Nodejs typescript API Rest

Documentación

Iván Betanzos Macías

Índice:

1 – Arquitectura usada:

Patrón por servicios:

Se ha optado por utilizar una arquitectura por capas bien diferenciadas. Con esto nos aseguramos de que cada capa tenga una única responsabilidad. Esto nos ofrece una serie de ventajas mientra la aplicación va creciendo:

- Fácil lectura y mantenimiento de cada componente
- Diferenciación de responsabilidades y posibilidad de abstraer el proceso
- Mayor escalabilidad

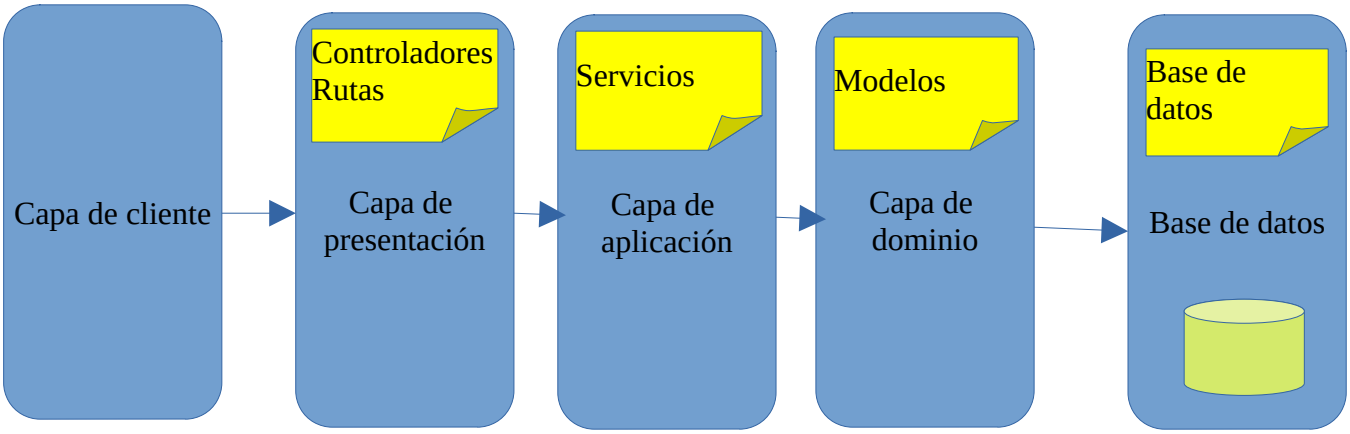
Cada capa sera capaz de comunicarse unicamente con su capa mas cercana, sin poder realizar saltos en ellas, y en una única dirección cada vez.

Las responsabilidades, y por tanto, las capas que vamos a encontrar en esta arquitectura seran las siguientes:

1. Capa de presentación (Controllers): Capa de entrada hacia el Rest API. Responsable unicamente de recibir el request por parte del cliente, comunicarse con la siguiente capa y devolver la respuesta al cliente.
2. Capa de aplicación (Services): Encargado de realizar la logica de negocio y comunicarse con la capa de comunicación con la base de datos.
3. Capa de dominio (Models/Entities): Capa encargada de comunicarse con la base de datos y con la capa de aplicación.

Como nota a esta arquitectura, se opto por añadir una documentación de endpoints basada en swagger, a modo de decoradores en el propio controlador. Esto hace que sea necesaria una transformación en el funcionamiento de estas capas, ya que se hace necesaria una estructura para el correcto funcionamiento de la documentación.

Así, en el proyecto encontraremos una responsabilidad cedida al método de comunicación entre el objeto de ruta y el controlador, que no será más que hacer de intermediario entre el controlador y los objetos de Request y Response.



1.1 – Capa de presentación

En esta capa encontraremos los controladores y rutas hacia los diferentes endpoints de nuestro Rest API.

Tendremos una interfaz base para los controladores, lo que nos asegura la abstracción necesaria para generar los controladores rápidamente.

```
export interface BaseController<T> {
    create(attributes: T): Promise<T>;
    getOne(id: number): Promise<T>;
    getAll(): Promise<T[]>;
    update(id: number, attributes: T): Promise<T>;
    delete(id: number): Promise<DeleteResult>;
}
```

