IT110 – WEB SYSTEMS AND TECHNOLOGIES

FINAL PROJECT REPORT

Group & Section: 4 – CG1

Members: Alvarez

Berdon

Cabase

Valeroso

Git Repository: https://github.com/trbyte/studio-ghibli-app.git

Ghibli Filmography

The **Studio Ghibli Filmography** is a dynamic Single Page Application designed to transform raw data from the Ghibli API into an engaging visual narrative. Rather than presenting the films as a static archive, the web app employs a timeline-based storytelling approach that chronologically portrays the legacy of Studio Ghibli, starting from Castle in the Sky (1986) to the present day.

Project Objective

The primary objective of this project is to create an interactive platform that allows users to track their personal viewing progress while exploring Studio Ghibli's filmography through time, and add a note depicting their thoughts regarding the film. The application serves both as a guide and a personal film journal by combining data visualization with user interactivity.

Theme

We chose to envision a timeline as our primary narrative device to honor Studio Ghibli's rich history. This chronological approach allows the users to witness how the studio's films evolved across nearly four decades of film-making.

Bridging Technology

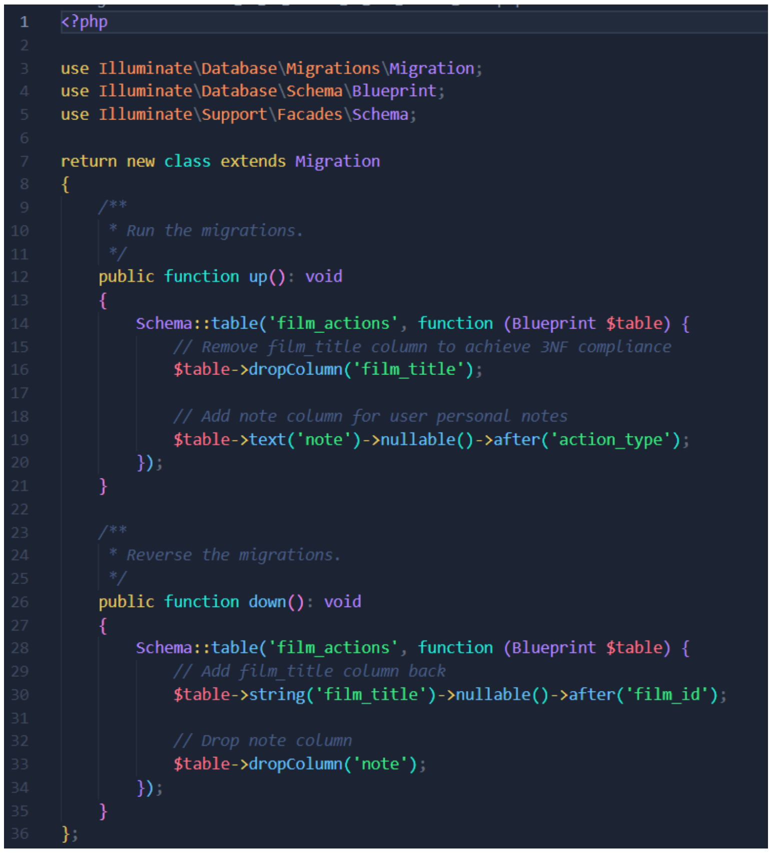
To achieve this vision, we utilized a modern hybrid architecture using Laravel (backend framework), Inertia.js (seamless bridge), and React (frontend framework). This tech stack combines the strengths of traditional server-side applications such as strong routing, security, and SEO optimization with the seamless interactivity and dynamic rendering capabilities of client-side Single Page Applications. The result is a fast, responsive user experience without sacrificing the architectural benefits of server-rendered applications.

