Ernesto Andrés Rosillo Rojas

Full-Stack App

Documentación instalación

Documentación técnica

Sumario

Instalación aplicativo – Código fuente

Al pasar el código fuente necesitaremos Xampp y tener el php del xampp actualizado, ya que se está usando la última versión de laravel.

También necesitaremos tener npm instalado.

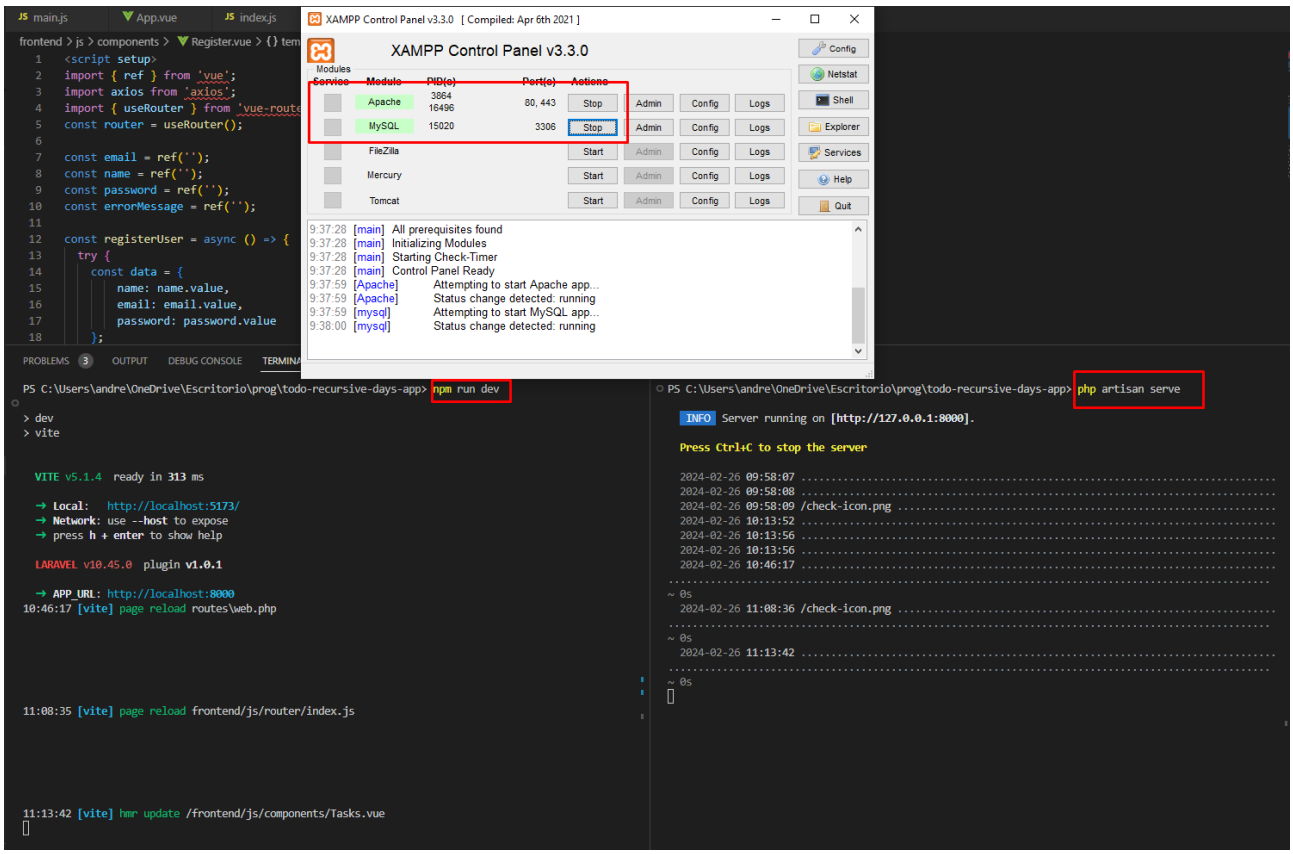
```
C:\Users\andre>php -v
PHP 8.2.12 (cli) (built: Oct 24 2023 21:15:15) (ZTS Visual C++ 2019 x64)
Copyright (c) The PHP Group
Zend Engine v4.2.12, Copyright (c) Zend Technologies

C:\Users\andre>npm -v
8.19.3

C:\Users\andre>
```