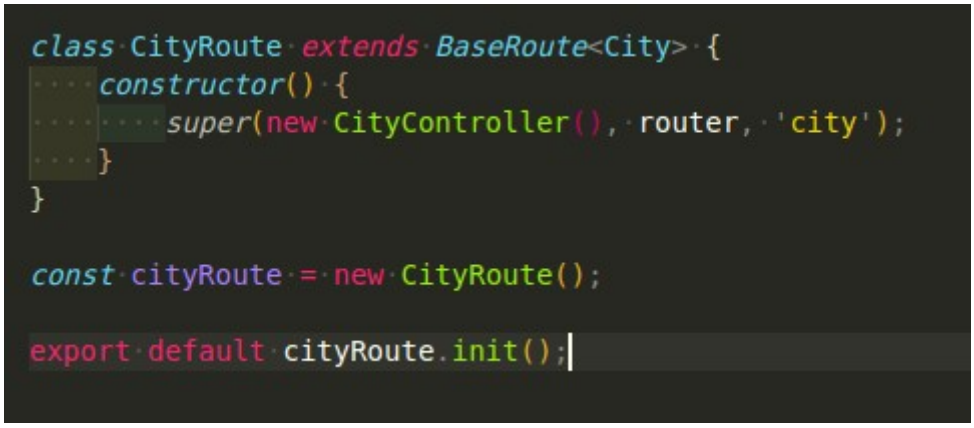
Se implementará la interfaz en cada controlador, añadiendo los decoradores necesarios para generar la documentación del endpoint a través de Swagger:

```
@Tags('Cities')
@Route('api/city')
export class CityController extends Controller implements BaseController<City> {

  /**
   * Create a new City
   * @param attributes
   *
   */
  @Example<BaseJsonResponse<Partial<City>>>({
    message: 'Ok',
    error: false,
    code: 201,
    result: {
      //todo
    }
  })
  @Post('/store')
  public async create(@Body() attributes: City): Promise<City> {
    return cityService.create(attributes);
  }
```

De cara a la generación de rutas, se trató de abstraer lo más posible la generación de estas. Así, podremos generar rápidamente las clases necesarias para el funcionamiento de cada endpoint:

```
//Generic crud routes. Extend it and call init method to generate all needed routes for each method
Complexity is 7 It's time to do something...
export class BaseRoute<T extends Base> {
    constructor(private controller: BaseController<T>, private router: Router, private baseRoute: string) { }

    Complexity is 7 It's time to do something...
    init() {
        this.router.post(`/${this.baseRoute}/store`, asyncError(async ({ body }: Request<{}, {}, T, {}>, res: Response) => {
            const city = await this.controller.create(body);
            res.status(200).json(city);
        }));
        this.router.get(`/${this.baseRoute}/:id`, asyncError(async ({ params: { id } }: Request<{ id: number }>, res: Response) => {
            const address = await this.controller.getOne(id);
            res.status(200).json(address);
        }));

        this.router.post(`/${this.baseRoute}`, asyncError(async (_, res: Response) => {
            const addresses = await this.controller.getAll();
            res.status(200).json(addresses);
        }));

        this.router.put(`/${this.baseRoute}/:id`, asyncError(async ({ params: { id }, body }: Request<{ id: number }, {}, T, {}>, res: Response) => {
            const address = await this.controller.update(id, body);
            res.status(200).json(address);
        }));

        this.router.delete(`/${this.baseRoute}/:id`, asyncError(async ({ params: { id } }: Request<{ id: number }>, res: Response) => {
            const result = await this.controller.delete(id);
            res.status(200).json(!!result);
        }));

        return this.router;
    }
}
```

Haremos que las clases de ruta extiendan esta clase base, para luego inicializarlo llamando a su método init():



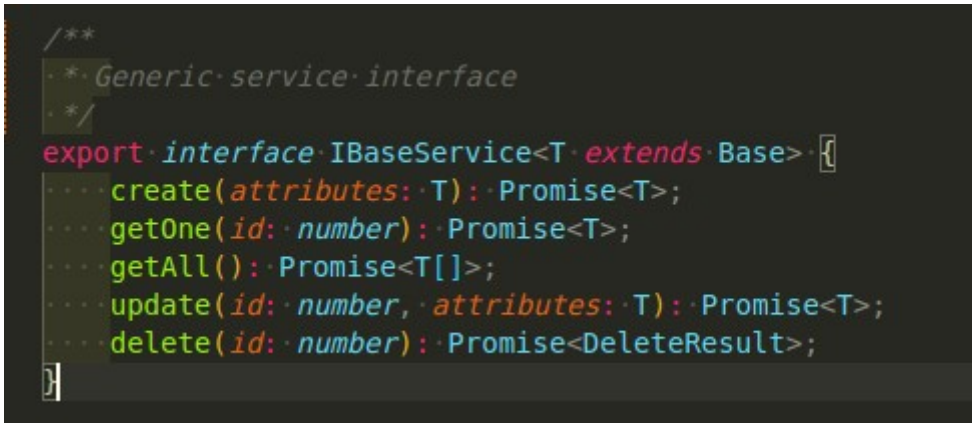
```
class CityRoute extends BaseRoute<City> {
    constructor() {
        super(new CityController(), router, 'city');
    }
}

const cityRoute = new CityRoute();

export default cityRoute.init();
```

Tan solo con esto estaremos generando todas las rutas necesarias para el correcto funcionamiento del CRUD.

