Realisation document: Pass The Ball

By: Simeon Markov

Institution Name: Fontys UAS

Course/Class: ICT/EN08

Date: 2025-10-09

Introduction

This document highlights the project development phase, detailing the technical structure, implementation strategy and development workflow. It includes explanation of the core logic, explaining major parts of the architecture like controllers, views, components, writing mysql queries and dealing with migrations (database changes, modifications).

Project structure

This section presents the system architecture and main components.

System architecture:

Architecture: Inertia.js hybrid SPA (no separate API, no client-side routing)

- **Backend:** Laravel handles routing, controllers, models, validation
- **Frontend:** Vue 3 SFCs with Composition API + TypeScript
- **Bridge:** Inertia.js manages page transitions without full page reloads

Directory structure:

```
pass_the_ball/
│
├── app/                            # Laravel Backend (PHP)
│   ├── Console/                    # Artisan CLI commands
│   ├── Enums/                      # Type-safe constants (notification types, roles)
│   ├── Events/                     # Broadcasting events for real-time updates
│   ├── Http/
│   │   ├── Controllers/            # Route handlers returning Inertia responses
│   │   ├── Middleware/             # Request filters (auth, Inertia shared data)
│   │   ├── Requests/               # FormRequest validation classes
│   │   └── Resources/              # API transformers (Eloquent → JSON)
│   ├── Models/                     # Eloquent ORM models (User, Post, Comment, Group, etc.)
│   ├── Notifications/              # Email/database/broadcast notification classes
│   ├── Policies/                   # Authorization logic (user permissions)
│   ├── Providers/                  # Service provider bootstrapping
│   └── Services/                   # Business logic (image optimization, AI enhancement)
│
├── bootstrap/                      # Application initialization
│   ├── app.php                     # Creates Laravel instance
│   ├── providers.php               # Registers service providers
│   └── cache/                      # Cached config/routes (auto-generated)
│
├── config/                         # Configuration files (.env values)
│   ├── app.php                     # Core settings (timezone, locale, debug)
│   ├── auth.php                    # Authentication guards
│   ├── broadcasting.php            # Pusher/Echo WebSocket config
│   ├── database.php                # Database connections (SQLite default)
│   ├── filesystems.php             # Storage disks (local, S3, public)
│   ├── fortify.php                 # Authentication features (2FA, password reset)
│   ├── inertia.php                 # Inertia.js server config
│   ├── openai.php                  # OpenAI API for AI features
│   ├── purifier.php                # HTML sanitization (XSS protection)
│   ├── queue.php                   # Queue drivers for background jobs
│   └── services.php                # Third-party service credentials
│
├── database/
│   ├── migrations/                 # Version-controlled database schema
│   ├── factories/                  # Faker factories for test data
│   └── seeders/                  # Database seeders for initial/demo data
│
├── docs/                         # Technical documentation (markdown guides)
│   ├── FLASH_MESSAGES_FLOW.md
│   ├── GROUPS_FEATURE_GUIDE.md
│   ├── NOTIFICATIONS_SYSTEM.md
│   ├── PHOTO_GALLERY_FEATURE.md
│   ├── POLYMORPHIC_REACTIONS.md
│   └── ... (20+ feature/architecture docs)
│
├── public/                       # Web root (publicly accessible)
│   ├── index.php                 # Entry point for all HTTP requests
│   ├── build/                    # Compiled Vite assets (JS/CSS bundles)
│   ├── images/                   # Static image assets
│   └── storage/                  # Symlink to /storage/app/public (user uploads)
│
├── resources/
│   ├── css/                      # Global CSS/Tailwind entry points
│   ├── js/                       # Vue 3 Frontend (TypeScript)
│   │   ├── actions/              # Reusable Inertia form actions
│   │   ├── components/
│   │   │   ├── ui/               # Reka UI primitives (Button, Dialog, Input)
│   │   │   ├── app/              # Feature components (CreatePost, PostList)
│   │   │   └── groups/           # Group-specific components
│   │   ├── composables/          # Reusable composition functions (useFlashMessage)
│   │   ├── layouts/              # Page layouts (AppLayout, AuthLayout)
│   │   ├── lib/                  # Utility libraries (cn() for class merging)
│   │   ├── pages/                # Inertia page components (Dashboard, Profile)
│   │   ├── routes/               # Auto-generated TypeScript route helpers (Wayfinder)
│   │   ├── types/                # TypeScript type definitions (User, Post, etc.)
│   │   ├── app.ts                # Inertia app initialization
│   │   ├── bootstrap.ts          # Axios/Laravel Echo setup
│   │   ├── echo.ts               # WebSocket configuration
│   │   └── ssr.ts                # Server-side rendering entry
│   └── views/
│       └── app.blade.php         # Root HTML template (mounts Vue app)
│
├── routes/                       # Laravel route definitions
│   ├── web.php                   # Main app routes (posts, groups, profiles)
│   ├── auth.php                  # Authentication routes (Fortify)
│   ├── settings.php              # User settings routes
│   └── console.php               # Artisan command routes
│
├── storage/                      # Private file storage
│   ├── app/
│   │   └── public/               # User uploads (symlinked to /public/storage)
│   ├── framework/                # Cache, sessions, compiled views
│   └── logs/                     # Application logs (daily rotation)
│
├── tests/                        # Pest PHP test suite
│   ├── Feature/                  # End-to-end tests (HTTP, database)
│   ├── Unit/                     # Isolated unit tests (models, services)
│   ├── Pest.php                  # Pest configuration
│   └── TestCase.php              # Base test class
│
├── vendor/                       # Composer dependencies (gitignored)
│
```

```
├── composer.json                # PHP dependencies & autoloading
├── package.json                 # Node.js dependencies & scripts
├── vite.config.ts               # Vite build configuration
├── tsconfig.json                # TypeScript compiler options
├── eslint.config.js             # ESLint code quality rules
├── components.json              # Reka UI component library config
├── phpunit.xml                  # Pest PHP test configuration
└── .env                         # Environment variables (not in repo)
```

