Adrià García Pons

Proyecto Typescript

Índice:

- **Aclaraciones previas**
- **Explicación general del proyecto**
- **Aplicación del proyecto en la vida real.**
- **Estructura del proyecto**
- **Conceptos básicos**

Aclaraciones previas:

Documentación del código:
El código vendrá documentado en el mismo archivo typescript, además de que se explicarán durante la exposición todo lo que pueda generar alguna dificultad a la hora de comprender el código.

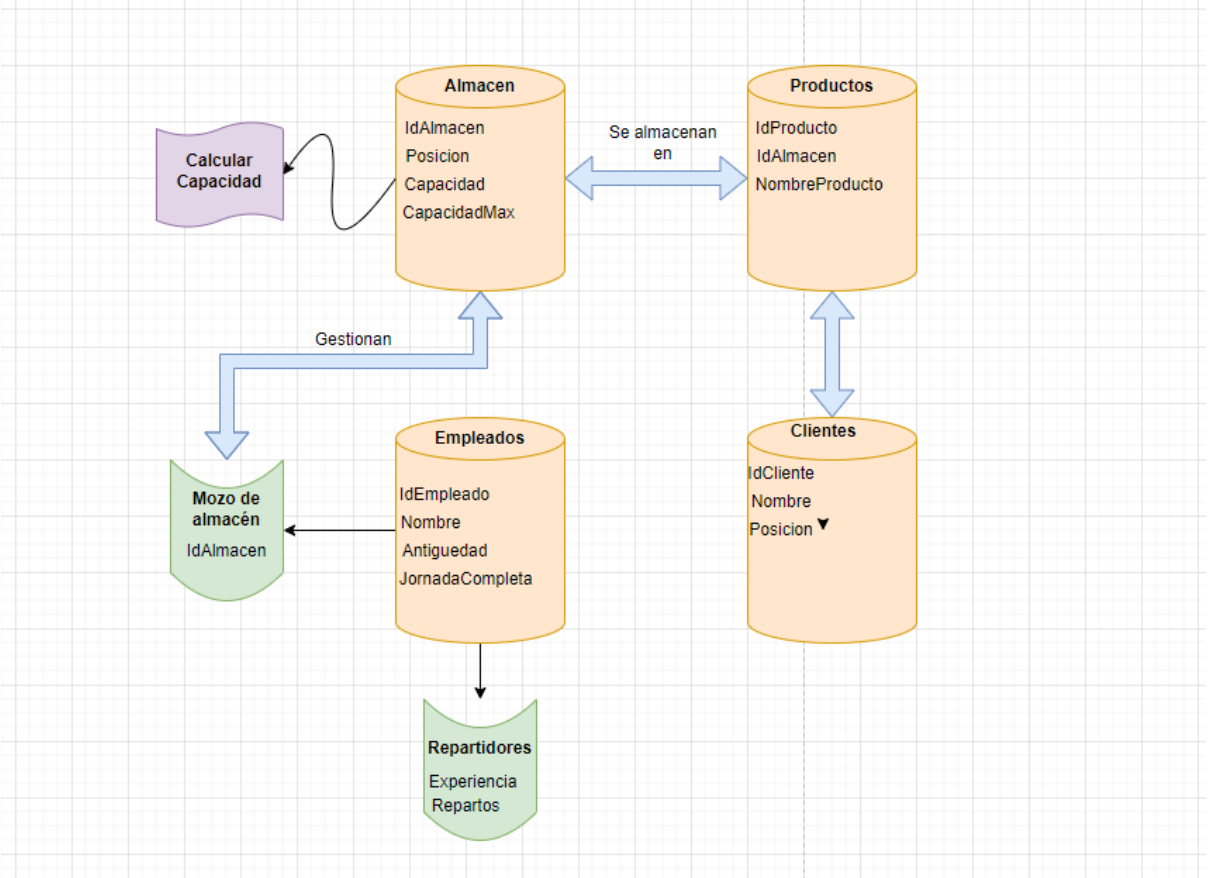
Otras documentaciones:
El resto de documentaciones que hemos realizado durante el curso estarán contenidas dentro del directorio doc del proyecto, toda su información se tendrá como dada por hecho.

Funcionamiento del proyecto:
El proyecto funciona sobre una base de datos en Mongo Atlas, pero tal y como se ha compartimentado el proyecto no es necesario el observar en el propio Mongo Atlas los cambios realizados o los documentos agregados/eliminados/editados/etc.

En caso de querer instalar el proyecto descargalo desde Github, utiliza npm para instalar las dependencias que necesites (entre ellas el compilador tsc).

Explicación General:

Diagrama de relación:



Explicación básica:

Como vemos en el diagrama el proyecto se compone de 4 clases principalmente (y en el caso de Empleados además de 2 herencias).

Cabría a explicar que productos contiene un array de “ProductoAlmacenado” que contiene la información para relacionar el respectivo producto, el almacén donde se almacena y la cantidad del mismo dentro de este almacén.

Dentro de Almacén tendremos un array de trabajadores (mozo de almacén), y en función del número de estos que tengamos albergaremos una mayor o menor capacidad.

Aplicación del proyecto en la vida real:

En caso de tener una empresa de almacenes nos vendría muy bien tener un proyecto como este, ya que nos permite gestionar perfectamente todas las colecciones, realizar búsquedas, crear nuevos datos (e incluso generarlos aleatoriamente). Y también podremos manejar nuestro propio almacén contratando a Mozos de almacén para ampliar nuestra capacidad.

