

# Proyecto Angular

**Jairo Rastrojo Rodríguez**

**2ºASIR :Administración de Sistemas Informáticos en Red**

**Administración de sistemas gestores de base de datos**

**Adolfo Salto Sánchez del Corral**

# **Indice:**

## **Introducción**

### **1-API-Rest**

#### **1.1-Base de datos**

#### **1.2-Schemas**

#### **1.3-Routes**

### **2-Angular**

### **3-Comandos por terminal**

### **4-app**

### **5-Servicios**

### **6-Clases y métodos**

#### **6.1-Clases**

#### **6.2-Métodos**

### **7-Componentes**

### **8-Problemas surgidos**

### **9-Conclusiones**

# Introducción:

Este proyecto se basa en la gestión de un hospital, concretamente en la gestión de urgencias, ya que podremos dar de alta a pacientes que vengan a urgencias y añadir nuevos empleados, todo ello haciendo uso de una res-api.

## 1-API-Rest

Para poder instalar una api-rest y subirla a heroku, consultar el siguiente link:

[https://github.com/rastrojorodriguezjairo2/RES\\_API\\_Hospital/blob/main/doc/respai-hospital.pdf](https://github.com/rastrojorodriguezjairo2/RES_API_Hospital/blob/main/doc/respai-hospital.pdf)

Hemos creado una api-res de la cual tirará nuestro proyecto en angular, con los siguientes elementos:

### 1.1-Base de datos

El archivo database.ts, en el cual ya vienen descritos los métodos necesarios para conectarnos y desconectarnos de la base de datos, nosotros lo que debemos determinar es la ruta de nuestra base de datos, en este caso, alojada en mongo atlas.

```
TS database.ts X
src > database > TS database.ts
1  import mongoose from 'mongoose';
2
3  class DataBase {
4
5      public _cadenaConexion: string = 'mongodb+srv://jairo:1234@cluster0.dynye.mongodb.net/hospital?retryWrites=true&w=majority';
6      constructor(){}
7
8  }
9
10     set cadenaConexion(_cadenaConexion: string){
11         this._cadenaConexion = _cadenaConexion
12     }
13
14     conectarBD = async () => {
15         const promise = new Promise<string>( async (resolve, reject) => {
16             await mongoose.connect(this._cadenaConexion, {
17             })
18             .then( () => resolve(`Conectado a ${this._cadenaConexion}`) )
19             .catch( (error) => reject(`Error conectando a ${this._cadenaConexion}: ${error}`) )
20         })
21         return promise
22     }
23
24     desconectarBD = async () => {
25         const promise = new Promise<string>( async (resolve, reject) => {
26             await mongoose.disconnect()
27             .then( () => resolve(`Desconectado de ${this._cadenaConexion}`) )
28             .catch( (error) => reject(`Error desconectando de ${this._cadenaConexion}: ${error}`) )
29         })
30         return promise
31     }
32
33     export const db = new DataBase()
```