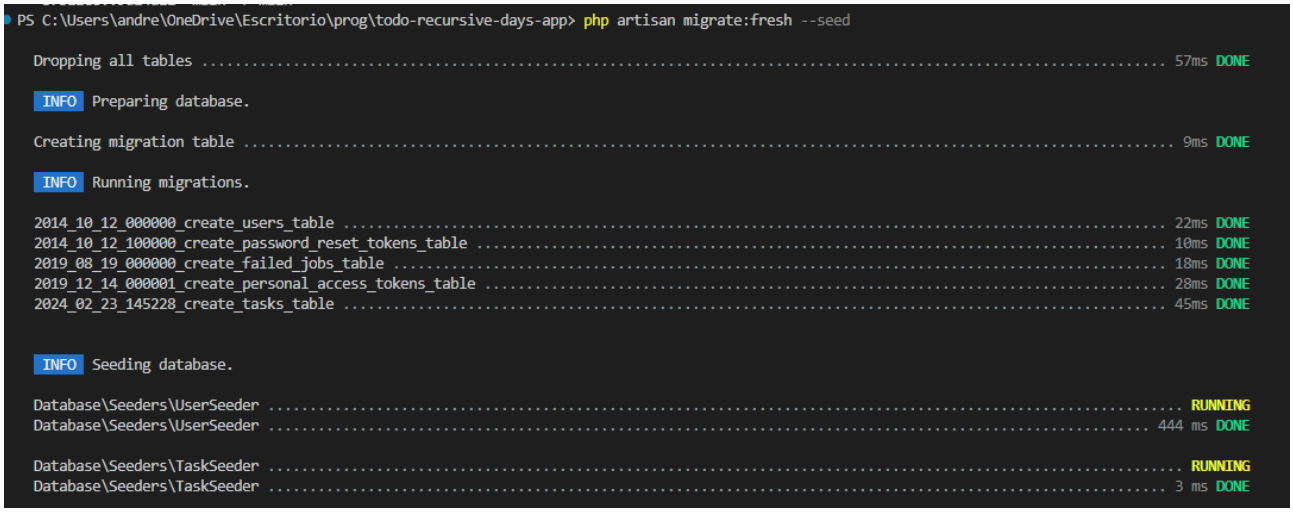
Una vez lo tengamos instalado iniciaremos xampp y, encenderemos mysql y apache.

Luego usaremos los comandos npm run dev y php artisan serve en el directorio de la app



Y en otra consola usaremos este comando para realizar la migración y poner algunos datos de testeo:

php artisan migrate:fresh –seed



Y ya habríamos acabado de instalar la app.

Instalación por git

Usando el siguiente comando podremos instalar también si lo deseamos mediante git (repo público).

git clone https://github.com/andresito1969/todo-recursive-days-app.git **nombreCarpeta**

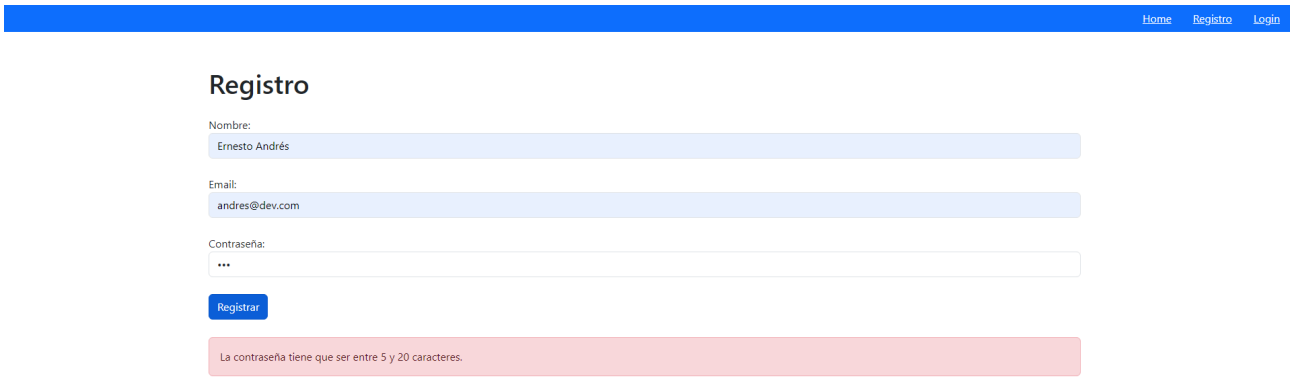
Con este comando pero, tendremos que realizar nosotros las configuraciones del archivo .env para aportar credenciales, ya que este archivo está en el gitignore, lo suyo sería cambiar de nombre el archivo de ejemplo que proporciona git

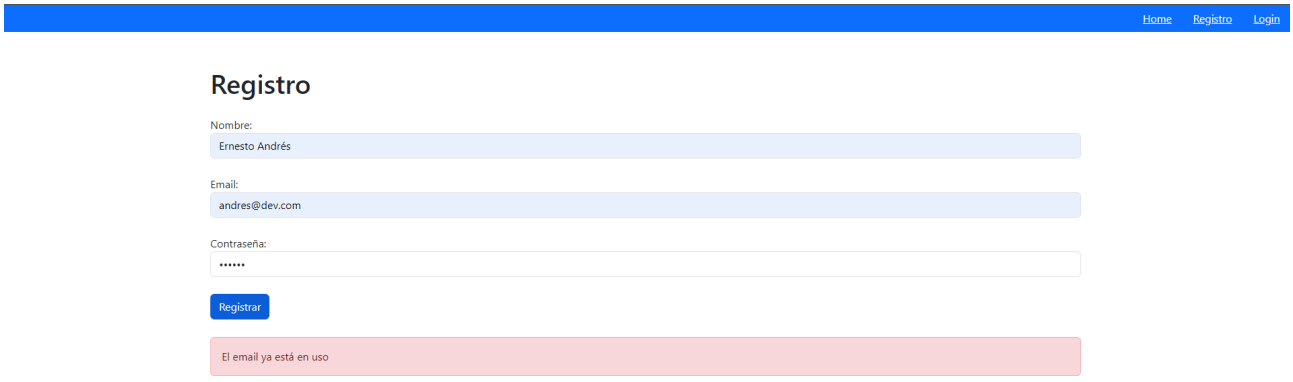
.env.example

Y a parte de instalar xampp + npm, deberíamos de instalar composer.

