

PROYECTO DE CICLO DE G.S. DESARROLLO DE APLICACIONES WEB
TÍTULO DEL PROYECTO: AGATHA



Autora: Ana María Moya Fernández

Fecha: 15 de diciembre de 2025

1. Datos del proyecto

Agatha es una aplicación web diseñada para fomentar la escritura creativa mediante retos diarios. Cada vez que un usuario quiere escribir, recibe:

- Una palabra aleatoria.
 - Un lugar aleatorio.
- Debe incluir ambos elementos en su historia antes de poder guardarla.

1.1. Descripción general del proyecto

La app permite:

Crear historias; editarlas; listarlas; eliminarlas; revisarlas en cualquier momento.

Cuenta además con:

- Sistema seguro de autenticación.
- API REST construida en Laravel.
- Interfaz en Vue 3 + Tailwind.
- Recuento en el back de palabras.
- Sistema de avisos por correo cuando el usuario lleva días sin escribir.

1.2. Objetivos

1.2.1. Objetivos del proyecto

- Crear una API REST funcional con Laravel.
- Implementar un sistema de autenticación seguro con Sanctum.
- Gestionar historias con operaciones CRUD.
- Validar que las historias incluyan la palabra y el lugar generados.
- Implementar un sistema de retos aleatorios.
- Construir un frontend usable en Vue 3.
- Añadir un sistema automático de avisos por inactividad.
- Mantener un código modular y escalable.

1.2.2 Objetivos descartados o no implementados

- Panel de administración de palabras.
- Publicación pública de historias.
- Sistema de comentarios o interacciones.
- Estadísticas avanzadas.

1.3. Materiales y recursos utilizados

- Portátil personal (Windows).
- Servidor local mediante Laragon, aunque en un principio use xampp.
- Laravel 12
- PHP 8.3
- MySQL
- Vue 3 + Vite
- Tailwind CSS
- Mailtrap (pruebas de correo).
- Git y GitLab
- Visual Studio Code
- Railway (preparación de base de datos en producción)

2. Tecnologías

En este capítulo se detalla las tecnologías principales utilizadas en el proyecto y la razón por la que fueron elegidas.

2.1 Backend — Laravel

Tecnologías empleadas:

- **Laravel 12**
- **PHP 8.3**
- **MySQL**
- **Laravel Sanctum** para autenticación
- **Mailtrap** para pruebas de correo

¿Por qué Laravel?

Laravel ofrece una estructura muy clara cuando la aplicación a desarrollar requiere autenticación, validación, controladores, modelos y una API REST como es el caso de este proyecto.

Sus ventajas:

- Generación rápida de endpoints y validaciones
- Integración natural con MySQL mediante Eloquent
- Sencillez al crear middleware, notificaciones y colas
- Sistema de autenticación seguro con Sanctum
- Arquitectura perfecta para una API REST modular

Se opta por Laravel porque permite desarrollar de forma ordenada y escalable aprovechando sus facades y servicios.

2.2 Frontend — Vue 3 + Vite

Tecnologías empleadas

- **Vue 3 (Composition API)** — Framework principal para la construcción de la interfaz.
- **Vite** — Herramienta de desarrollo y build ultrarrápida.
- **Tailwind CSS** — Framework utilitario para estilos rápidos y responsivos.
- **Vue Router** — Navegación entre páginas sin recarga.
- **Pinia** — Gestión global del estado (sesión, usuario, autenticación).
- **Fetch API** — Utilizado para todas las peticiones al backend.

¿Por qué Vue 3?

Quería una interfaz limpia, rápida y sencilla de mantener, con una herramienta que tuviera una curva de aprendizaje moderada. Vue 3 destaca porque:

- Tiene curva de aprendizaje suave
- La Composition API permite organizar mejor la lógica
- Reactividad muy sencilla sin necesidad de librerías adicionales
- Ecosistema muy ligero comparado con React

Además, **Vite** aporta:

- Recarga ultrarrápida en desarrollo
- Builds muy optimizadas

Y **Tailwind CSS** fue elegido porque:

- Ya lo había usado antes y me resulta muy sencillo e intuitivo
- Acelera mucho el maquetado
- Permite un estilo consistente sin escribir toneladas de CSS
- Es ideal para interfaces modernas y limpias (justo lo que buscaba para Agatha)

2.3 Herramientas de Desarrollo

Principales herramientas utilizadas:

- **Visual Studio Code** como editor principal
- **Postman** para probar la API durante el desarrollo
- **Git + GitLab** para control de versiones
- **Laragon** como servidor local rápido
- **Chrome DevTools** para depuración de frontend

2.4 Gestión de Base de Datos

Tecnología empleada:

MySQL 8

¿Por qué MySQL?

- Es estable y compatible con Laravel.
- Facilidad para definir una estructura y relaciones entre usuarios, historias, palabras y lugares.
- Perfecta para estructuras relacionales y consultas sencillas.
- Herramientas GUI como phpMyAdmin facilitan el mantenimiento.