## 1.2-Schemas

En los schemas, debemos dar tipo a nuestro campos, además de las restricciones que queramos añadirle con la que queramos que suban a nuestra base de datos, recordemos que los schemas crean la estructura y la configuración de los datos que serán los documentos que se almacenen en colecciones diseñadas por nosotros.

```
TS pacientes.ts X
src > model > TS pacientes.ts > ...
1 import {Schema, model} from 'mongoose'
2
3 const pacienteSchema = new Schema({
4   _id: {
5     type: Number,
6     unique: true
7   },
8   _nombre: {
9     type: String
10  },
11  _apellido1: {
12    type: String
13  },
14  _apellido2: {
15    type: String
16  },
17  _telefono: {
18    type: String
19  },
20  _medico: {
21    type: String
22  },
23  _urgencia: {
24    type: String
25  },
26  _tipo: {
27    type: String
28  },
29  _pruebas: {
30    type: String[]
31  },
32  _test: {
33    type: String
34  }
35 })
36 export const pacientes = model('pacientes', pacienteSchema)
```

```
TS empleados.ts X
src > model > TS empleados.ts > ...
1 import {Schema, model} from 'mongoose'
2
3 const empleSchema = new Schema({
4   _id: {
5     type: Number,
6     unique: true
7   },
8   _nombre: {
9     type: String,
10    required: true
11  },
12  _apellido: {
13    type: String,
14    required: true
15  },
16  _telefono: {
17    type: String
18  },
19  _puesto: {
20    type: String
21  },
22  _especialidad: {
23    type: String
24  },
25  _idioma: {
26    type: String[]
27  }
28 })
29 export const trabajadores = model('empleados', empleSchema)
```

```
TS pacientes.ts X
src > model > TS pacientes.ts > ...
47 export type pacigeneral = {
48   _id: number | null,
49   _nombre: string | null,
50   _apellido1: string | null,
51   _apellido2: string | null,
52   _edad: number | null,
53   _dni: string | null,
54   _telefono: number | null,
55   _medico: number | null,
56   _urgencia: string | null,
57   _tipo: string | null,
58   _pruebas: string[] | null,
59   _test: string | null
60 }
61 export const pacientes = model('pacientes', pacienteSchema)
```

```
TS empleados.ts X
src > model > TS empleados.ts > ...
34   default: 'String'
35 }
36 })
37
38 export type empgeneral = {
39   _id: number | null,
40   _nombre: string | null,
41   _apellido: string | null,
42   _contacto: number | null,
43   _sueldo: number | null,
44   _puesto: string | null,
45   _especialidad: string | null,
46   _idioma: string[] | null
47 }
48 export const trabajadores = model('empleados', empleSchema)
```

```
TS pacientes.ts X
src > model > TS pacientes.ts > ...
89 export const Atendidos = model('pacientes', pacienteSchema)
```

```
TS empleados.ts X
src > model > TS empleados.ts > ...
67 export const Trabajadores = model('empleados', empleSchema)
```

Una vez que hemos visto las estructuras con las que se subirán nuestros datos a mongo atlas, pasemos a lo siguientes.

## 1.3-Routes

En las rutas, generamos unos métodos que para su ejecución le damos una ruta, por lo que para la activación dicho método es necesario, el link del proyecto seguido de la ruta asignada, por ejemplo:

<https://res-api-hospital-jairo.herokuapp.com/verempleado>

Para estas rutas se han usado diferentes tipos de método:

Post: Permite la creación de nuevos elementos en una base de datos dada.

```
//Añadir un nuevo Empleado
private postempleado = async (req: Request, res: Response) => {
  console.log("HOLA")
  const { id, nombre, apellido, contacto, sueldo, puesto, especialidad, idiomas } = req.body
  await db.conectarBD()
  console.log(req.body)
  const dSchema={
    _id: id,
    _nombre: nombre,
    _apellido: apellido,
    _contacto: contacto,
    _sueldo: sueldo,
    _puesto: puesto,
    _especialidad: especialidad,
    _idiomas: idiomas
  }
  const oSchema = new Trabajadores(dSchema)
  await oSchema.save()
  .then( (doc: any) => res.send(doc))
  .catch( (err: any) => res.send('Error: '+ err))
  await db.desconectarBD()
}
```

Get: Permite visualizar una lista de elementos de una base de datos, en este caso se ha usado un metodo aggregate junto a un \$lookup para poder crear una relación entre las clases de empleados y pacientes.

```
//Listar todos los empleados de la BD
private getEmpleados = async (req: Request, res: Response) => {
  await db.conectarBD()
  .then( async ()=> {
    const query = await Trabajadores.aggregate([
      {
        $lookup: {
          from: 'pacientes',
          localField: '_apellido',
          foreignField: '_medico',
          as: "pacientes"
        }
      }
    ])
    res.json(query)
  })
  .catch((mensaje) => {
    res.send(mensaje)
  })
  await db.desconectarBD()
}
```

Put: Sirve para editar o modificar un documento ya existente en la base de datos.

```
//Actualizar o cambiar los datos de un empleado especifico
private updateempleado = async (req: Request, res: Response) => {
  const {id} = req.params
  const {nombre, apellido, contacto, sueldo, puesto, especialidad, idiomas} = req.body
  await db.conectarBD()
  await Trabajadores.findOneAndUpdate({
    _id: id
  },{
    _nombre: nombre,
    _apellido: apellido,
    _contacto: contacto,
    _sueldo: sueldo,
    _puesto: puesto,
    _especialidad: especialidad,
    _idiomas: idiomas
  },{
    new:true,
    runValidators:true
  })
  .then((doc: any) => res.send(doc))
  .catch((err: any) => res.send ('Error: ' + err))
  await db.desconectarBD()
}
```