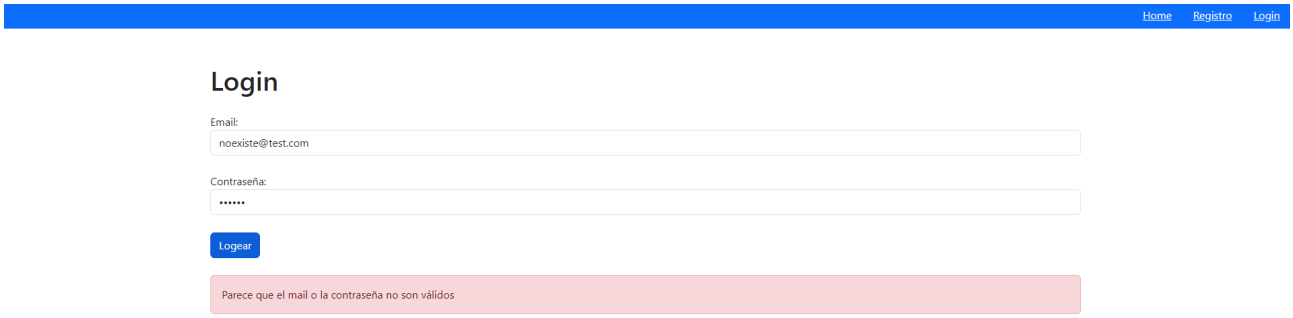
Y finalmente antes de realizar los comandos de npm run dev y php artisan serve, debemos hacer un composer install y npm install, ya que node_modules y vendor son directorios añadidos por gitignore.