1.2 - Capa de aplicación

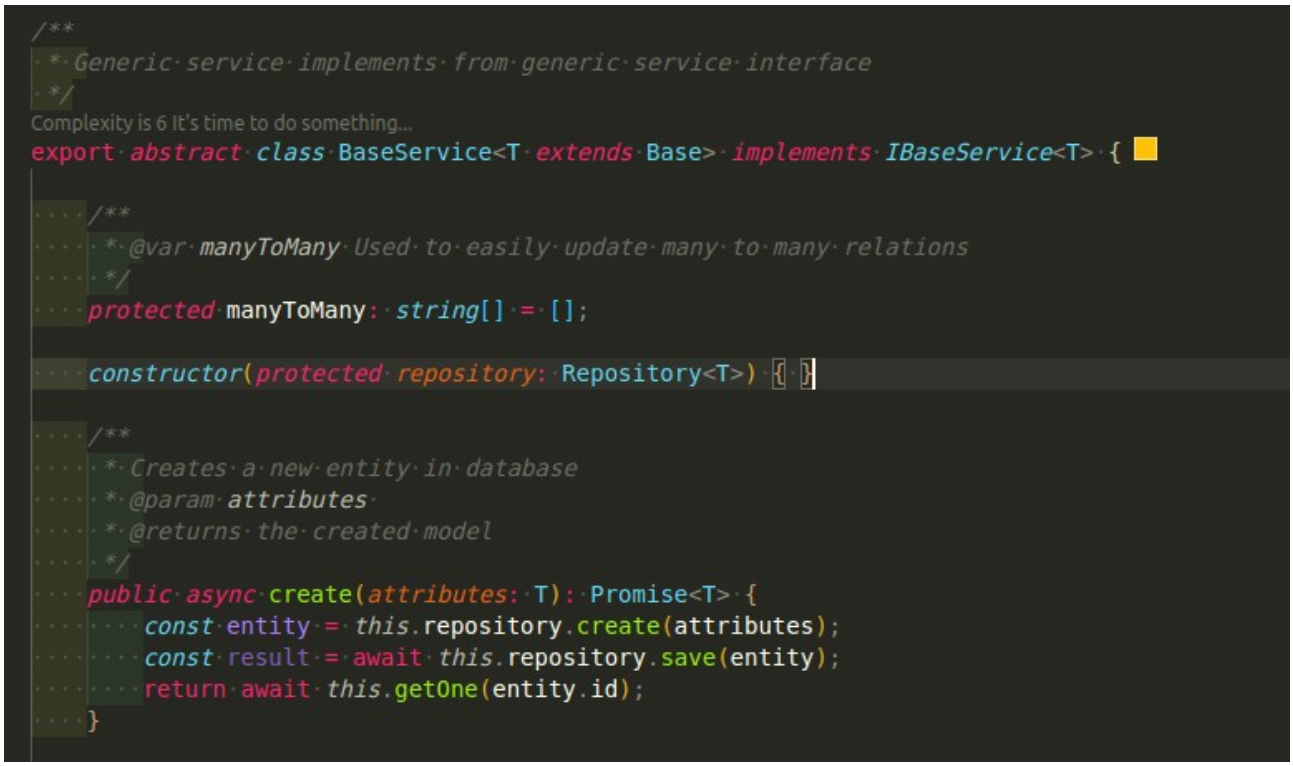
Encontramos aquí los servicios de nuestra aplicación, cuya responsabilidad es recibir los datos del request que el controlador provee y comunicarse con la capa de dominio.

Se encontrará una interfaz como base de todos los servicios. Esto hará que podamos crear un tipo base para todos los servicios.



```
/**
 * Generic service interface
 */
export interface IBaseService<T extends Base> {
    create(attributes: T): Promise<T>;
    getOne(id: number): Promise<T>;
    getAll(): Promise<T[]>;
    update(id: number, attributes: T): Promise<T>;
    delete(id: number): Promise<DeleteResult>;
}
```

Tendremos una clase abstracta, responsable de implementar la interfaz base:



```
/**
 * Generic service implements from generic service interface
 */
Complexity is 6 It's time to do something...
export abstract class BaseService<T extends Base> implements IBaseService<T> {

    /**
     * @var manyToMany Used to easily update many to many relations
     */
    protected manyToMany: string[] = [];

    constructor(protected repository: Repository<T>) { }

    /**
     * Creates a new entity in database
     * @param attributes
     * @returns the created model
     */
    public async create(attributes: T): Promise<T> {
        const entity = this.repository.create(attributes);
        const result = await this.repository.save(entity);
        return await this.getOne(entity.id);
    }
```

Se añadirá una variable “manyToMany”, en el que daremos los nombres de los campos que se relacionen muchos a muchos en la base de datos. Esto es, de cara a la integración automática de estas relaciones, de manera sencilla.