Delete: Precisamente sirve para eliminar un documento en especifico en la base de datos

```
private deleteempleado = async (req: Request, res: Response) => {
  const {id} = req.params
  await db.conectarBD()
  await Trabajadores.findOneAndDelete(
    {
      _id:id
    }
  )
  .then( (doc: any) => {
    if (doc == null) {
      res.send(`No encontrado`)
    }else {
      res.json({"Borrado": true})
    }
  })
  .catch ((err: any) => res.send('Error: ' + err))
  db.desconectarBD()
}
```

Para poder hacer que dichos metodos correspondan a rutas, debemos crear un método que se encargue precisamente de vincular ruta con método, para ello esta este método “misRutas()”

```
misRutas(){
  this._router.post('/newpaciente', this.postpacientes),
  this._router.post('/newempleado', this.postempleado),
  this._router.get('/verpaciente', this.getPacientes),
  this._router.get('/verurgencias', this.getUrgencias),
  this._router.get('/vercovid', this.getCovid),
  this._router.get('/verempleado', this.getEmpleados),
  this._router.get('/vermedicos', this.getMedicos),
  this._router.get('/veradministrativos', this.getAdministrativos),
  this._router.get('/buspaciente/:id', this.getbuspaciente),
  this._router.get('/busempleado/apellido', this.getbusempleado),
  this._router.put('/actualizarpaciente/:id', this.updatepaciente),
  this._router.put('/actualizarepleado/:id', this.updateempleado),
  this._router.delete('/eliminarpaciente/:id', this.deletepaciente),
  this._router.delete('/eliminarpleado/:id', this.deleteempleado)
```

Para subir los datos a las colecciones, se ha empleado el programa postman, con el cual podemos también comprobar y ver los errores que van surgiendo para su posterior resolución

The screenshot shows the Postman interface. At the top, a GET request is made to the URL `https://res-api-hospital-jairo.herokuapp.com/vermedicos`. Below the URL bar, tabs for Params, Authorization, Headers (8), Body, Pre-request Script, Tests, and Settings are visible. The 'Headers' tab is selected, showing 12 headers. Below the headers, there are tabs for Pretty, Raw, Preview, Visualize, and JSON. The 'JSON' tab is selected, displaying the response body in a formatted JSON structure. The response is a JSON object with the following structure:

```
{
  "_id": 1,
  "_nombre": "Marcos",
  "_apellido": "Lopez",
  "_contacto": 639561262,
  "_sueldo": 10000,
  "_puesto": "medico",
  "_especialidad": "Pediatria",
  "_idiomas": [],
  "_v": 0,
  "pacientes": [
    {
      "_id": 2,
      "_nombre": "Ana",
      "_apellido1": "Asero",
      "_apellido2": "Amelo",
      "_edad": 10,
      "_dni": "45112168n",
      "_telefono": 654539879,
      "_medico": "Lopez",
      "_urgencia": "Brecha en la cabeza",
      "_tipo": "urgencia",
      "_pruebas": [
        "Tac"
      ]
    }
  ]
}
```

## 2-Angular

Angular es un framework de Javascript de código abierto escrito en Typescript (un lenguaje de programación fuertemente tipado) su objetivo principal es el desarrollo de aplicaciones de una sola página.

## 3-Comandos por terminal

Vamos a repasar unos comandos básicos para poder llevar a cabo nuestro proyecto en angular, lo primero seria instalar angular, para poder ejecutar sus comando y hacer uso de sus funciones, usaremos el comando:

1º “npm install -g @angular/cli” lo cual nos dará acceso a todos los comandos de angular que para llamarlos usaremos “ng”.

2º “ng new angular **proyecto**” creará un proyecto angular en el lugar donde estemos situados, por cierto siempre que queramos emplear comandos dentro de nuestro proyecto angular, debemos situarnos en nuestro proyecto, para ello nos desplazaremos con “cd **proyecto**” al menos en este caso de ejemplo.

3º “ng serve --open” sirve para lanzar nuestra aplicación angular, a la cual podremos acceder en “<http://localhost:4200>” por defecto, si quisiéramos cambiar el puerto que emplea nuestro proyecto usaremos el comando “ng serve --port **4201**” para que usemos ese puerto.