Key technologies

Layer	Technology	Purpose
Backend	Laravel 12	MVC framework, routing, ORM, authentication
Frontend	Vue 3 + TypeScript	Reactive UI with Composition API
Bridge	Inertia.js 2	SPA experience without client-side routing
Styling	Tailwind CSS 4	Utility-first CSS framework
UI Components	Reka UI, HeadlessUI	Radix Vue wrapper components
Database	MariaDB	Relational database
Real-time	Laravel Echo + Pusher	WebSocket broadcasting
Build Tool	Vite 7	Fast frontend asset bundling
Testing	Pest PHP	Modern PHP testing framework
Image Processing	Intervention Image	Image optimization/resizing
Rich Text	CKEditor 5	WYSIWYG editor with HTML sanitization
AI	OpenAI SDK	AI-powered post enhancements

Table 1: Key technologies used

Design vs realisation

UI Additions/Alterations

- *Settings/Profile:* A profile page view for the users was added, where users could preview others profiles and edit their own profiles.

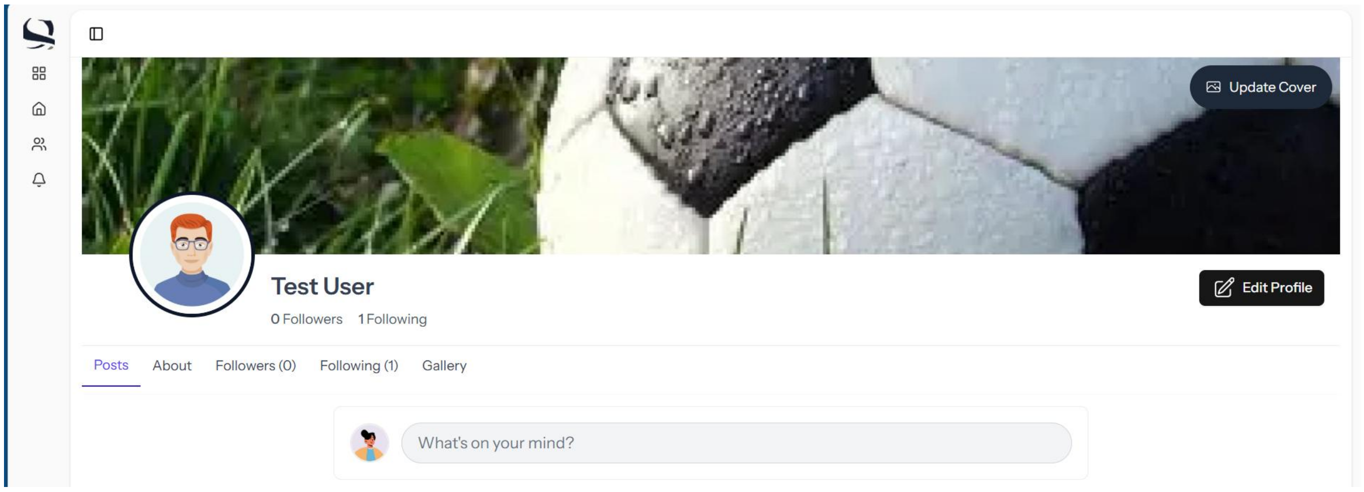


Figure 1: Profile page

The page contains internal links to post, about, gallery and other pages (visibility: public).

- *Groups:* A group page view for the users to join in was added. When users try to join a group, an request is being send to the admin of that group to accept requests.

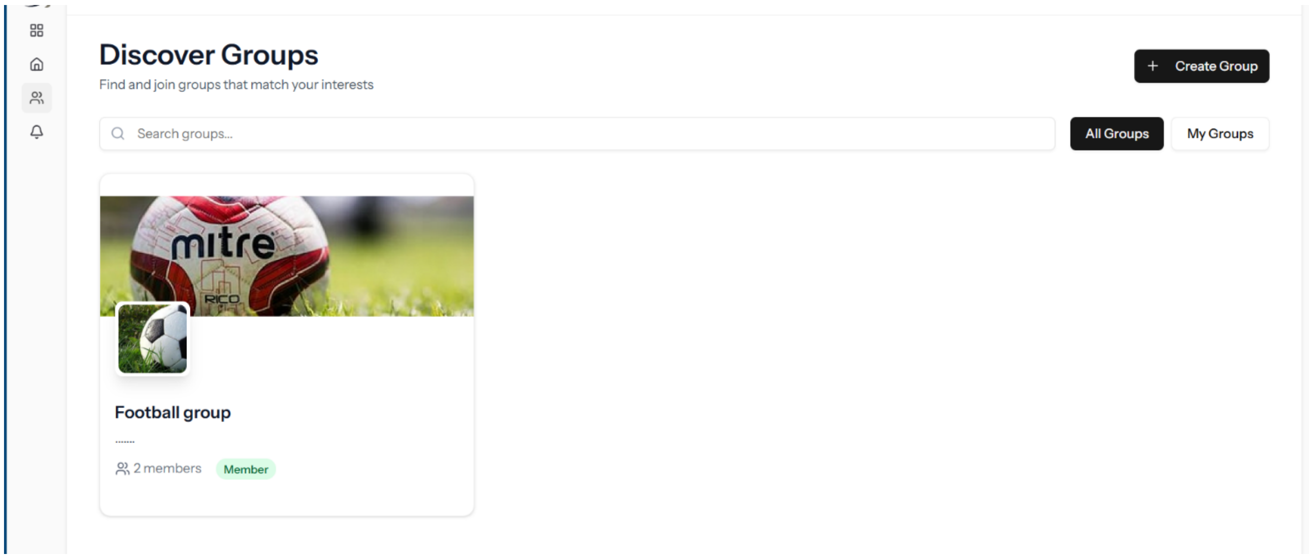
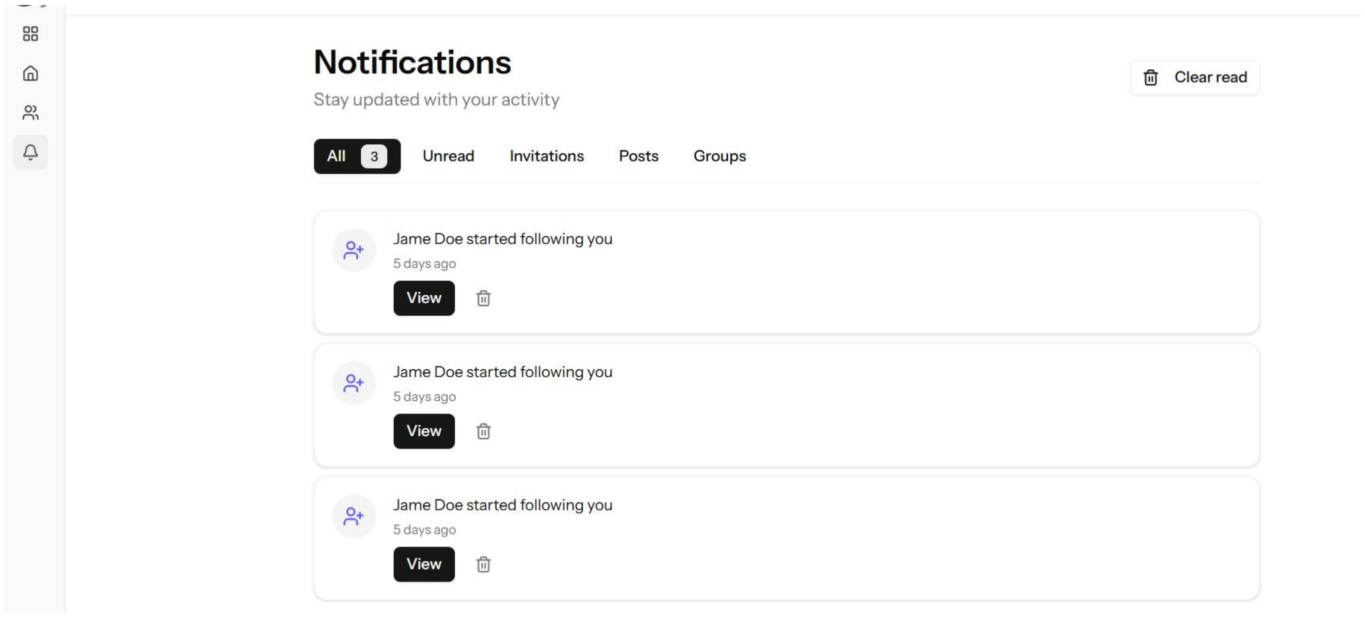