Casos de uso - Registro

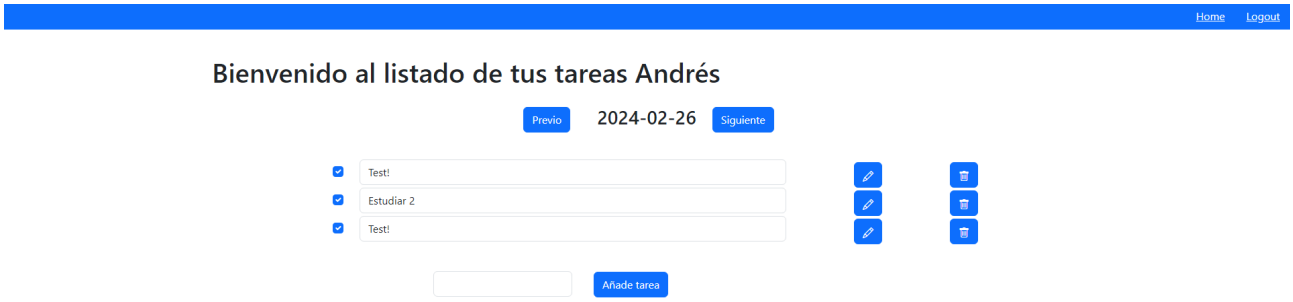


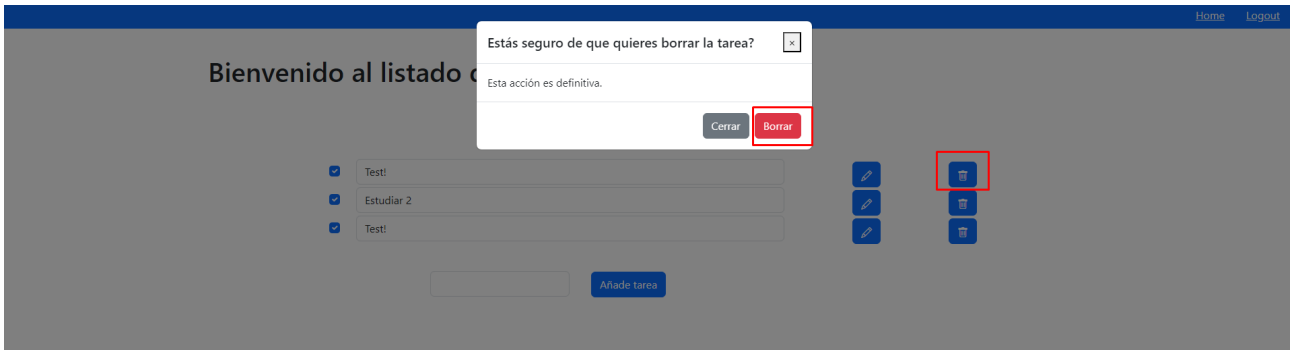


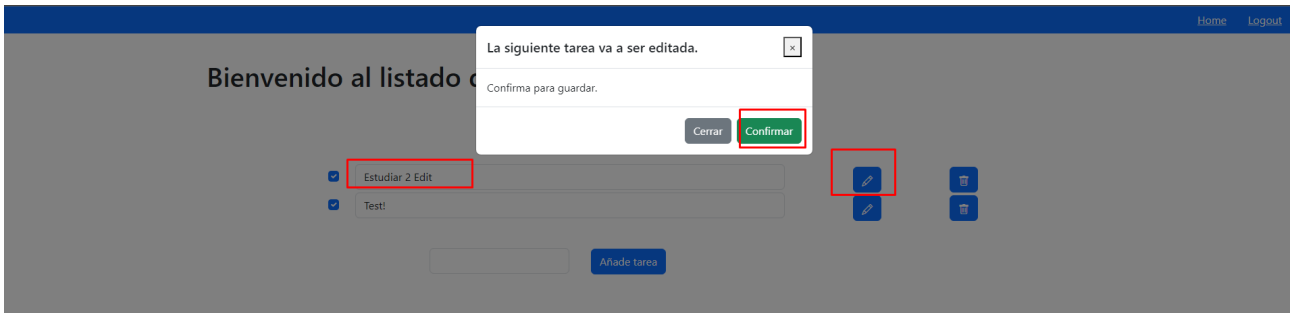
Casos de uso – Login

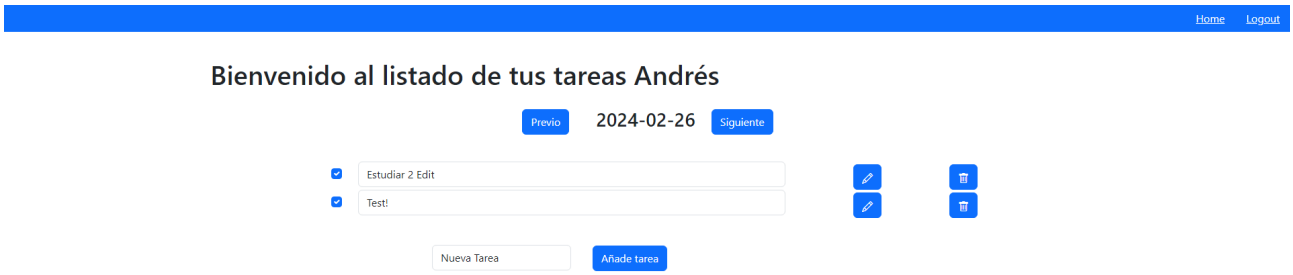


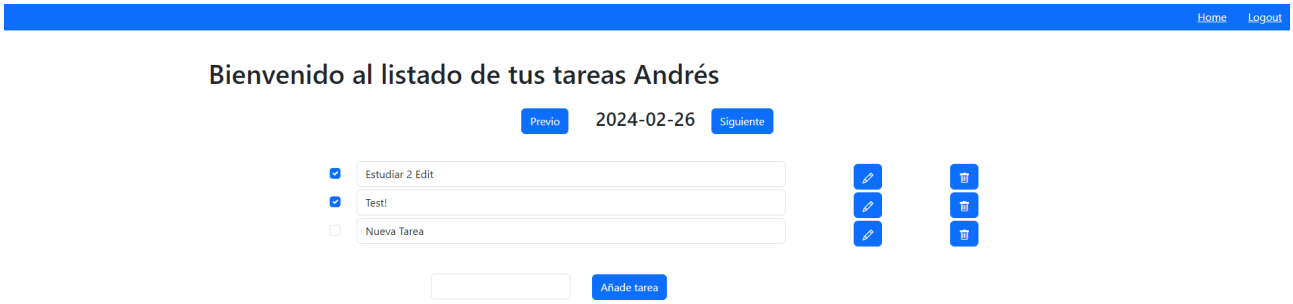
Casos de uso – Home



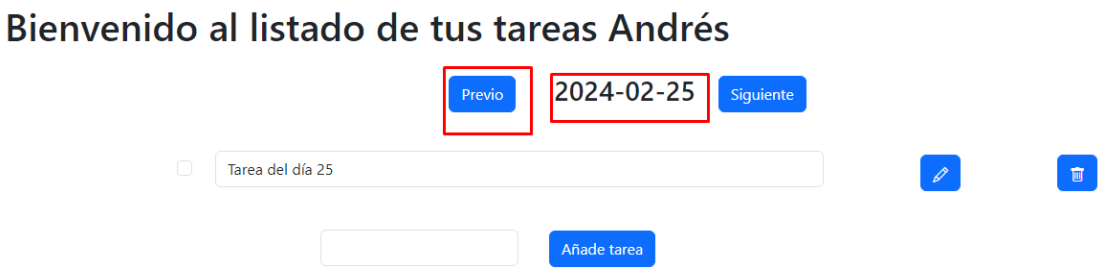


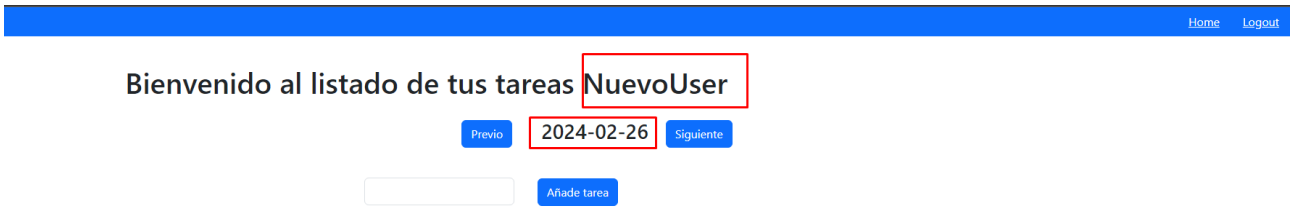



Home
Logout
Bienvenido al listado de tus tareas Andrés
Previo
2024-02-26
Siguiente
Estudiar 2 Edit
Test!
Nueva Tarea
Añade tarea


Home
Logout
Bienvenido al listado de tus tareas Andrés
Previo
2024-02-26
Siguiente
Estudiar 2 Edit
Test!
Nueva Tarea
Añade tarea


Home
Logout
Bienvenido al listado de tus tareas Andrés
Previo
2024-02-27
Siguiente
Añade tarea


Bienvenido al listado de tus tareas Andrés
Previo
2024-02-25
Siguiente
Tarea del día 25
Añade tarea

