Documentación del Proyecto

Autor: Yonatan Vigilio Lavado

Conocimientos Técnicos

- **Frontend**: HTML, CSS, TailwindCSS, TypeScript (decoradores), React/preact
- **Backend**: MySQL, Sequelize
- **Arquitectura**: Software en Servicios

Extensiones de VSCode para Productividad

- **Linting/Formatting**: Biome, Prettier, Pretty TypeScript
- **Tailwind**: TailwindCSS Intellisense
- **React**: ES7+ React/Redux/React-Native
- **Productividad**:
  - Auto Close Tag, Auto Rename Tag
  - Inline Fold, Multiple Cursor Case Preserve
  - TypeScript Importer

Getting Started

Este proyecto utiliza **Arquitectura de Monorepositorio** con NPM Workspaces:
Guía de Monorepositorios Frontend y servidor en un solo proyecto. Personalmente la mejor arquitectura de software

---

```
npm install # instalar dependecia node
npm run db:migrate # migraciones
npm run serve # correr servidor
npm run dev # correr cliente
npm run build:serve # hacer build en servidor
npm run build:client # hacer build en cliente
```

```
http:localhost:400/api/seed
```

Notas Técnicas

Backend

- Inspirado en NestJS (servicios, controladores, pipes, middlewares, guards) pero optimizado.
- Framework escalable `@vigilio/express-core` (usa Express@4 como core).
  → Ubicación: `/app/config/server.ts`

Frontend

- **Librerías clave**:
  - `@vigilio/preact-fetching`: Alternativa ligera a React Query.
  - `@vigilio/preact-table`: Tablas dinámicas.
  - `@vigilio/preact-paginator`: Paginación.
  - `@vigilio/sweet`: Alternativa a SweetAlert2.
  - `@preact/signals`: Reemplazo de `useState` (mejor rendimiento).
  - `wouter-preact`: Alternativa a React Router v6.
  - `million-js`: Optimización de reactividad.
  - `react-hook-form`: Manejo de formularios.
  - `@vigilio/valibot`: Validaciones (alternativa a Zod).
- **Herramientas**:
  - `@biomejs/biome`: Linting y formateo (solo desarrollo). `bash npm run biome:format npm run biome:check`

Buenas Prácticas y Convenciones

Sintaxis

- **Funciones**:

```
//  Incorrecto
const hi = () => {};

//  Correcto
function hi() {}

// Arrow functions solo en callbacks
array.map(() => {}); //
element.addEventListener(() => {}); //
```

- SQL prácticas Usar JSON no es mála práctica si el lenguajes que usas es tipado fuerte y ventajas de json menos tablas, mas escalable y eficiente.

SERVER PRACTICAS

Schemas

Es el nucleo de cada servicio, la cual servirá para que los demas trabajen (dtos,pipes, entidades,etc).

```
export const usersSchema = objectAsync({
    id: number(),
    fullname: string([minLength(3), maxLenth(20)]),
    email: string([email()]),
    age: number(),
    rol: union([literal("admin"), literal("client")]),
    address: nullable(object({ zip: string(), code: string() })),
    enabled: boolean(),
});
export type UsersSchema = Input<typeof usersSchema>;
```

Entidades

Es conocido tambien como model, el que se conecta y inserta en la base de datos.

```
import { Column, DataType, HasMany, Model, Table } from "sequelize-typescript";
import type { UsersSchema } from "../schemas/users.schema";

@Table({ tableName: "users" })
export class UsersEntity extends Model implements Omit<UsersSchema, "id"> {
    @Column({ type: DataType.STRING(255), allowNull: false })
    fullname: string;

    @Column({ type: DataType.STRING(255), allowNull: false })
    email: string;

    @Column({ type: DataType.INTEGER, allowNull: false })
    age: number;

    @Column({ type: DataType.ENUM("admin", "client"), allowNull: false })
    role: "admin" | "client";

    @Column({ type: DataType.INTEGER })
    address: UsersSchema["address"];
}
```

Controladores

Crear un controlador - Solo sirve para crear endpoint y validar los body, parametros y obtener los resultados. Buena practica usar convesión de laravel:

index, show, store, update, delete. Si muestras vistas: create y edit

```
//@Injectable() - Esto es obligatorio si inyectas un servicio
@Controller("/users")
export class UsersController{
    constructor(private readonly usersService:UsersService){}
     //mostrar todos los usuarios
    @Get("/",[/* entra  middlewares nativos */])
    async index(){
        const result = await this.usersService.index()
        return result;
    }

    @Get("/:id")  //mostrar un usuario
    async show(@Params("id")id:string){}

    @Post("/")
    async store(@Body()body){}

    @Put("/:id")  //editar un usuario
    async update(@Params("id")id:string@Body()body){}

    @Delete("/:id")  //eliminar un usuario
    async destroy(@Params("id")id:string){}
}
```

Dtos y Pipes

Dtos: Para validar el body que viene desde el cliente Pipes: Para validar los parametros /:id Ambos se usan en los controladores

```
export const usersStoreDto = omit(usersSchema, ["id"]);
export type UsersStoreDto = Input<typeof usersStoreDto>;

export const usersUpdateDto = omit(usersSchema, ["id"]);
export type UsersUpdateDto = Input<typeof usersStoreDto>;

@Controller("/users")
export class UsersController {
    constructor(private readonly usersService: UsersService) {}

    @Validator(usersStoreDto) //dto validacion
    @Pipe(objectAsync({ id: string() })) //aca puedes hacer validacion del parametro
    @Get("/:id") //editar un  usuario por id
    async update(@Params("id") id: string, @Body() body: UsersStoreDto) {
        const result = await this.usersService.update(id, body);
        return result;
```

```
    }
}
```

Servicios

Se encarga de la logica de cada controlador.

```
export class UsersService {
    async index() {
        const data = await Users.findAll();
        return { success: true, data };
    }

    async show(id: string) {
        const data = await Users.findByPK(id);
        if (!data) {
            throw new NotFoundException("usuario no found");
            // NotFoundException(404), BadRequestException(400), InternalServerException(500
        }
        return { success: true, data };
    }

    async store(body: UsersStoreDto) {
        const user = await Users.create(body);
        return { success: true, user };
    }

    async update(id: string, body: UsersStoreDto) {
        const { user } = await this.show(id); // aqui de show y lo elimina
        await user.update(body);
        return { success: true, user };
    }

    async destroy(id: string, body: UsersStoreDto) {
        const { user } = await this.show(id);
        await user.destroy();
        return { success: true, message: "Eliminado correctamente" };
    }
}
```

Middlewares

Antes de entrar a un endpoint se ejecuta un middleware.

```
export function Auth() {
    return (
        target: unknown,
```

```
        propertyKey: string,
        _descriptor: PropertyDescriptor
    ) => {
        attachMiddleware(
            target,
            propertyKey,
            async (req: Request, res: Response, next: NextFunction) => {
                const autenticado = true;
                if (!autenticado) {
                    return res
                        .status(401)
                        .json({ success: false, message: "Unauthorized" });
                }
                next();
            }
        );
    };
}

@Controller("/users")
export class UsersController {
    @Auth() // se ejecuta esto primero antes de entrar al enpoint
    @Post("/")
    async store() {}
}
```

Guards

Es un middleware pero se encarga proteger los endpoint segun roles.

```
const user = { id: 1, name: "yonatan", role: "admin" };
if (!user.role !== "admin") {
    return res.status(401).json({ success: false, message: "Unauthorized" });
}
next();
```