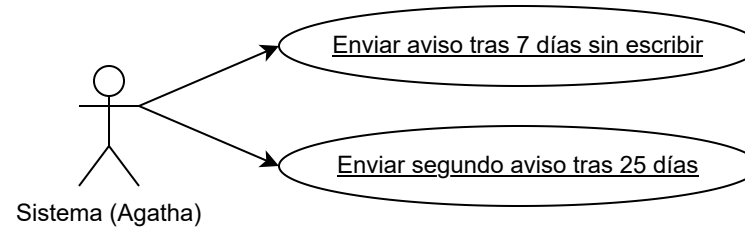
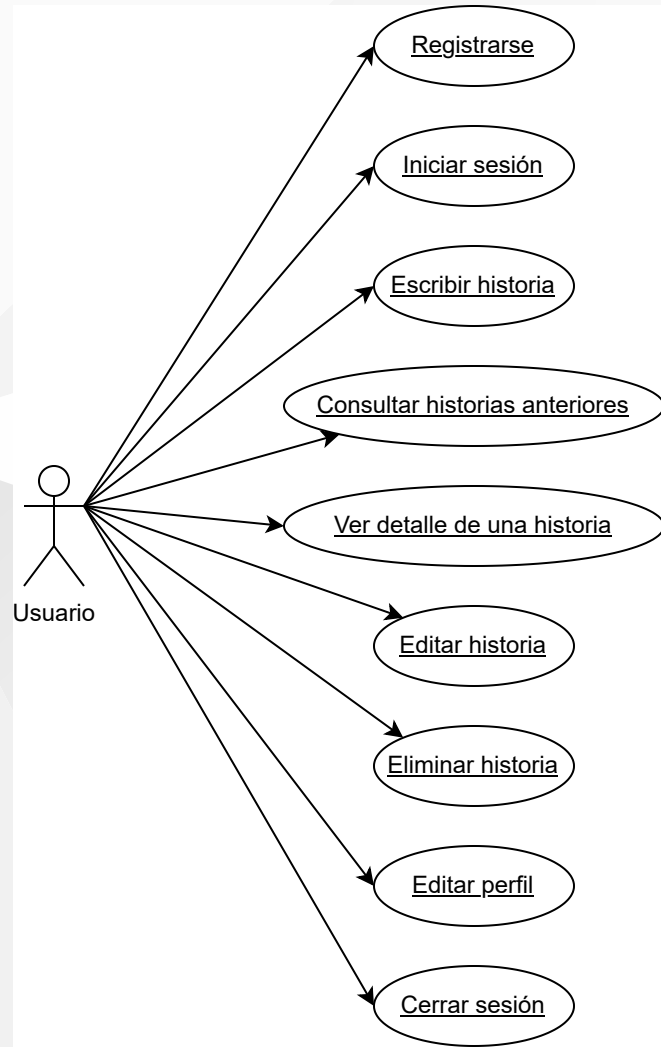
MySQL me encajaba con el proyecto por ser este pensado para una estructura clara y estricta, donde las relaciones entre entidades son necesarias.

3. Análisis

En este capítulo se recogen los elementos clave que definen cómo debe funcionar la aplicación Agatha a nivel funcional y estructural. Incluye los casos de uso principales, el diagrama entidad-relación lógico de la base de datos y varios diagramas que ayudan a entender los flujos internos, como el sistema automático de avisos por inactividad.