Figure 2: Group page

Each authorized user could create a group (the owner of the group is the administrator of that group, full rights).

- *Notifications:* A notification page was created, where users receive any kind of notifications (following, group joining, etc.).


Notifications

Stay updated with your activity

Jame Doe started following you

5 days ago

Jame Doe started following you

5 days ago

Jame Doe started following you

5 days ago

Figure 3

Database

- Several tables were added to the original database schema, including photos, photo_tags, albums, notifications, group_invitations.

Tables description *'albums':*

Column	*Type*	*Constraint*	*Description*
Id	*BIGINT*	*PRIMARY KEY*	*Unique identifier for each album.*
user_id	*BIGINT*	*NOT NULL*	*Stores album's user's id.*
slug	*VARCHAR*	*NOT NULL,* UNIQUE	*Used for improving URL readability and search optimization.*
description	*TEXT*	*NULLABLE*	*Stores album's description (optional).*
visibility	*ENUM*	*DEFAULT=public*	*Stores visibility enumeration of pre-defined values.*

cover_path	*VARCHAR*	*NULLABLE*	*Stores the URL of the album’s cover image.*
deleted_by	*BIGINT*	*FOREIGN KEY*	*Stores the user who deleted the album.*
deleted_at	*DATETIME*	*NOT NULL*	*Used for storing the date when the album was deleted.*
created_at	*TIMESTAMP*	*NOT NULL*	*Stores data about when the album was created.*
updated_at	*TIMESTAMP*	*NOT NULL*	*Stores data about when the user made updates on his/her album.*

Table 1: Album table

‘photos”:

Column	Type	Constraint	Description
Id	BIGINT	PRIMARY KEY	Unique identifier for each image in album.
album_id	BIGINT	FOREIGN KEY	Foreign key for album’s photos.
title	VARCHAR	NULLABLE	Stores the title (if any) of each image.
slug	VARCHAR	NOT NULL , UNIQUE	Stores the slug for each image.