Technical Implementation & Challenges

1. **Data Consistency:** One of the biggest technical hurdles we face was balancing normalization and speed, designing the database for the film tracking feature. Originally, we stored both the film_id and film_title in our local film_actions table so it would be easier for us to display the user's list. However, it came to our attention that this violated the Third Normal Form (3NF) by having a transitive depency because the film title is technically dependent on the external API, not just our primary key. This created a redundancy issue due to the fact that if the API changed a title, our database would be outdated.

   In order to fix this without compromising speed, we used a two-step solution:

   - **Strict Normalization:** We removed the film_title column from our database, storing only the film,_id as the foreign key reference.

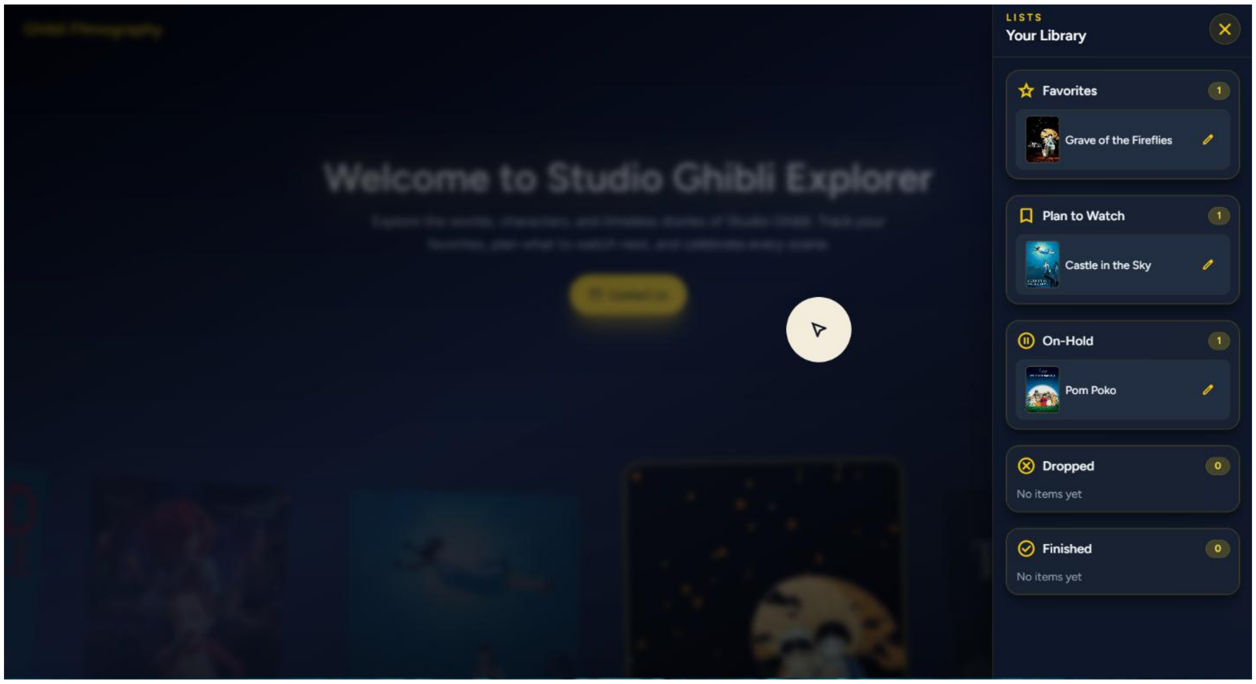


```php
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::table('film_actions', function (Blueprint $table) {
            // Remove film_title column to achieve 3NF compliance
            $table->dropColumn('film_title');

            // Add note column for user personal notes
            $table->text('note')->nullable()->after('action_type');
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::table('film_actions', function (Blueprint $table) {
            // Add film_title column back
            $table->string('film_title')->nullable()->after('film_id');

            // Drop note column
            $table->dropColumn('note');
        });
    }
};
```

Database Schema showing the normalized film_actions table.