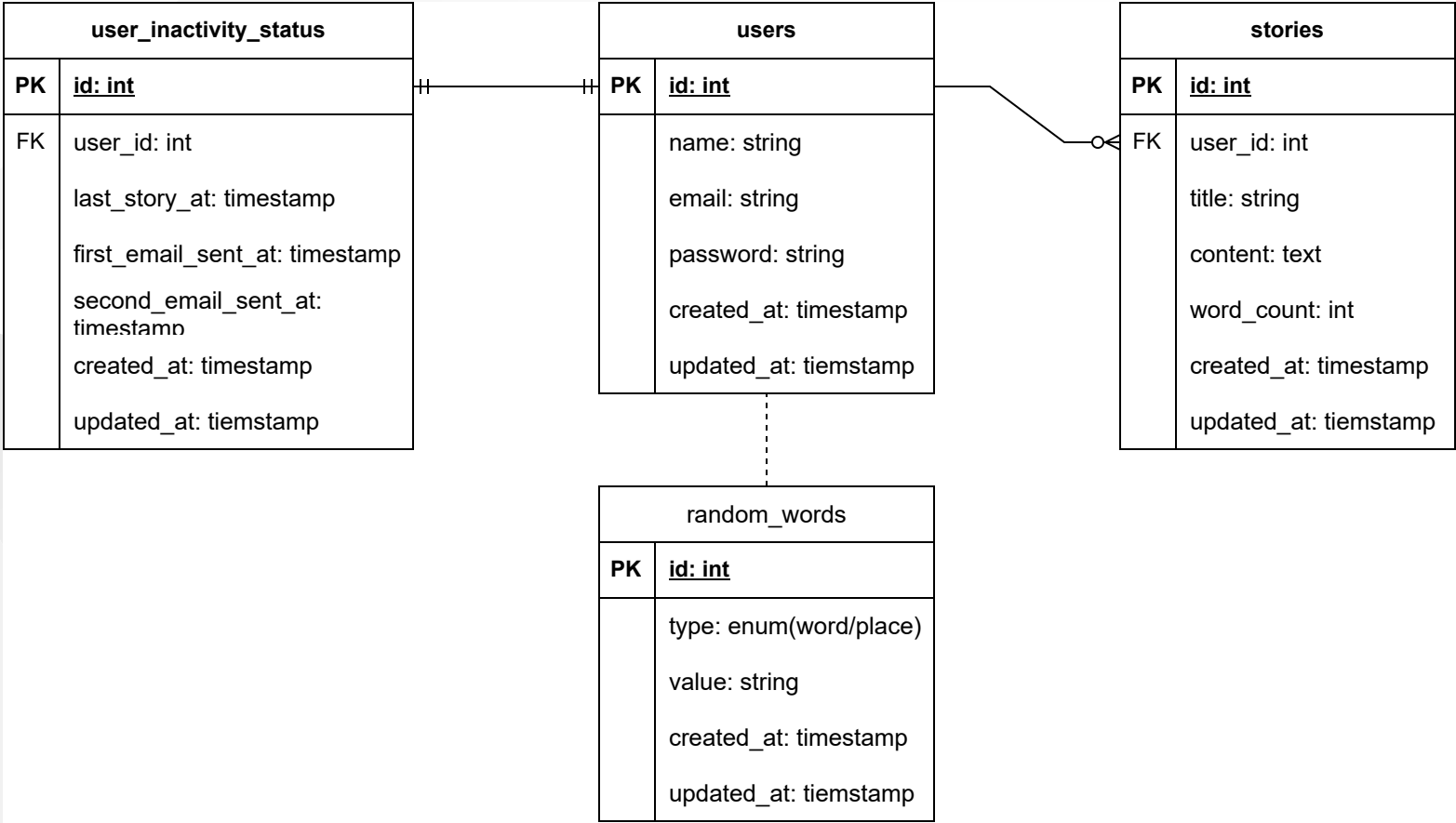
3.1. Casos de Uso

A continuación se representan los casos de uso básicos del sistema, centrándonos en la interacción entre el usuario y la plataforma.



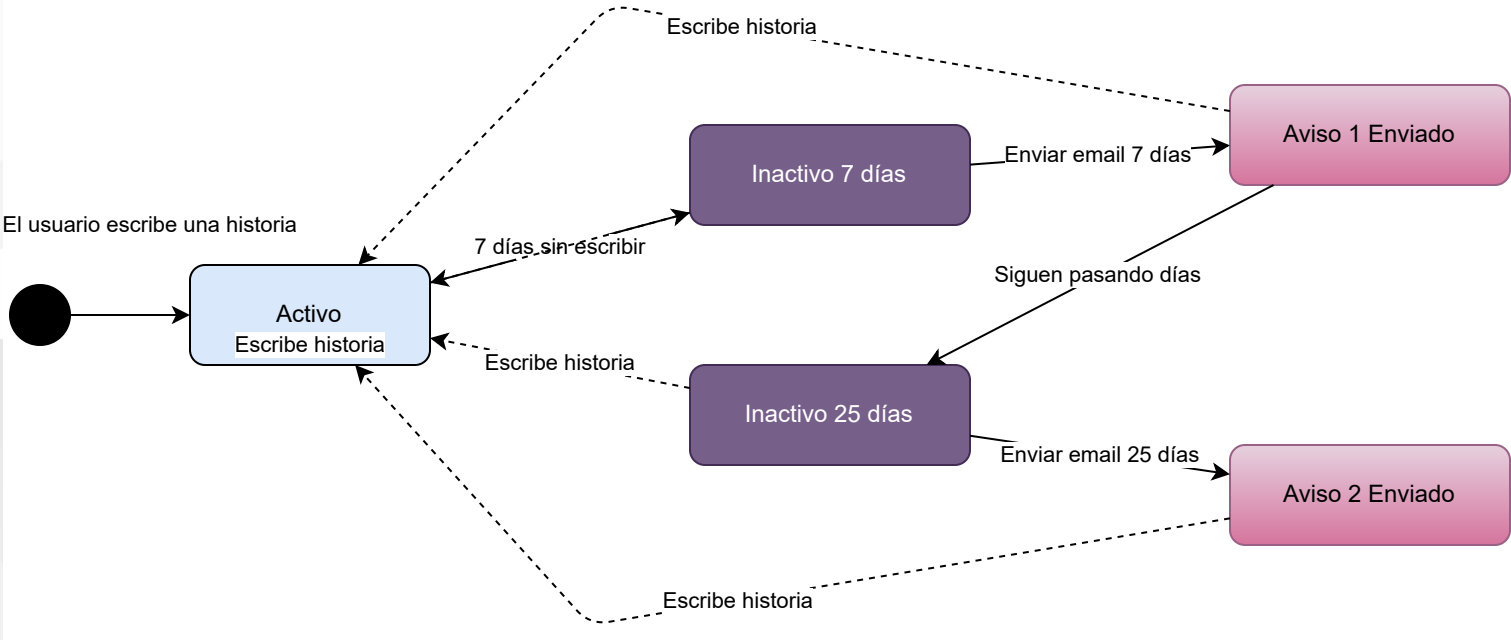
3.2. Diagrama Entidad-Relación (Versión Lógica)

Este diagrama representa la estructura principal de la base de datos utilizada en Agatha. Está adaptado al modelo real del proyecto.