4º “ng generate component **proyecto**” permite generar un nuevo componente en el cual podremos empezar a trabajar.

5º “ng generate service **proyecto**” sirve para crear un servicio nuevo con el nombre asignado para poder trabajar con ello.



## 4-app

Al crear un nuevo proyecto en angular nos percataremos que existen muchos archivos, sin embargo en el que vamos a trabajar sobre todo es en los archivos dentro del directorio app, ya que existen algunos elementos a tener en cuenta:

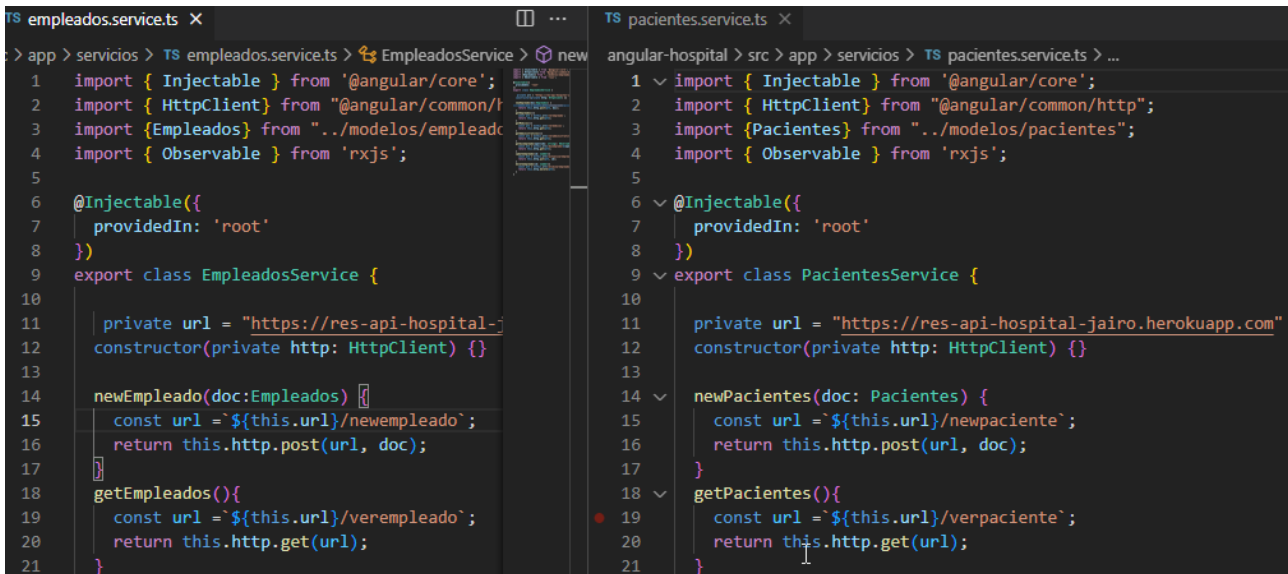
`app-routing.modules.ts`: En este archivo debemos importar cada uno de los componentes que creemos y darles una ruta de acceso, para poder llegar a dichos componentes.

`app.modules.ts`: Este archivo es en el cual de forma automática (por lo general) se genera cada componente, servicio o modulo que creemos o instalemos, por lo que es necesario revisarlo cada vez que instalamos o creamos algo nuevo para comprobar que esta, de lo contrario lo debemos importar y definir si es una importación, un modulo o un servicio.

También existe un `app.component.ts`, `app.component.html`, `app.component.css`, los cuales repercutirán y servirán como elementos generales que permanecerán en todas las páginas de nuestro proyecto, por encima de los otros componentes, por ejemplo en este proyecto en el `app.component` se ha definido un menú que será el general que aparece en todos los componentes.