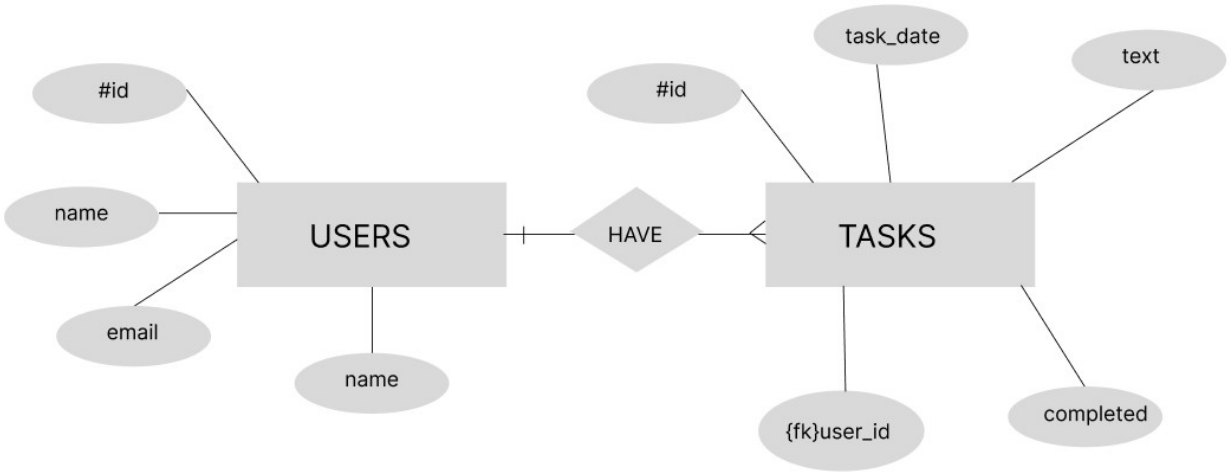

Home
Logout
Bienvenido al listado de tus tareas NuevoUser
Previo
2024-02-26
Siguiente
Añade tarea

BBDD

Modelo ER

La solución encontrada por la parte de BBDD es una solución simple con 2 tablas y una relación 1:N entre Users y Tasks.

No necesitamos realmente más para que la app pueda funcionar, cada tasks tendrá la id del usuario para filtrar y que cada usuario tenga su tarea y, a su vez tendrá task_date para poder filtrar por días en la parte del controlador.



BACKEND

Para esta capa, se ha realizado una arquitectura hexagonal, usando tecnologías/patrones de diseño como MVC, POO, Repository Pattern, Testing, Factory Pattern y API RESTFUL , la arquitectura montada consta de la siguiente estructura:

Modelo:

Modelaje de datos para persistir la BBDD.

app\Models

```php
class Task extends Model
{
    use HasFactory;

    protected $fillable = [
        'task_date',
        'text',
        'completed',
        'user_id'
    ];

    public function user(): BelongsTo {
        return $this->belongsTo(User::class);
    }
}
```

BBDD – Migraciones:

Para poder crear la BBDD en base al modelo, contendrá tanto las tablas como sus propiedades y relaciones.

database\migrations

```php
public function up(): void
{
    Schema::create('tasks', function (Blueprint $table) {
        $table->id();
        $table->timestamps();
        $table->dateTime("task_date");
        $table->string("text");
        $table->boolean("completed");

        // Foreign key
        $table->unsignedBigInteger('user_id');
        $table->foreign('user_id')->references('id')->on('users');
    });
}

/**
 * Reverse the migrations.
 */
public function down(): void
{
    Schema::dropIfExists('tasks');
}
```

BBDD – Factories:

Para poder crear datos falsos de una forma dinámica, se llama a estos métodos estáticos (que pertenecerán al modelo) principalmente al testear la aplicación o al hacer un seeding.

database\factories

```php
public function definition(): array
{
    return [
        'name' => fake()->name(),
        'email' => fake()->unique()->safeEmail(),
        'email_verified_at' => now(),
        'password' => static::$password ??= Hash::make('password'),
        'remember_token' => Str::random(10),
    ];
}
```

BBDD – Seeders:

Para poder añadir valores por defecto a la BBDD o para testear con valores por defecto que nosotros mismos creemos o que nuestra factoria cree por nosotros.

database\seeders

```php
class UserSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        //
        User::factory()
            ->count(5)
            ->hasTasks(2)
            ->create();

        User::factory()
            ->count(1)
            ->hasTasks(10)
            ->create();

        DB::table('users')->insert([
            'name' => 'Andrés',
            'email' => 'andres@dev.com',
            'password' => bcrypt('test123')
        ]);
    }
}
```

Una vez tengamos los datos definidos, podemos pasar a la capa de la lógica de negocio.

Encargada de establecer las reglas y métodos para poder realizar las consultas necesarias.

Controlador:

Va a establecer la comunicación entre la vista y el modelaje de datos.

Ya que la vista (front) al lanzar una petición podrá acceder únicamente a lo que el controlador le proporcione y este a su vez podrá acceder a toda la lógica de negocio.

Principalmente se encargará de que la información pasada por la vista sea veraz y, luego se comunicará con el repositorio para realizar las consultas necesarias.

app\Http\Controllers

```php
class TaskController extends Controller
{
    private $taskRepository;

    public function __construct(TaskRepository $taskRepository) {
        $this->taskRepository = $taskRepository;
    }

    public function getTasksByDay($userId, $day) {
        $taskList = $this->taskRepository->getAllTasksByUserAndDay($userId, $day);
        return new TaskCollection($taskList);
    }
```

Repositorios:

Establecerá los métodos CRUD que interactúan con la BBDD, principalmente el controlador tendrá acceso a esta clase y sus métodos.

