and statuses.
- **Chatroom & Message:** Support internal communication features.
- **Role & Access:** Define hierarchical permissions to support role-based access control (RBAC).
- **Bed:** Associates individual beds with rooms, ensuring one-to-one occupancy mapping.

Figure 31: Example of model file

```
class Room extends Model
{
    protected $fillable = ['residence_id', 'room_number', 'capacity'];

    public function residence()
    {
        return $this->belongsTo(Residence::class);
    }

    public function occupants()
    {
        return $this->hasMany(User::class);
    }
}
```

The DormDash database guarantees relational consistency between users, rooms, and residences by implementing foreign key constraints to enforce referential integrity. This ensures data reliability and cuts down on redundancy.

Middleware and Security

Middleware components serve the purpose of filtering requests and controlling access. Authentication middleware (such as `auth:sanctum`), rate limiting middleware (the throttle middleware), and CORS middleware are commonly used examples. Middleware helps ensure unauthorized users cannot filter or manipulate protected resources.

Data security measures align with POPIA (Protection of Personal Information Act) compliance standards, incorporating:

- Encrypted password storage using **bcrypt**.
- HTTPS enforcement for all requests.
- Token-based session management.
- Data anonymization for sensitive information in logs.

Testing and Quality Assurance

Testing Overview

Testing was conducted throughout the development process and for both manual and automated testing approaches with the intention of confirming reliability, usability, and system integrity prior to deployment. Testing was also aligned with STLC best practices with validating the system in iterations through unit testing, integration testing, and user acceptance testing.

The testing process was divided into three main phases:

1. Developer Testing (Unit & Integration)
2. User Testing (Functional & Usability)
3. Quality Assurance and Review

Developer Testing

The goal of developer testing was to confirm that each module was correct. To ensure that inputs resulted in the desired outputs, unit testing was performed on each controller and model.

Tools and methods included:

- Postman for endpoint verification.
- Laravel's PHPUnit framework for automated test scripts.
- Mock data generation for testing database interactions without production data risks.

Examples of test cases:

| Module | Objective | Expected Outcome |
|------------------------------|---|--|
| AuthController | Validate login credentials and token generation | Secure tokens issued for valid users |
| RoomController | Verify room allocation prevents duplicates | No student can occupy multiple rooms |
| VoteController | Enforce single-vote-per-user policy | Duplicate votes rejected |
| NotificationController | Ensure timely message delivery | Push notifications sent within 2 seconds |
| MaintenanceRequestController | Track issue status updates | State changes logged correctly |

Test results confirmed full endpoint functionality with minimal errors. Detected issues, such as duplicate room assignments, were resolved by adding validation constraints in the backend.

User Testing

User testing was performed with resident students and administrative staff to assess usability in a real-world context. The participants were asked to perform key tasks in the system, including logging in, assigning a room, submitting a maintenance request, and making votes for events.

Feedback indicated that:

- Authentication and navigation were intuitive.
- Room allocation logic worked as expected after validation fixes.
- Notifications were timely, improving overall communication.
- The voting feature was highly engaging and accurate.

Updates implemented post-testing:

- Improved password recovery flow for ease of access.
- Added backend validation to prevent duplicate bookings.
- Introduced front-end success messages after voting.
- Optimized database queries to enhance load times.

Figure 32: Example of a code after feedback from user testing

```
public function test_user_can_login()
{
    $user = User::factory()->create([
        'password' => bcrypt($password = 'secret123'),
    ]);

    $response = $this->postJson('/api/login', [
        'email' => $user->email,
        'password' => $password,
    ]);

    $response->assertStatus(200)
        ->assertJsonStructure(['access_token']);
}
```

Quality Assurance Process

After each testing phase, a structured Quality Assurance (QA) review ensured compliance with functional and non-functional requirements. QA focused on four core dimensions:

1. **Functional Accuracy:** Verification of expected system behaviour across all endpoints.
2. **Performance Testing:** Load tests simulated multiple concurrent users to ensure stability and response times below 300ms.
3. **Security Auditing:** Checked for injection vulnerabilities, session hijacking, and unauthorized access attempts.
4. **Regression Testing:** Re-tested existing features after bug fixes to ensure stability and backward compatibility.

Security Measures

The system uses several methods and practices to ensure that those who need access to the system, do in fact have access to it as well as the information that they are to be privy to. The in-house API implements strict Role-Based Access Control (RBAC) and uses Laravel Sanctum for authentication.