Todo el proyecto es persistente por lo que se adapta a los datos que estén introducidos en todo momento.

Estructura del proyecto:

Como vemos en el archivo readme.md el proyecto se distribuye en distintos directorios los cuales contienen varios archivos.

Dividimos el proyecto en SRC/Doc, el SRC a su vez en clases, schemas, métodos, etc.

Conceptos básicos:

- **Creación de clase y de método constructor**

Para crear la clase la crearemos mediante “export class <nombre>” y le estableceremos todas las variables y el tipo de variable del objeto.

Luego el constructor lo crearemos simplemente poniendo “constructor”, indicando los parámetros que recibirá que serán básicamente las variables necesarias para construir nuestro objeto y finalmente realizaremos los set pertinentes.

```
export class ventilador {
  protected id: number;
  protected tipo: String;
  protected potencia: number; // para acceder en la subclase
  protected peso: number; // para acceder en la subclase
  protected color: String; // para acceder en la subclase
  protected marca: String; // para acceder en la subclase
  protected silenciador: boolean; // para acceder en la subclase

  constructor(id: number, tipo: string, marca: string, potencia: number, peso: number, color: string, silenciador:
    this.id= id;
    this.tipo = tipo;
    this.marca = marca;
    this.potencia = potencia;
    this.peso = peso;
    this.color = color;
    this.silenciador = silenciador;
  }
```

- **Creación de métodos**

Para crear un método de un objeto simplemente dentro de donde declaramos el objeto declararemos el propio método. En este caso vemos como declaramos precio el cual retornará una variable de tipo number y otros dos métodos los cuales devolverán variables de nuestro objeto.

```
  precio(): number {
    let precio = this.potencia/5 + this.peso/10;
    if (this.silenciador == true) {
      precio += 20;
    }
    return precio;
  }

  todo() {
    return `Id: ${this.id}, Tipo: ${this.tipo}, Marca: ${this.marca}, Potencia: ${this.potencia}, Peso: ${this.pes
  }
  general() {
    return `Id: ${this.id}, Tipo: ${this.tipo}, Marca: ${this.marca}, Potencia: ${this.potencia}`;
  }
}
```

- **Utilización de get y set**

Aquí podemos ver como podemos declarar un método get para que nos retorne el valor de una variable del objeto o un método set para modificar este valor, pidiendo como parámetro el dicho valor.

```
get aspasGet() {
  return this.aspas;
}

setAspas(aspas: string) {
  this.aspas=aspas
}
```

- **Creación de herencias, sobreescritura del constructor y uso de super**

En la segunda línea vemos como declararemos un objeto como hicimos con el anterior objeto sin embargo este al tratarse de una herencia tendremos que acompañar a esta declaración de “extends <objeto padre>”.

Como podemos ver unicamente le declararemos la variable aspas ya que el resto las hereda.

En la línea 4 declararemos el método constructor con sus parámetros igual que antes pero incluyendo la nueva variable de la herencia.

Abajo, eso si, realizaremos la función super() para decir que actualizaremos el valor de las variables indicadas tal y como se actualizan en el constructor del objeto padre.

```
1  import { ventilador } from './ventilador';
2  export class ventiladorTecho extends ventilador {
3    protected aspas: string;
4    constructor(id: number, tipo: string, marca: string, potencia: number, peso: number, color: string, silenciador:
5      super(id, tipo, marca, potencia, peso, color, silenciador);
6      this.aspas = aspas;
7    }
```

- **Sobreescritura de métodos**

Podemos ver como el método de Precio() lo sobreescribimos haciendo que la variable que tenemos en esta herencia entre en juego a la hora de calcular el precio del objeto.

```
precio(): number {
  let precio: number;
  precio = super.precio()
  interface AspasList {
    [key: string]: number;
  }
  let foo:AspasList = {};
  foo['madera'] = 15;
  foo['acero'] = 25;
  foo['plastico'] = 5;
  if(this.aspas=="madera" || this.aspas=="acero" || this.aspas=="plastico") {
    precio += foo[this.aspas];
  }
  return precio;
}
```

- **Polimorfismo**

Un claro ejemplo de polimorfismo lo tenemos en el método general() el cual es declarado en el objeto padre ventilador y no es sobreescrito por ninguna de las herencias, y a pesar de esto el método es compatible con cada uno de nuestros objetos ya que todos ellos son capaces de responder a lo que pide.

```
  }
  general() {
    return `Id: ${this.id}, Tipo: ${this.tipo}, Marca: ${this.marca}, Potencia: ${this.potencia}`;
  }
}
```

- **Encapsulación**

La encapsulación se refiere a ser capaces de gestionar correctamente nuestras variables para impedir que estas se puedan ver editadas sin nosotros quererlos por otras funciones.

Para proteger estas variables las declararemos como “protected”, mostraremos el valor de la misma mediante una función get concretada en el objeto y cambiaremos su valor mediante una función set en el objeto también creada.

```
protected potencia: number; // para acceder en la subclase
```

```
get potenciaObjeto()  {
  return this.potencia;
}
```

```
setPotencia(potencia: number) {
  this.potencia=potencia
}
```

```
key.setPotencia(potencia);
```