El funcionamiento es el siguiente:

El controlador verifica que la petición cumple con lo necesario y, en caso de que sea correcto, llamará al repositorio y realizará la consulta necesaria a la BBDD.

app\Http\Repositories

Tendrá una interface, para poder “firmar” esa clase

```php
class UserRepository implements UserRepositoryInterface {
    public function getUserById($id) {
        return User::findOrFail($id);
    }

    public function getAllUsers() {
        return User::all();
    }

    public function getUserByMail($mail) {
        return User::where('email', $mail)->first();
    }

    public function storeUser(array $data) : void {
        $user = new User($data);
        $user->save();
    }
}
```

JWT - Controlador

Sistema de autenticación, para poder establecer la lógica necesaria para no permitir consultas maliciosas usaremos JWT.

En este archivo en el método login, será donde crearemos y proporcionaremos el token al usuario.

app\Http\Controllers\UserController.php

```php
public function login(Request $request) {
    try {
        $request->validate([
            'email' => 'required|email',
            'password' => 'required'
        ]);
    } catch(Exception $e) {
        return response()->json([
            'error' => "Te faltan datos por rellenar"
        ], 401);
    }

    $credentials = $request->only('email', 'password');
    $token = JWTAuth::attempt($credentials);
    if(!$token) {
        return response()->json([
            'error' => 'Parece que el mail o la contraseña no son válidos'
        ], 401);
    }

    $user = JWTAuth::user();
    return response()->json([
        'token' => $token,
        'token type' => 'Bearer',
        'full_token' => 'Bearer ' . $token,
        'user_id' => $user->id,
        'name' => $user->name,
        'email' => $user->email
    ]);
}
```

JWT - Middleware

Y finalmente en el middleware estableceremos la lógica para denegar cualquier petición que tenga un token inválido o que no sea igual al usuario autenticado.

app\Http\Middleware\JWTMiddleware.php

```php
public function handle(Request $request, Closure $next): Response
{
    try{
        $user = JWTAuth::parseToken()->authenticate();
        $tokenUserId = $user->id;
        $requestUserId = $request->route()->parameter('user_id');

        if($tokenUserId != $requestUserId) {
            return response()->json(['error' => 'Unauthorized'], 401);
        }
    } catch (Exception $e){
        if($e instanceof \Tymon\JWTAuth\Exceptions\TokenInvalidException) {
            // invalid token
            return response()->json(['error' => 'Token is invalid'], 401);
        } else {
            return response()->json(['error' => 'Authorization Token not found'], 401);
        }
    }
    return $next($request);
}
```

Error – Handler

Para que nuestra vista tenga errores dinámicos, se han creado los errores en el controlador, para que al acceder a la api si falla , aprovechar y centralizar la lógica en un solo punto. (Lo suyo sería en realidad no hardcodear valores, lo tengo en mente, crear una tabla incluso de config para manejar MAX y Min length en los campos o crear una constante en toda la aplicación o por .env, hay distintas formas de atacar el problema).

```php
private function registerApiErrorHandler($credentials) : void{
    if(strlen($credentials['email']) > 25) {
        throw new Exception('El email no puede contener más de 25 caracteres.');
    }
    if(strlen($credentials['name']) < 3 || strlen($credentials['name']) > 15) {
        throw new Exception('El nombre tiene que ser entre 3 y 15 caracteres.');
    }
    if(strlen($credentials['password']) < 5 || strlen($credentials['password']) > 20) {
        throw new Exception('La contraseña tiene que ser entre 5 y 20 caracteres.');
    }
}

public function register(Request $request) {
    try {
        $request->validate([
            'email' => 'required|email',
            'name' => 'required',
            'password' => 'required'
        ]);
        $credentials = $request->only('email', 'name', 'password');

        $this->registerApiErrorHandler($credentials);

        $this->userRepository->storeUser($credentials);
        $succeedMessage = 'User ' . $credentials['email'] . ' created!';
        return response()->json(['succeed' => $succeedMessage], 200);
    } catch(Exception $e) {
        $errorMessage = $e->getMessage();
        if($e instanceof ValidationException){
            $errorMessage = "Te faltan datos por rellenar";
        } else if($e instanceof UniqueConstraintViolationException) {
            $errorMessage = "El email ya está en uso";
        }
        return response()->json([
            'error' => $errorMessage
        ], 400);
    }
}
```

Una vez tengamos la lógica de negocio definida y la capa de seguridad del usuario, nos tocará simplemente crear las rutas necesarias, para que nuestro front pueda lanzar las peticiones.

Para ello usaremos el enrutador de laravel.

Router – API:

En este archivo tendremos todas nuestras peticiones API RESTFUL.