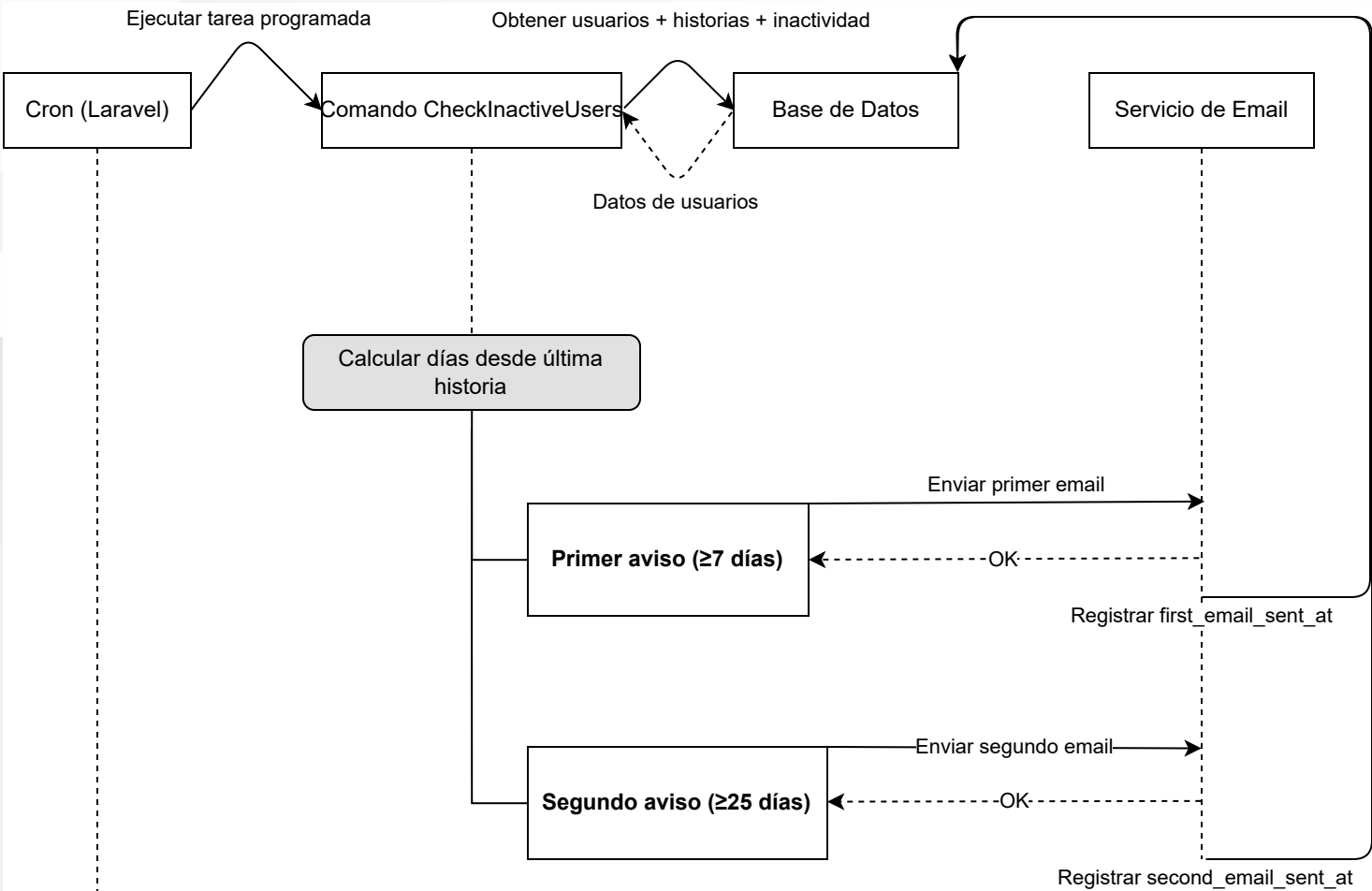
3.3. Diagrama de Estados (Actividad del usuario)

Este diagrama muestra cómo evoluciona el estado de un usuario dentro del sistema en función de su actividad al escribir historias. Es especialmente ilustrativo para entender el sistema automático de correos.



3.4. Diagrama de Secuencia (Proceso de envío automático)

Este diagrama explica cómo se ejecuta el comando users:check-inactive y cómo se envían los correos.



3.5. Requisitos del Proyecto

Requisitos funcionales:

- El usuario debe poder registrarse, iniciar y cerrar sesión.
- El usuario puede escribir nuevas historias.
- Puede consultar la lista de historias escritas.
- Puede ver el detalle de una historia.
- Puede editar o eliminar historias.
- El sistema debe enviar un email tras 7 días sin actividad.
- El sistema debe enviar un segundo email tras 25 días sin actividad.
- Los correos deben enviarse solo una vez por cada periodo.

Requisitos no funcionales:

- La API debe seguir el estándar REST.
- La base de datos debe garantizar integridad y relaciones.
- La aplicación debe ser segura mediante tokens Sanctum.
- Debe ser desplegable tanto en entorno local como remoto.
- Los tiempos de respuesta deben ser inferiores a 200ms en operaciones básicas.