description	TEXT	NULLABLE	Stores the description (if any) of each image.
file_path	VARCHAR	NOT NULL	Stores the path of the optimized image.
original_file_path	VARCHAR	NULLABLE	Stores the original, uncompressed path of the image.
thumbnail_path	VARCHAR	NULLABLE	Stores the thumbnail path of each image.
medium_path	VARCHAR	NULLABLE	Stores the medium size path.
mime_type	VARCHAR	NOT NULL	Stores the mime type of each image.
size	BIGINT	NULLABLE	Stores the size in bytes.
width	INT UNSIGNED	NULLABLE	Stores the width of the image.
height	INT UNSIGNED	NULLABLE	Stores the height of the image.
views_count	BIGINT UNSIGNED	NOT NULL	Stores each image’s view count.

downloads_count	BIGINT UNSIGNED	NOT NULL	Stores each image’s download count.
metadata	JSON	NULLABLE	Stores the metadata for the images.
deleted_by	BIGINT	FOREIGN KEY, NULLABLE	Stores the user who deleted the image/images.
deleted_at	DATETIME	NULLABLE	Stores the date when the image was deleted.
created_at	DATETIME	NOT NULL	Stores the date when the image was uploaded to the album.
updated_at	DATETIME	NOT NULL	Stores the date when the image was updated.

Table 2: Photos table

‘photo_tags:

Column	Type	Constraint	Description
Id	BIGINT	PRIMARY KEY	Unique identifier for each image tag.
name	VARCHAR	NOT NULL	Stores the name of the name of the tag.
slug	VARCHAR	UNIQUE, NOT NULL	URL unique identifier for each tag.
user_id	BIGINT	FOREIGN KEY	Foreign key pointing to particular user.
created_at	TIMESTAMP	NOT NULL	Stores data about when a tag is created.

updated_at	TIMESTAMP	NOT NULL	Stores data about when a tag is updated.

Table 3: Tags table

‘notifications’:

Column	Type	Constraint	Description
Id	CHAR	PRIMARY KEY	Unique identifier for each notification.
type	VARCHAR	NOT NULL	Specifies the type of the notification (e.g. group invitation).
notifiable	VARCHAR	NOT NULL	Specifies the type of the model that is being ‘notified’ within the morph relationship.
data	VARCHAR	NOT NULL	Stores the notification message.
read_at	DATETIME	NULLABLE	Stores data about when a notification is read.
created_at	DATETIME	NOT NULL	Stores data about when a notification is sent.
updated_at	DATETIME	NOT NULL	Stores data about when a notification is updated.

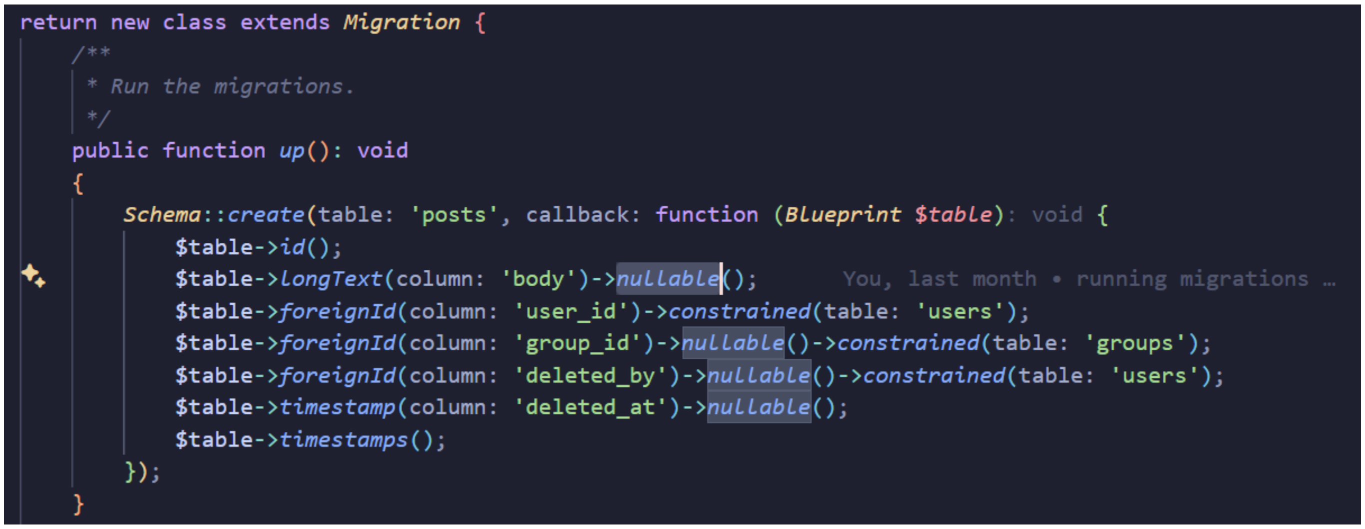
Table 4: Notification table

Code documentation

Main logic

This section explains the main logic behind the project (MVC three-tier architecture).