Como podemos observar añadimos el middleware JWT, para autenticar las rutas que no sean de inicio de sesión y, añadimos un prefijo user_id que el JWT usará como verificación de usuario que intenta realizar la petición y token del usuario.

rotues\api.php

```php
Route::middleware('auth:sanctum')->get('/user', function (Request $request) {
    return $request->user();
});

Route::group(['middleware' => ['JWTAuth'], 'prefix' => '{user_id}'], function() {
    Route::get('/task/{day}', [TaskController::class, 'getTasksByDay']);
    Route::post('/task/{day}', [TaskController::class, 'storeTaskByDay']);
    Route::patch('/task/{task_id}', [TaskController::class, 'updateTaskById']);
    Route::delete('/task/{task_id}', [TaskController::class, 'deleteTaskById']);
});


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

Router – VIEW:

Como tenemos una aplicación monolítica (Back y front en el mismo repo y directorio), inicialmente la primera petición la devolverá el back, de manera que sea cual sea la ruta que queramos poner, redirigirá al index.blade.php el cual tiene la llamada al front.

routes\web.php

```php
Route::get('/{any}', function() {
    return view('index');
})->where('any', '.*');
```

resources\views\index.blade.php

```
    @vite('frontend/js/main.js')
</head>
<body>
    <div id="app"></div>
</body>
</html>
```

FRONTEND

El frontend se ha realizado con vue y consta de una arquitectura simple, de enrutador – vista – componente, el funcionamiento y estructura sería tal que así:

Instancia app

Instanciaremos la app primeramente gracias al index.blade.php mencionado antes, ya que tiene la id que nuestro createApp usará como instancia.

Y mediante los siguientes archivos montaremos la app, típico en vue

frontend\js\main.js

```
import './bootstrap';

import {createApp} from 'vue';

import App from './App.vue'
import router from './router'

const app = createApp(App);

app.use(router)

app.mount("#app");
```

frontend\js\App.vue

```
<script setup>
import NavMenu from './components/NavMenu.vue'
import {RouterView} from 'vue-router';
</script>

<template>
  <NavMenu/>
  <div class="container">
    <RouterView />
  </div>
</template>

<style scoped>
  0 references
  .auth-wrapper {
    margin-top: 3%;
  }
</style>
```

Router

No solo tendremos el router de la API, también tendremos nuestro propio enrutador para el front, así tendremos la web funcionando SPA.

Aquí definiremos las rutas junto a sus configuraciones.

frontend\js\router\index.js

```
const router = createRouter({
  history: createWebHistory(import.meta.env.BASE_URL),
  routes: [{
      path: '/',
      name: 'home',
      component: HomeView,
      meta: {requiresAuth: true}
    }, {
      path: '/register',
      name: 'register',
      component: RegisterView,
      meta: {
        requiresAuth: false
      }
    }, {
      path: '/login',
      name: 'login',
      component: LoginView,
      meta: {
        requiresAuth: false
      }
    }
  ]
})
```

Y en este bloque usaremos las configuraciones para proteger y manejar nuestras rutas.

```
router.beforeEach((to, from, next) => {
  const isAuthenticated = JSON.parse(sessionStorage.getItem('userData'));
  const isValidRoute = checkValidRoute(to);

  // if user is not auth and route requires auth or is not valid route, redirect to login
  if ((to?.meta?.requiresAuth && !isAuthenticated) || !isValidRoute) {
    console.log("guard 1");
    next('/login');
  }

  // if user is auth and route requires to exclusive not auth, stay in same page
  if(!to?.meta?.requiresAuth && isAuthenticated) {
    console.log("guard 2")
    next(from.path)
  }

  next();
});

const checkValidRoute = (to) => {
  let i = 0;
  let isValidRoute = false;
  console.log(router);
  while(i < router.options.routes.length && !isValidRoute) {
    const routeOption = router.options.routes[i];
    if(routeOption.path === to.path) {
      isValidRoute = true;
    }
    i++;
  }

  return isValidRoute;
}

export default router
```

Vistas

Tendremos 3 vistas en toda la aplicación, como podemos ver en el enrutador y en el directorio de vistas.

Desde las vistas iremos instanciando los componentes que necesitemos.

frontend\js\views

```vue
<script setup>
import Tasks from '../components/Tasks.vue';
import { onMounted, ref } from 'vue';
import moment from 'moment';
import Paginate from '../components/Paginate.vue';
const userData = JSON.parse(sessionStorage.getItem('userData'));
const email = ref('');
const name = ref('');
onMounted(() => {
    email.value = userData?.email || '';
    name.value = userData?.name || '';
})

const selectedDate = ref(moment().format("YYYY-MM-DD"));

const handleDate = (dayNumber) => {
  selectedDate.value = moment(selectedDate.value).add(dayNumber, 'd').format("YYYY-MM-DD");

}

</script>
<template>
  <div class="home">
    <h1>Bienvenido al listado de tus tareas {{ name }}</h1>
    <Paginate @change-date="handleDate" :date="selectedDate"/>
    <Tasks :date="selectedDate"/>
  </div>
</template>

  <style>
  .home{
    margin-top: 3%;
  }
  </style>
```

Componentes - controlador

Y este sería un componente, el cual acabará de dar forma a nuestra aplicación.

Como vemos, usamos los tokens en los headers de cada petición que requiere token.

```vue
<script setup>
import { onMounted, ref, watch } from 'vue';
import axios from 'axios';
import AddTask from './AddTask.vue';
const userData = JSON.parse(sessionStorage.getItem('userData'));
const props = defineProps(['date']);
const date = ref(props.date);
const tasks = ref();
const modalTask = ref();

const getTaskRequest = (date) => {
    const getTaskReq = '/api/' + userData.user_id + '/task/' + date;
    axios.get(getTaskReq, {
        headers: {
            'Authorization' : userData.full_token
        }
    }).then(response => response.data.data.map((values)=> {
        //mapper needed in order to set true/false to boolean completed DDBB value
        // the reason is sql only saves 0 and 1, and our checkbox checked only checks for true and false, not for falsy values
        values.isCompleted = !!values.completed;
        return values
    })).then(response => {
        tasks.value = response;
        return response;
    });
}
```

Componentes - vista

Y este sería un ejemplo del template del componente, como vemos por ejemplo tenemos un modal que nos avisa si queremos confirmar el borrado de cada tarea o no.