3.6 Tabla Resumen de Endpoints de la API

Categoría	Endpoint	Método	Auth	Descripción	Campos
Autenticación	<code>/register</code>	POST	No	Registro de usuario	name, email, password, password_confirmation
Autenticación	<code>/login</code>	POST	No	Inicio de sesión. Devuelve token Sanctum	email, password
Autenticación	<code>/logout</code>	POST	Sí	Cierra sesión y elimina todos los tokens del usuario	—
Usuario	<code>/me</code>	GET	Sí	Datos del usuario autenticado	—
Usuario	<code>/user</code>	PUT	Sí	Actualiza el perfil del usuario	name?, email?, password?, password_confirmation?
Historias	<code>/story/random</code>	GET	Sí	Devuelve palabra y lugar aleatorios	—
Historias	<code>/story/list</code>	GET	Sí	Lista todas las historias del usuario autenticado	—
Historias	<code>/story/store</code>	POST	Sí	Crea una historia nueva	title?, content, user_id, word, place
Historias	<code>/story/show?storyToken=...</code>	GET	Sí	Devuelve los datos de una historia concreta	storyToken
Historias	<code>/story/update</code>	PUT	Sí	Actualiza una historia del usuario	storyToken, content, user_id, title?
Historias	<code>/story/destroy?storyToken=...</code>	DELETE	Sí	Elimina una historia	storyToken
Inactividad	<i>(sin endpoint)</i>	—	—	Tarea automática ejecutada vía cron	—

3.7. Configuración Base

URL base de la API:

`http://agatha-api.test/api`

Formato de datos:

JSON (tanto para peticiones como para respuestas)

Autenticación:

Bearer Token obligatorio para todos los endpoints protegidos (se envía en la cabecera `Authorization: Bearer <token>`)

Duración del token:

12 horas (generado mediante Laravel Sanctum)

Requisitos para acceder a recursos protegidos:

- Estar autenticado
- Enviar el token válido en cada petición

4. Implementación

En este apartado se describen los componentes más relevantes del proyecto, tanto del **backend en Laravel** como del **frontend en Vue 3**, junto con ejemplos reales de código utilizados.

4.1. Autenticación con Laravel Sanctum

Agatha-api utiliza **Laravel Sanctum** para gestionar autenticación mediante tokens.

El flujo es el siguiente:

1. El usuario inicia sesión desde el frontend.
2. El backend valida las credenciales.
3. Si son correctas, genera un **token con duración de 12 horas**.
4. El token se guarda en LocalStorage y se envía en cada petición con Axios.

Controlador AuthController

```
public function login(Request $request)
{
    $data = $request->validate([
        'email' => 'required|email',
        'password' => 'required',
    ]);

    if (!Auth::attempt($data)) {
        return response()->json(['success' => false, 'message' => 'Incorrecto'], 401);
    }

    $user = $request->user();
    $token = $user->createToken('api-token', ['*'], now()->addHours(12))->plainTextToken;

    return response()->json([
        'success' => true,
        'token' => $token,
    ]);
}
```

4.2. Registro y actualización de usuario

El registro exige contraseñas fuertes (mínimo 8 caracteres, mayúsculas, minúsculas, números y símbolos).

Validación de registro:

```
$data = $request->validate([
'name' => 'required|string|max:255',
'email' => 'required|string|email|max:255|unique:users,email',
'password' => [
'required',
'confirmed',
Password::min(8)->letters()->mixedCase()->numbers()->symbols(),
],
]);
```

Actualización de usuario:

```
$data = $request->validate([
'name' => 'sometimes|string|max:255',
'email' => 'sometimes|email|unique:users,email,' . $user->id,
'password' => [
'nullable',
'confirmed',
Password::min(8)->letters()->mixedCase()->numbers()->symbols(),
],
]);
```

4.3. Frontend de Autenticación (Vue 3 + Pinia)

El frontend almacena el token en LocalStorage y lo añade automáticamente con un interceptor Axios.

Interceptor en api.js

```
api.interceptors.request.use((config) => {
const token = localStorage.getItem('token')
if (token) {
config.headers.Authorization = `Bearer ${token}`
}
return config
})
```

Ejemplo real: vista LoginView.vue

```
async function handleLogin() {
  const data = await login(email.value, password.value)

  if (!data.success) {
    message.value = 'Credenciales incorrectas'
    return
  }

  authStore.setUser(data.user)
  authStore.setToken(data.token)
  router.push('/home')
}
```

4.4. Rutas API

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

Route::middleware('auth:sanctum')->group(function () {
  Route::prefix('story')->group(function () {
    Route::get('/list', [StoryController::class, 'index']);
    Route::post('/store', [StoryController::class, 'store']);
    Route::get('/show', [StoryController::class, 'show']);
    Route::put('/update', [StoryController::class, 'update']);
    Route::delete('/destroy', [StoryController::class, 'destroy']);
    Route::get('/random', [StoryController::class, 'randomWords']);
  });
});
```

```
Route::post('/logout', [AuthController::class, 'logout']);
Route::get('/me', [AuthController::class, 'me']);
```