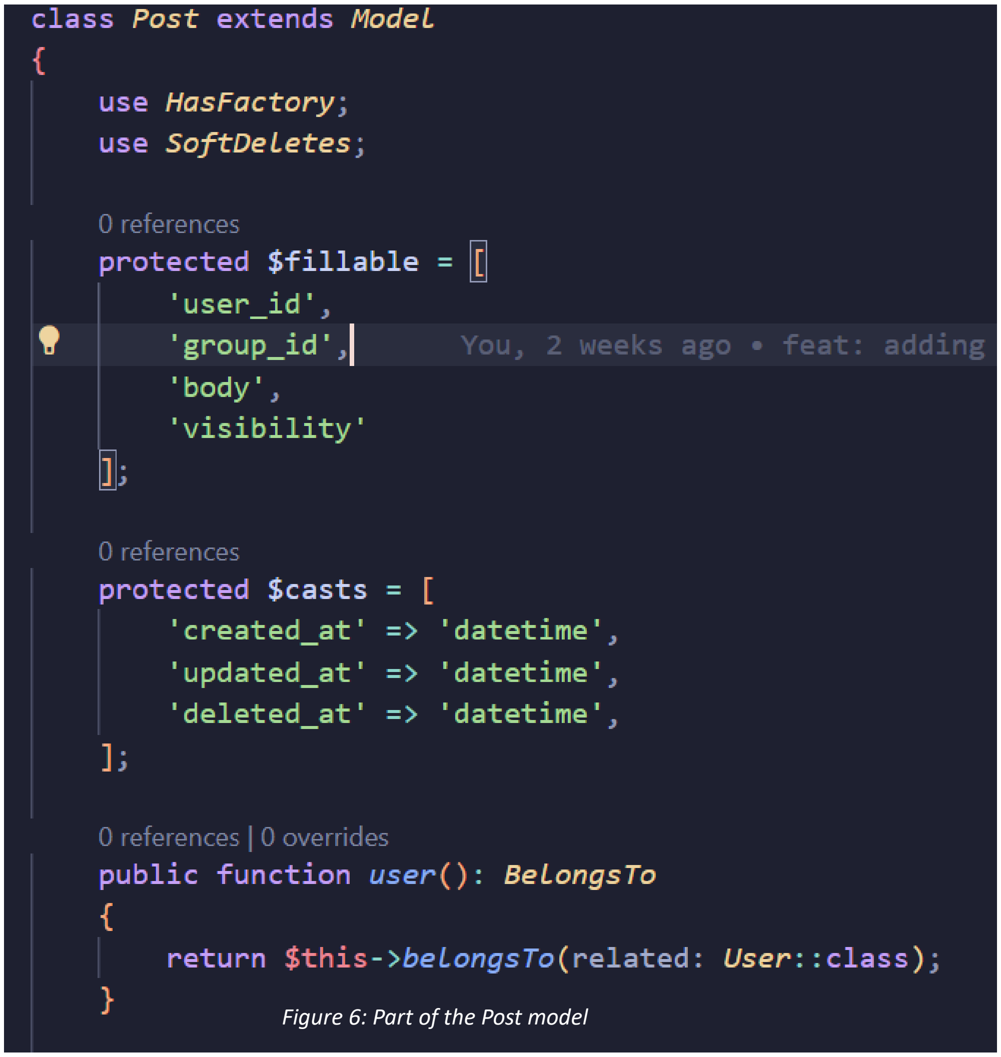
<u>*Example flow:*</u> First a model with a migration file is created, where the migration file is used to define the model objects, so that the ORM could pass it as data object (code-first approach) and the model prepares the data that the application is going to work with (business layer). Then the view reads from the model, handles user’s requests, passes data to the defined controller, which then returns a response to the view.



```
return new class extends Migration {
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::create(table: 'posts', callback: function (Blueprint $table): void {
            $table->id();
            $table->longText(column: 'body')->nullable();       You, last month • running migrations …
            $table->foreignId(column: 'user_id')->constrained(table: 'users');
            $table->foreignId(column: 'group_id')->nullable()->constrained(table: 'groups');
            $table->foreignId(column: 'deleted_by')->nullable()->constrained(table: 'users');
            $table->timestamp(column: 'deleted_at')->nullable();
            $table->timestamps();
        });
    }
```