- **Runtime Merging:** Since we still needed to show film titles to the user, we wrote logic in the FilmActionController that fetches the user's list and the cached API data separately. We then merged these two datasets in memory using PHP's map function. This allows us to maintain a strict 3NF database without compromising the app's speed with multiple API calls.

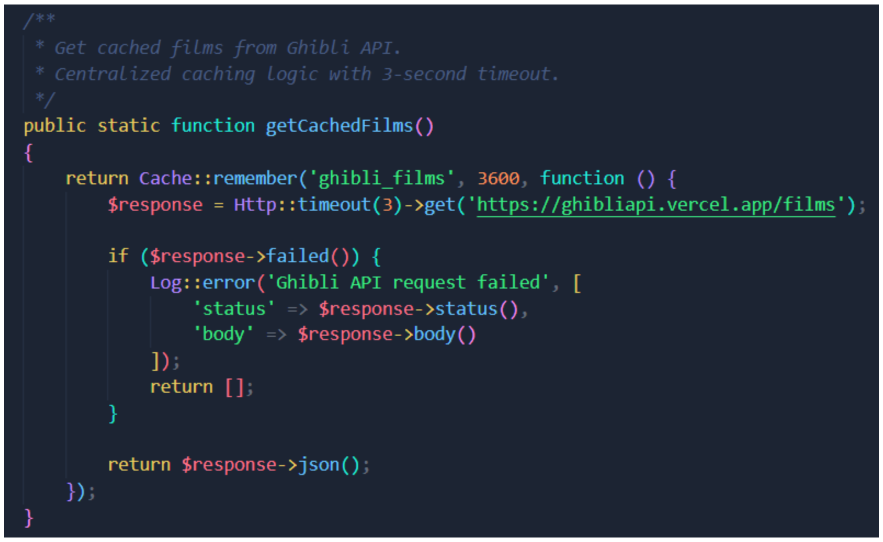
2. **Performance Optimization:** During our initial testing, we noticed a critical flaw in the performance where the application would "freeze" for 1-3 seconds whenever we navigated between pages. We came to know that this was due to our HandleInertiaRequests middleware trying to fetch Ghibli API data on every single request to populate the sidebar.



The "Your List" Sidebar, populated by the optimized controller.

To solve this, we refactored our architecture to stop it from blocking the user:

- **Centralized Caching:** We relocated the fetching logic out of the middleware and into a static method in GhibliController. We used Laravel's Cache::remember() to store the API response for one hour.



```
/**
 * Get cached films from Ghibli API.
 * Centralized caching logic with 3-second timeout.
 */
public static function getCachedFilms()
{
    return Cache::remember('ghibli_films', 3600, function () {
        $response = Http::timeout(3)->get('https://ghibliapi.vercel.app/films');

        if ($response->failed()) {
            Log::error('Ghibli API request failed', [
                'status' => $response->status(),
                'body' => $response->body()
            ]);
            return [];
        }

        return $response->json();
    });
}
```

GhibliController implementation showing Caching and Timeout logic.

- **Non-blocking Logic:** Now, the app only checks the cache when necessary such as on the Timeline or User List pages instead of on every page load.

- **Safety Timeouts:** Additionally, we added a 3-second HTTP timeout. This ensures that even if the Ghibli API goes down, our web app will still load at high speed for the user instead of pending permanently.

3. **State Management:** For the note feature, we wanted users to be able to add and update their thoughts on a film without having to reload the entire page. Standard HTML forms would have caused a full refresh.

   We worked this out by using Inertia.js and secure routing:

   - **Secure PATCH Route:** We set up a specific PATCH /film-actions/{id} route in Laravel. This adheres to the correct HTTP standards for partial updates and ensures we can validate that the user truly owns the note before saving it.
   - **Inertia Form Handling:** On the frontend, we used useForm hook. This enabled us to handle the submission asynchronously. Although the page doesn’t reload, the data is securely sent to the database, and the UI updates instantly to reflect the changes.