```
});
```

4.5. Generación de palabra y lugar aleatorios

Agatha usa una única tabla `random_words`, con dos tipos:

word → palabras

place → lugares

```
word = RandomWord::where('type', 'word')->inRandomOrder()->value('value');
place = RandomWord::where('type', 'place')->inRandomOrder()->value('value');
```

4.6. Validación de historias

Se exige que la historia incluya la palabra y el lugar proporcionados:

```
if (
stripos($request->content, word) === false||stripos(request->content,
place) === false ) { return response()->json([ "success" => false, "message" => "La historia debe incluir la palabra '{
word}' y el lugar '{$place}'."
], 422);
}
```

4.7. Guardado de historias con token único

```
$story = Story::create([
'story_token' => Str::uuid()->toString(),
'random_word' => $request->word,
'random_place' => $request->place,
'title' => $request->title,
'content' => $request->content,
'word_count' => this->countWords(request->content),
'user_id' => $request->user_id,
]);
```

4.8. Reset de inactividad al escribir

```
$status = $story->user->inactivity;
```

```
if ($status) {
$status->update([
'last_story_at' => now(),
'first_email_sent_at' => null,
'second_email_sent_at' => null,
]);
} else {
$story->user->inactivity()->create([
'last_story_at' => now(),
]);
}
```

4.9. Frontend de historias (Vue)

Obtener historias

```
export async function getStories() {  
  const { data } = await api.get('/story/list')  
  return data  
}
```

Crear historia

```
export async function createStory(story) {  
  return api.post('/story/store', {  
    title: story.title,  
    word: story.word,  
    place: story.place,  
    content: story.content,  
    user_id: story.user_id,  
  })  
}
```

4.10. Formateo de fechas (frontend)

```
function formatDate(dateString) {  
  return new Date(dateString).toLocaleDateString('es-ES', {  
    year: 'numeric',  
    month: 'numeric',  
    day: 'numeric'  
  })  
}
```

4.11. Sistema de avisos automáticos por inactividad

El comando `users:check-inactive` recorre todos los usuarios y:

- Calcula días sin escribir.
- Envía primer aviso.
- Envía segundo aviso.
- Registra fechas en la tabla `inactivities`.

Código real del comando:

```
$lastStory = user-> stories()-> latest()-> first();days = $lastStory->created_at->diffInDays(now());
```

```
$status = $user->inactivity?: $user->inactivity()->create([  
'last_story_at' => $lastStory->created_at  
]);
```

Primer aviso:

```
if ($days >= 1 && $days < 5) {
```

```
    if (!$status->first_email_sent_at) {  
        Mail::to($user->email)->send(new FirstInactiveUserMail($user));  
        $status->update(['first_email_sent_at' => now()]);
```

```
    }
```

Segundo aviso:

```
if ($days >= 5) {
```

```
    if (!$status->second_email_sent_at) {  
        Mail::to($user->email)->send(new SecondInactiveUserMail($user));  
        $status->update(['second_email_sent_at' => now()]);
```

```
    }
```

4.12. Mailable real

```
class FirstInactiveUserMail extends Mailable
{
    use Queueable, SerializesModels;
```

```
    public function __construct(public $user) {}

    public function content(): Content
    {
        return new Content(
            markdown: 'emails.inactive_first',
            with: ['user' => $this->user]
        );
    }
}
```

4.13. Plantilla Markdown del email

```
@component('mail::message')
¡Te echamos de menos, {{ $user->name }}!
```

Hace más de una semana que no escribes una historia.

```
@component('mail::button', ['url' => config('app.url').'/home'])
Volver a escribir
@endcomponent
```

Gracias,