Figure 5: Posts migrations

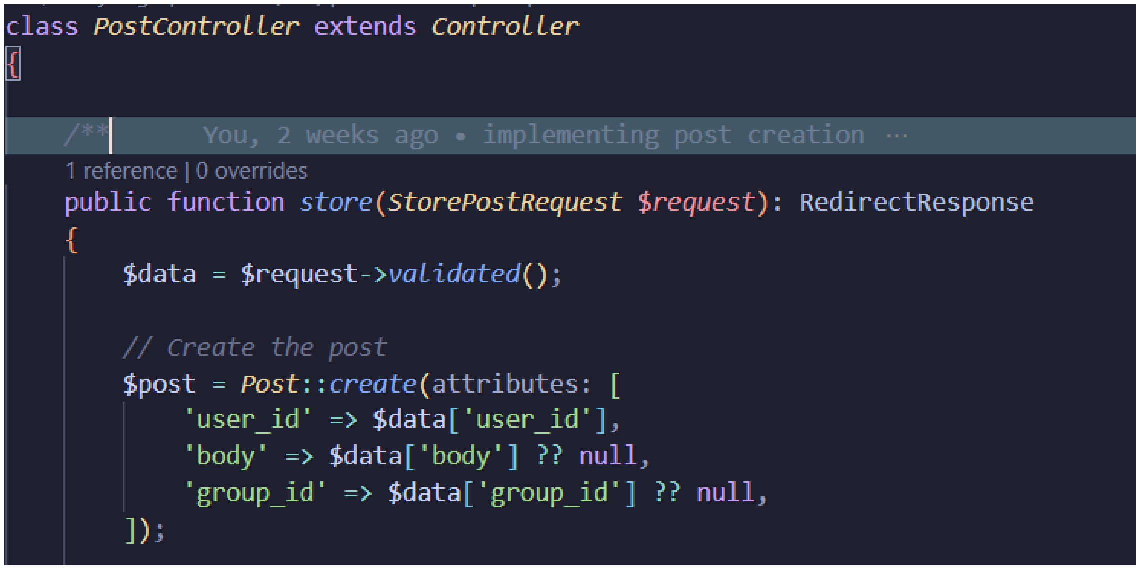
In the **up()** method are defined all the fields of the table with the constraints and relations. After the migration file runs, the ORM maps it to a database table.



```
class Post extends Model
{
    use HasFactory;
    use SoftDeletes;

    0 references
    protected $fillable = [
        'user_id',
        'group_id',        You, 2 weeks ago • feat: adding
        'body',
        'visibility'
    ];

    0 references
    protected $casts = [
        'created_at' => 'datetime',
        'updated_at' => 'datetime',
        'deleted_at' => 'datetime',
    ];

    0 references | 0 overrides
    public function user(): BelongsTo
    {
        return $this->belongsTo(related: User::class);
    }
```