In terms of the users themselves, they are tasked with creating a username and password that the system will use to authenticate their ability to access the system and determine their role.

These methods include :

- Each request must include an authorization token.
- Sensitive operations (room allocation, event creation, etc.) are protected by middleware that verifies user roles before access is granted.
- All data transmitted between frontend and backend is encrypted using HTTPS.
- The system complies with POPIA (Protection of Personal Information Act) standards to ensure privacy of student and staff data.

Appendices

Appendix A: Environment Variables Reference

This appendix lists key environment variables required for local and production deployment. Each variable must be adapted to the target environment for optimal security and functionality.

| Variable | Purpose | Example Value |
|-----------------|------------------------|---|
| APPNAME | Application name | DormDash |
| APPURL | Base application URL | http://localhost:8000 |
| DBCONNECTION | Database driver | mysql |
| DBHOST | Database host | localhost |
| DBPORT | Database port | 3306 |
| DBDATABASE | Database name | dormdash |
| DBUSERNAME | Database username | root |
| DBPASSWORD | Database password | [secure] |
| MAILMAILER | Mail driver | log |
| BROADCASTDRIVER | Broadcasting driver | pusher |
| PUSHERAPPID | Pusher App ID | [yourappid] |
| PUSHERAPPKEY | Pusher App Key | [yourappkey] |
| PUSHERAPPSECRET | Pusher App Secret | [yourappsecret] |
| SESSIONDRIVER | Session storage method | database |
| REDISCLIENT | Redis client driver | phpredis |

Appendix B: API Endpoint Catalogue

This appendix summarizes major API routes for the DormDash Resident Management Platform. Endpoints are grouped by function and indicate whether authentication is required.

Authentication and User Routes

- POST /login — Authenticate user, return token
- POST /logout — Terminate session (auth required)
- GET /user — Fetch current user info (auth required)
- User management (GET /user, POST /user, PUT /user/{id}, DELETE /user/{id})

Residence & Room Management

- Residence CRUD: GET/POST/PUT/DELETE /residences, /residences/{id}
- Room CRUD: GET/POST/PUT/DELETE /rooms, /rooms/{id}

Event & Voting

- Event CRUD: GET/POST/PUT/DELETE /events, /events/{id}
- Voting: GET/POST/PUT/DELETE /votes, /votes/{id}, POST /votes/submit-response

Notifications

- Notification CRUD: GET/POST/PUT/DELETE /notifications, /notifications/{id}
- Recent announcements: GET /notifications/recent

Maintenance Requests

- CRUD: GET/POST/PUT/DELETE /maintenancerequests, /maintenancerequests/{id}
- Count: GET /maintenancerequestspendingRequestcount

Chat and Messaging

- Chatroom CRUD: GET/POST/PUT/DELETE /chatrooms, /chatrooms/{id}
- Message CRUD: GET/POST/PUT/DELETE /messages, /messages/{id}

Appendix C: Typical Test Cases and Quality Assurance

A summary of typical automated and manual test cases performed during development and QA. Each test is linked to a module and outcome:

| Module | Objective | Expected Outcome |
|------------------------------|------------------------------------|---|
| AuthController | Validate login, token generation | Secure tokens issued for valid users |
| RoomController | Prevent duplicate room allocation | No student can occupy multiple rooms |
| VoteController | Enforce single vote per user | Duplicate votes rejected |
| NotificationController | Deliver push notifications quickly | Notification delivered within 2 seconds |
| MaintenanceRequestController | Track maintenance status | Log state changes correctly |

Appendix D: Glossary of Terms

- RBAC: Role-Based Access Control, a security method to restrict system access.
- MVC: Model–View–Controller, an architectural pattern for code structure.
- MVP: Minimum Viable Product, a version with core features for deployment.
- POPIA: Protection of Personal Information Act (South Africa data security law).
- SPA: Single Page Application, web application or site that fits on one web page for a fluid user experience.
- ORM: Object-Relational Mapping, a way to interact with databases using objects.

Appendix E: Directory Structure Snapshot

Snapshot of directory and file structures for reference and onboarding:

- Laravel Backend
 - app/Http/Controllers/
 - database/migrations/
 - routes/web.php, routes/api.php
- React Frontend
 - resources/js/components/
 - resources/js/pages/
 - resources/js/layouts/
 - resources/js/hooks/
 - resources/js/types/
 - app.tsx (Main entry point)