```
El equipo de Agatha
@endcomponent
```

4.14. Estructura de carpetas del backend (Laravel)

app/	database/
└-- Actions/Fortify	└-- factories/UserFactory.php
└-- Console/Commands	└-- migrations/
└-- Http/Controllers/	└-- 0001_01_01_000000_create_users_table.php
└-- Controller.php	└-- 2025_10_10_095825_create_stories_table.php
└-- StoryController.php	└-- 2025_10_19_162629_add_two_factor_columns...
└-- Api/	└-- 2025_11_12_132255_create_random_words_table.php
└-- Controller.php	└-- 2025_11_26_184153_create_user_inactivity_status_table.php
└-- Http/Resources	└-- seeders/
└-- Mail/	└-- DatabaseSeeder.php
└-- FirstInactiveUserMail.php	└-- RandomWordSeeder.php
└-- InactiveUserMail.php	└-- StorySeeder.php
└-- SecondInactiveUserMail.php	
└-- Models/	resources/views/emails/
└-- RandomWord.php	└-- inactive_first.blade.php
└-- Story.php	└-- inactive_second.blade.php
└-- User.php	
└-- UserInactivityStatus.php	routes/
└-- Providers/	└-- api.php
└-- Traits/WordCountTrait.php	└-- web.php
	└-- console.php

storage/	tests/ y resto
└-- app/private + public	└-- tests/Feature/
└-- framework/	└-- tests/Unit/
└-- cache/data	└-- public/
└-- sessions	└-- docs/ (esta documentación)
└-- views	└-- bootstrap/cache/
└-- logs/	└-- config/

4.15. Comunicación Front-Back (resumen)

- Axios comunica Vue con Laravel.
- Interceptor añade token automáticamente.
- Pinia almacena usuario y token.
- Router protege rutas privadas.
- Backend devuelve JSON estructurado.
- Vue muestra errores y mensajes adecuados.

4.16. Estructura de carpetas del frontend (Vue 3 + Vite + Pinia + Vue Router)

Raíz del proyecto	src/ (código principal)
└─ node_modules/	└─ App.vue
└─ public/	└─ main.js
└─ .gitignore	└─ assets/style.css
└─ index.html	└─ components/Sidebar.vue
└─ jsconfig.json	└─ router/index.js
└─ package.json	└─ services/api.js
└─ package-lock.json	└─ stores/
└─ postcss.config.js	└─ auth.js
└─ tailwind.config.js	└─ stories.js
└─ vite.config.js	└─ views/
└─ README.md	└─ DashboardView.vue
	└─ LoginView.vue
	└─ MainLayout.vue
	└─ MainView.vue
	└─ RegisterView.vue
	└─ StoriesView.vue
	└─ StoryDetailView.vue

5. Conclusiones

5.1. Resultados Obtenidos

El desarrollo de Agatha ha dado como resultado una API sólida, estructurada y funcional, pensada para ofrecer una experiencia fluida en una aplicación creativa centrada en la escritura.

Se han implementado todas las características principales planteadas al inicio:

- Sistema de autenticación seguro con Laravel Sanctum.
- Gestión completa de historias con validaciones, aleatoriedad y formato.
- Generación automática de palabras/lugares aleatorios para fomentar la creatividad.
- Sistema automatizado de notificaciones por inactividad mediante comandos programados.
- Estructura clara de rutas y controladores que permite escalar funcionalidades.

En conjunto, Agatha es una base estable para una aplicación real.

5.2. Aprendizajes Valiosos

Este proyecto ha supuesto un salto importante en mis conocimientos tanto de backend como de frontend, además de en herramientas de desarrollo reales que no había utilizado antes. A lo largo del desarrollo he aprendido:

Laravel y desarrollo backend

- Gestión de autenticación con **Laravel Sanctum** (tokens personales, middleware `auth:sanctum`).
- Construcción de **Mailables**, plantillas Markdown y envío de correos desde la app.
- Programación de tareas automáticas (**Scheduler**).
- Creación y ejecución de **Console Commands**, y cómo integrarlos con lógica real del proyecto.

Vue 3 + Vite (Frontend)

Empece el proyecto sin haber trabajado antes con Vue, y ha sido una de las partes donde más he aprendido:

- Uso de **Composition API** (`ref`, `reactive`, `onMounted`, etc.).
- Manejo de **Pinia** como store global para:
 - guardar el usuario autenticado,
 - guardar y refrescar el token,
 - gestionar sesiones caducadas.
- Router de Vue y protección de rutas mediante **guards** basados en meta `requiresAuth`.
- Envío de peticiones con Axios e interceptores para añadir el token automáticamente.
- Validación de formularios y visualización de errores provenientes del backend.
- Experiencia real con comunicación asíncrona con la API.

Infraestructura local y herramientas nuevas

Este proyecto también me ha permitido aprender varias herramientas que nunca había usado:

- **Laragon**
 - Instalación, configuración y uso como entorno de desarrollo.
 - Manejo del host virtual automático (`agatha-api.test`).
- **Mailtrap**
 - Configuración SMTP para pruebas de envío de emails.
 - Lectura y depuración de correos sin afectar a usuarios reales.
- **Scheduler y Cron**
 - Ejecución de tareas recurrentes con `php artisan schedule:work`.
 - Cómo funcionan los crons reales en servidores Linux.
- **Comandos personalizados**
 - Crear tareas que se ejecutan desde CLI con `php artisan`.
 - Integrarlos con el scheduler.

Gestión de errores y depuración

- Leer y entender errores de Laravel y Vue.
- Depurar respuestas 422 de validación.
- Manejo de excepciones con `try/catch` y mensajes JSON claros.
- Aprendí la importancia de devolver siempre un formato consistente desde el backend.

Git y documentación

- Organización del proyecto con Git/GitLab.
- Creación de documentación en **Markdown**, incluyendo diagramas, tablas, ejemplos y fragmentos de código.

En conjunto, este proyecto me ha permitido aprender cómo se construye una aplicación completa de principio a fin: desde la base de datos, pasando por la API, hasta la interfaz web final que consume esa API.

Además de lo técnico, también ha sido un ejercicio de planificación y resolución de problemas, especialmente al depurar comportamientos inesperados con fechas, validaciones o límites de servicio.

5.3. Cosas que haría de otra manera

Con lo aprendido durante el desarrollo, hay ciertos puntos que replantearía si empezara de cero:

- **Implementar pruebas automatizadas** para evitar errores en funciones sensibles como el scheduler o los mailables.
- **Introducir un sistema de colas (queues)** para el envío de correos, evitando bloqueos por límites de envío.
- **Definir la arquitectura de base de datos desde el principio**, incluyendo tablas auxiliares que se añadieron más tarde.

Cada uno de estos cambios facilitaría el mantenimiento a largo plazo.

5.4. Posibles Mejoras Futuras

El proyecto ha quedado bien definido, pero aun puede mejorarse:

- Añadir un panel estadístico para que el usuario vea su actividad (días consecutivos, número de historias, etc.).
- Implementar etiquetas, categorías o filtros para organizar historias.
- Permitir compartir historias mediante enlaces públicos.
- Incluir un modo competitivo (reto diario, rankings, etc.).
- Autoguardado de la historia en borradores mientras que no se pulse en guardar para no perder el trabajo nunca.
- Boton para mostrar contraseña al hacer login, cambio de contraseña y registro.
- Modo oscuro/claro del sitio web.

5.5. Continuidad y Planes

Una vez finalizado este proyecto, las opciones de continuidad son amplias:

- Integrar un editor de texto con estilos, emoticonos, etc.
- Preparar un despliegue en producción (Railway, Vercel, Forge...).
- Convertirlo en un proyecto personal a largo plazo para practicar nuevas tecnologías.

5.6. Despliegue y Entorno de Producción

Aunque el proyecto se ha desarrollado y probado principalmente en entorno local con Laragon, se ha preparado la base de datos para un futuro despliegue en producción utilizando **Railway** como plataforma cloud.

Migración de la base de datos a Railway

- Se exportó la base de datos completa desde phpMyAdmin (incluyendo estructura, datos de prueba, usuarios, historias, palabras y lugares).
- Se importó exitosamente en un servicio MySQL de Railway.
- La conexión remota desde el entorno local de Laravel se configuró correctamente modificando las variables de entorno en `.env`:

```
DB_CONNECTION=mysql
DB_HOST=[host Railway]
DB_PORT=[puerto Railway]
DB_DATABASE=[nombre BBDD]
DB_USERNAME=[usuario]
DB_PASSWORD=[contraseña]
```

Se verificó la conexión correcta ejecutando comandos como:

```
php artisan config:clear  
php artisan cache:clear  
php artisan tinker
```

Y dentro de tinker:

```
DB::connection()->getPdo();
```

El resultado confirmó la conexión exitosa a través del proxy de Railway (maglev.proxy.rlwy.net).

Ventajas de esta configuración

- Permite trabajar en desarrollo local contra datos reales en la nube (usuarios de prueba, historias existentes, estado de inactividad).
- Facilita pruebas más realistas del sistema de avisos por inactividad y del comportamiento general de la aplicación.
- Prepara el terreno para un despliegue completo futuro: el backend Laravel puede subirse a Railway. El frontend Vue podría desplegarse en Vercel.

6. Bibliografía

A continuación se recoge una lista de los recursos, documentación, tutoriales y materiales que han servido de apoyo durante el desarrollo del proyecto Agatha, junto a una breve descripción de qué se aprendió con cada uno.

- **Documentación oficial de Laravel**
<https://laravel.com/docs>
Guía principal para entender controladores, modelos, migraciones, rutas, mailables, colas, scheduler y configuración general del framework.
- **Laravel Sanctum – Autenticación mediante tokens**
<https://laravel.com/docs/sanctum>
Me permitió aprender cómo funciona la generación y validación de tokens personales, así como la protección de rutas con `auth:sanctum`.
- **Eloquent ORM – Relaciones y consultas**
<https://laravel.com/docs/eloquent>
Recurso clave para aprender a trabajar con relaciones (`hasMany`, `belongsTo`, `hasOne`), consultas avanzadas y formateo de atributos desde el modelo.
- **Mailtrap – Testing de envío de correos**
<https://mailtrap.io/>
Utilizado para probar el envío real de correos sin necesidad de usar un servidor SMTP de producción. También ayudó a entender límites de envío y manejo de errores SMTP.
- **Laravel Scheduler (tareas programadas)**
<https://laravel.com/docs/scheduling>
Fundamental para implementar el sistema automático de avisos por inactividad mediante `php artisan schedule:work` y comandos personalizados.
- **Markdown para documentación técnica**
<https://www.markdownguide.org/basic-syntax/>
Referencia para estructurar la documentación y entender cómo formatear tablas, listas, código y títulos.
- **UUID y manejo de cadenas en Laravel**
<https://laravel.com/docs/helpers>
Sirvió para generar tokens únicos (`Str::uuid()`) y manipular cadenas y fechas.

- **Carbon – Manejo de fechas en PHP**

<https://carbon.nesbot.com/docs/>

Recurso clave para trabajar con intervalos de tiempo, diferencias en días, formateo de fechas y operaciones complejas con timestamps.

- **Diagrams.net (Draw.io)**

<https://app.diagrams.net/>

Utilizado para realizar casos de uso, diagramas de clases y diagramas de flujo de manera visual y exportable a Markdown o imagen.

- **PHP The Right Way**

<https://phptherightway.com/>

Ayudó a entender buenas prácticas generales en PHP, estándares de código y patrones comunes.

- **StackOverflow**

<https://stackoverflow.com/>

Fuente de soluciones rápidas a errores concretos relacionados con SMTP, migraciones, relaciones mal definidas y problemas comunes con el scheduler.