Figure 6: Part of the Post model

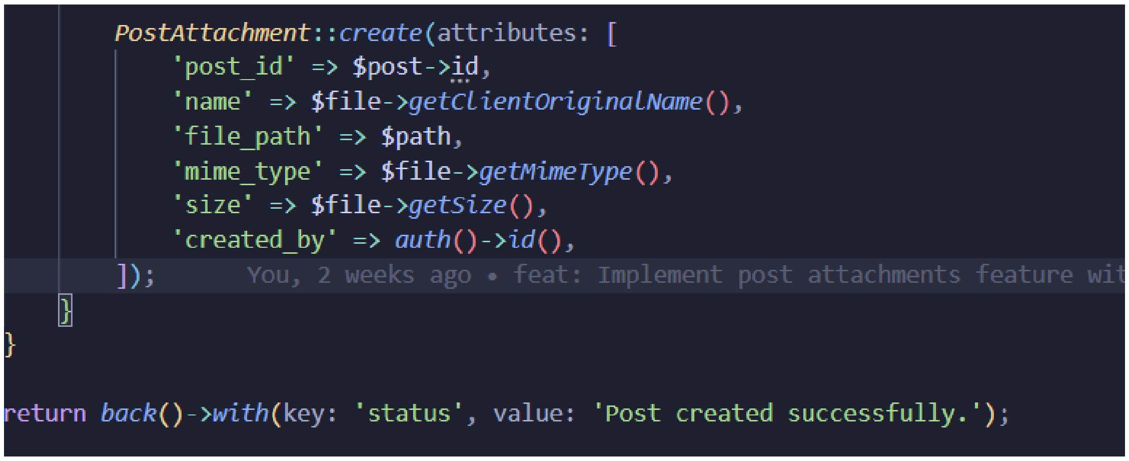
In the model are defined the properties (see **fillable** variable) and the relations of the model, for example see the **user()** method – first it declares the name of the relation (in this case the relationship is with User model), then is specified the type of the relationship (one-to-one, many-to-one, etc…) and at the end it returns the relationship. In php this is done through implementing the return type (**BelongsTo**) and then on the instance of the model (keyword **$this**) is called the method of that class (in this case the method **belongsTo()** does the job for the one-to-many relationship, which in this context means that one user could have multiple posts).



```
class PostController extends Controller
{

    /**        You, 2 weeks ago • implementing post creation ···
    1 reference | 0 overrides
    public function store(StorePostRequest $request): RedirectResponse
    {
        $data = $request->validated();

        // Create the post
        $post = Post::create(attributes: [
            'user_id' => $data['user_id'],
            'body' => $data['body'] ?? null,
            'group_id' => $data['group_id'] ?? null,
        ]);
```

Figure 7: Part of Post controller (part 01 of store method)

First there is defined a function called **store()** that takes a custom Post request (**StorePostRequest:** used for data sanitization). Then an instance of Post object is created, where the php function from the ORM is called (**create()**), as the parameters are passed to it like the user of that post (**user_id**) and the content (**body**).



```
            PostAttachment::create(attributes: [
                'post_id' => $post->id,
                'name' => $file->getClientOriginalName(),
                'file_path' => $path,
                'mime_type' => $file->getMimeType(),
                'size' => $file->getSize(),
                'created_by' => auth()->id(),
            ]);       You, 2 weeks ago • feat: Implement post attachments feature wit
        }
}

return back()->with(key: 'status', value: 'Post created successfully.');
```

Figure 8: Part of Post controller (part 02 of store method)

Then the file attachments are handled (if any). The class is called statically, its method is filled with the required parameters, where a validation on the type of the image is allowed, size validation and the file is given a unique name, which consists of a timestamp and unique id (chars) appended to the original file name. The field **'created_by'** holds the authenticated user of the post. After all, a response is returned with a flash message, which is being displayed to the user (after successful post creation the user is redirected back to the same instance of page).

NOTE: For extensive reading of the code documentation please visit: https://github.com/Simeon31/Pass-The-Ball/tree/master.

Development methodology and techniques

Architectural Patterns & Principles

Category	**Patterns/Principles**
Patterns	Layered architecture, Domain-Driven Design-inspired aggregates, Repository & Adapter patterns at boundaries, CQRS-lite for read-heavy timelines, Event-driven async processing for notifications/media
Principles	Separation of concerns, Single Responsibility, Dependency Inversion (services depend on contracts), Security by Design, Modularity with explicit interfaces, Fail-fast error handling, Observability-first mindset
UI Patterns	UI components, Composition over inheritance, Conventional slots/props for extendability

Table 2: Architectural Patterns & Principles

Conclusion

In conclusion, the project successfully implements a scalable, maintainable social platform with modern web technologies and architectural best practices, delivering a responsive user experience with real-time features and extensibility through modular design.

References/Sources

- AI Transparency: Perplexity AI for extensive research, summary & Claude AI as code helper (used for generating partial code of the overall project with set instructions).
- YouTube video, inspired by: https://youtu.be/4iiEyOKhvao?si=vQARG5ZIs5uFc_jl .