Para ello, en el update se realizará una verificación de esta variable (manyToMany) y se verificará tanto que venga en el request los ids para su sincronización y que en la clase de entidad exista el método necesario para realizar la sincronización:

```
/**
 * Update the model with the provided id
 * @param id
 * @param attributes
 * @returns the updated model
 */
Complexity is 6 It's time to do something...
public async update(id: number, attributes: Partial<T>): Promise<T> {
    const model = await this.getOne(id);

    //Check if the relation name is in our many to many name array and check if a similar
    //method exists in the model class. If so, update the current many to many relationships
    //by calling these methods
    for (let relation of this.manyToMany) {
        if (attributes[relation] && typeof model[`set${relation}`] === 'function') {
            model[`set${relation}`](attributes[relation])
            delete attributes[relation];
        }
    }
    const entity = await this.getOne(id);
    Object.keys(attributes).forEach(key => {
        entity[key] = attributes[key];
    });
    await this.repository.save(entity);
    return this.getOne(id);
}
```

Extracto de la entidad User:

```
/**
 * Set permissions to user. It will be called from generic service if the many to many relation is provided
 * @param permissions
 */
async setpermissions(permissions: any) {    "setpermissions": Unknown word.
  await updatePermissions(this, permissions, db.getRepository(User));
}
```

1.3 - Capa de dominio

Esta capa se encargará de representar los modelos (tablas) de nuestra aplicación.

ORM usado:

El ORM usado en el proyecto es TypeORM. Se optó por el uso de esta tecnología debido a sus múltiples ventajas:

- Facilidad de uso y estructura amigable. Esto hace que sea muy fácilmente legible, mantenible y escalable en el tiempo. Estructura basada en decoradores.

```
@Entity("auth_users")
export class User extends Base {
  @Column({ type: "varchar" })
  name: string;

  @Column({ type: "varchar" })
  last_name: string;

  @Column({ type: "varchar", unique: true })
  username: string;

  @Column({ type: "varchar", unique: true })
  email: string;

  @Column({ type: "varchar", unique: true })
  personal_email: string;

  @Column({ type: "varchar", select: false, })
  password: string;

  @Column({ type: "varchar", nullable: true, select: false })
  password_token: string;

  @Column({ type: "date" })
  birth_date: Date;

  @Column({ type: "integer", nullable: true })
  gender: number;

  @Column({ type: "varchar", nullable: true })
  mobile_phone: string;

  @Column({ type: "varchar", nullable: true })
  home_phone: string;
```

- Compatibilidad con gran parte de las bases de datos disponibles: Podemos crear conexiones con bases de datos relacionales (por ejemplo, MySQL) o bases de datos documentales (ejemplo, MongoDB).

- Posibilidad de crear varios DataSource, realizando operaciones sobre bases de datos diferentes muy fácilmente.

- Fácil uso de consultas hacia la base de datos: Posibilidad de utilizar los métodos del propio ORM (find, findOne, save,…) o crear sql queries encadenando instrucciones.

- No es necesario generar o crear migraciones: No es necesario crear migraciones de cara a crear o modificar tablas en la base de datos. Esto será realizado a través de la propia clase de entidad. Sin embargo, tenemos la posibilidad de crear migraciones para un mayor control de los cambios realizados a las tablas. Esto es especialmente útil una vez el proyecto está en producción.

2 – Estructura de carpetas

```
/
|__ __tests__
        |__ ejemplo_test.test.ts

|__ controllers
        |__ BaseController.ts
        |__ index.ts
        |__ nombre_modulo
                |__ ejemplo.controller.ts
|__ database
        |__ db.ts
|__ interfaces
        |__ index.ts
        |__ nombre_modulo
                |__ ejemplo.interface.ts
|__ middlewares
        |__ ejemplo.ts
|__ models
        |__ Base.ts
        |__ index.ts
        |__ nombre_modulo
                |__ ejemplo.ts
|__ routes
        |__ BaseRoute.ts
        |__ index.ts
        |__ nombre_modulo
                |__ ejemplo.route.ts
|__ Server
        |__ Server.ts
|__ services
        |__ IBaseService.service.ts
        |__ BaseService.service.ts
        |__ index.ts
        |__ nombre_modulo
                |__ ejemplo.service.ts
|__ util
        |__ index.ts
```

|__ ejemplo.ts
|__ .env
|__ .env.example
|__ .gitignore
|__ docker-compose.yaml
|__ index.ts (Punto de entrada de la aplicación)
|__ jest.config.ts
|__ package.json
|__ README.md
|__ tsconfig.json
|__ tsao.json

3 – Flujo de datos