```
<template>
    <div class="tasks-container">
        <div v-for="task in tasks" class="row justify-content-md-center">

            <div class="col-sm-6">
                <input v-model="task.text" class=" form-control">
            </div>
            <div class="col-sm-2 text-center">
                <button @click="setModalTask(task)" class="btn btn-primary"
                data-toggle="modal" data-target="#editModal"><i class="bi bi-pencil"></i></button>
            </div>
            <div class="col-sm-2 text-start">
                <button class="btn btn-primary" @click="setModalTask(task)"
                data-toggle="modal" data-target="#deleteModal"><i class="bi bi-trash"></i></button>
            </div>
        </div>
    </div>
    <AddTask :date="date" :tasks="tasks"/>

    <!-- Delete Modal -->
    <div class="modal fade" id="deleteModal" tabindex="-1" role="dialog" aria-labelledby="exampleModalLabel" aria-hidden="true">
        <div class="modal-dialog" role="document">
            <div class="modal-content">
            <div class="modal-header">
                <h5 class="modal-title" id="exampleModalLabel">Estás seguro de que quieres borrar la tarea?</h5>
                <button type="button" class="close" data-dismiss="modal" aria-label="Close">
                <span aria-hidden="true">&times;</span>
                </button>
            </div>
            <div class="modal-body">
                Esta acción es definitiva.
            </div>
            <div class="modal-footer">
                <button type="button" class="btn btn-secondary" data-dismiss="modal">Cerrar</button>
                <button @click="deleteTask(modalTask)" type="button" class="btn btn-danger"
                data-dismiss="modal">Borrar</button>
            </div>
            </div>
        </div>
    </div>
```

Vista -Validaciones – error handler

Como se ha mencionado antes, los errores se manejan en la BBDD y nos llegan de forma dinámica a la vista, aquí tenemos un ejemplo. (se ha recortado parte del código, para que se viera correctamente). Como vemos hay un timeout que limita el tiempo en el que aparecerá el error por pantalla.

frontend\js\componentes\Register.vue

```
const email = ref('');
const name = ref('');
const password = ref('');
const errorMessage = ref('');

const registerUser = async () => {
  try {
    const data = {
        name: name.value,
        email: email.value,
        password: password.value
    };
    await axios.post('/api/register', data);
    sessionStorage.setItem('userMail', JSON.stringify({email: email.value}));
    router.push('/login', data);
  } catch (error) {
    errorMessage.value = error?.response?.data?.error
    setTimeout(() => errorMessage.value = "", 3000);
    console.error('Error al registrar el usuario:', error);
  }
};

</script>

<template>
  <div class="login-container">

      <div class="alert alert-danger" role="alert" v-if="errorMessage">
        {{ errorMessage }}
      </div>
  </div>
</template>
```

Disclaimer realizar este acceso a las propiedades con ? , nos evita escribir mucho código innecesario (error?.response?.data?.error), es algo un poco tricky de js pero funcional.

Testing - Backend

Se han creado testeos de integración por la parte del backend que prueban cada una de las peticiones y más.

Para ejecutarlos:

php artisan test

```
PS C:\Users\andre\OneDrive\Escritorio\prog\todo-recursive-days-app> php artisan test

  PASS  Tests\Unit\ExampleTest
  ✓ that true is true

  PASS  Tests\Feature\TaskTest
  ✓ get user tasks
  ✓ incorrect get tasks missing auth
  ✓ creation task
  ✓ incorrect creation task missing auth
  ✓ incorrect creation task missing fields
  ✓ update task
  ✓ incorrect update task missing auth
  ✓ delete task

  PASS  Tests\Feature\UserTest
  ✓ register
  ✓ incorrect register missing info
  ✓ incorrect register duplicated mail
  ✓ login
  ✓ incorrect login wrong pass

  Tests:    14 passed (14 assertions)
  Duration: 1.83s
```

\tests\Feature

```php
public function test_update_task() : void {
    $responseBody = $this->getUserTestInfo();
    $task = Task::where('user_id', $responseBody->user_id)->first();
    $response = $this->withHeaders([
        'Authorization' => $responseBody->full_token
    ])->patch('/api/' . $responseBody->user_id . '/task/' . $task->id , [
        "text" => "Test!",
        "completed" => 1
    ]);

    $response->assertStatus(200);
}

public function test_incorrect_update_task_missing_auth() : void {
    $responseBody = $this->getUserTestInfo();
    $task = Task::where('user_id', $responseBody->user_id)->first();
    $response = $this->withHeaders([
        'Authorization' => ''
    ])->patch('/api/' . $responseBody->user_id . '/task/' . $task->id, [
        "text" => "Test!",
        "completed" => 1
    ]);

    $response->assertStatus(401);
}
```

Como podemos observar tenemos tanto tests en caso de que funcione (verificando que el status de la respuesta sea 200) como tests en caso de que no funcione (verificando que el status es 40x)

Testing – Frontend

Se han realizado testeos QA por la parte del frontend.