## 5-Servicios

Un servicio es un archivo.ts que permite gracias al decorador `@injectable()` utilizar los metodos y dependencias de otras api, precisamente gracias a estos servicios es como podremos acceder a los métodos que hemos creado en la res-api



```
TS empleados.service.ts X
> app > servicios > TS empleados.service.ts > EmpleadosService > new
1 import { Injectable } from '@angular/core';
2 import { HttpClient } from '@angular/common/http';
3 import { Empleados } from '../modelos/empleado';
4 import { Observable } from 'rxjs';
5
6 @Injectable({
7   providedIn: 'root'
8 })
9 export class EmpleadosService {
10
11   private url = "https://res-api-hospital-";
12   constructor(private http: HttpClient) {}
13
14   newEmpleado(doc: Empleados) {
15     const url = `${this.url}newempleado`;
16     return this.http.post(url, doc);
17   }
18   getEmpleados(){
19     const url = `${this.url}verempleado`;
20     return this.http.get(url);
21   }
}

TS pacientes.service.ts X
angular-hospital > src > app > servicios > TS pacientes.service.ts > ...
1 import { Injectable } from '@angular/core';
2 import { HttpClient } from '@angular/common/http';
3 import { Pacientes } from '../modelos/pacientes';
4 import { Observable } from 'rxjs';
5
6 @Injectable({
7   providedIn: 'root'
8 })
9 export class PacientesService {
10
11   private url = "https://res-api-hospital-jairo.herokuapp.com";
12   constructor(private http: HttpClient) {}
13
14   newPacientes(doc: Pacientes) {
15     const url = `${this.url}newpaciente`;
16     return this.http.post(url, doc);
17   }
18   getPacientes(){
19     const url = `${this.url}verpaciente`;
20     return this.http.get(url);
21   }
}
```

En private url, hemos guardado la dirección de la api-rest, para poder usar en los métodos de abajo, las rutas.

## 6-Clases y métodos

Primero, decir que dispondremos de un total de dos clases principales: Empleados y Pacientes, las cuales a su vez dispondrán cada uno de dos subclases Empleados(Médicos, Administrativos) y Pacientes(Urgencias, Covid), los cuales gracias a las herencias y el uso del polimorfismo podremos realizar algunos métodos heredados interesantes.

## 6.1-Clases

Primero la clase de empleados y sus subclases, médicos y administrativos

```
TS empleados.ts • TS medicos.ts • TS administrativo.ts •
angular-hospital > src > app > modelos > TS empleados.ts > angular-hospital > src > app > modelos > TS medicos.ts > Medicos > angular-hospital > src > app > modelos > TS administrativo.ts > Administrativo > cc

1 export class Empleados {
2   private _id: number;
3   private _nombre: string;
4   private _apellido: string;
5   private _contacto: number;
6   protected _sueldo: number;
7   private _puesto: string;
8   constructor(id: number,
9     nombre:string,
10    apellido: string,
11    contacto: number,
12    sueldo: number,
13    puesto: string){
14     this._id=id;
15     this._nombre=nombre;
16     this._apellido=apellido;
17     this._contacto=contacto;
18     this._sueldo=sueldo;
19     this._puesto=puesto
20   }
21   get id(){
22     return this._id;
23   }
24 }

1 import {Empleados} from "../empleados";
2
3 export class Medicos extends Empleados {
4   private _especialidad: string;
5   constructor (id: number,
6     nombre:string,
7     apellido: string,
8     contacto: number,
9     sueldo: number,
10    puesto: string,
11    especialidad: string,){
12     super (id, nombre,
13       apellido, contacto,
14       sueldo, puesto)
15     this._especialidad = especialidad
16   }
17   get especialidad () {
18     return this._especialidad
19   }
20   override get sueldo () {
21     return this._sueldo
22   }
23 }

1 import {Empleados} from "../empleados";
2
3 export class Administrativo extends Empleados {
4   private _idiomas: Array<string>;
5   constructor (id: number,
6     nombre:string,
7     apellido: string,
8     contacto: number,
9     sueldo: number,
10    puesto: string,
11    idiomas: Array<string>){
12     super (id, nombre, apellido,
13       contacto, sueldo, puesto)
14     this._idiomas = idiomas
15   }
16   get idiomas () {
17     return this._idiomas
18   }
19   override get sueldo () {
20     return this._sueldo
21   }
22 }
```

Y ahora veremos la clase pacientes, junto a sus subclases, urgencias y covid

```
TS pacientes.ts • TS urgencias.ts • TS covid.ts •
angular-hospital > src > app > modelos > TS pacientes.ts > Pacientes > constructor angular-hospital > src > app > modelos > TS urgencias.ts > Urgencias > constructor angular-hospital > src > app > modelos > TS covid.ts > Covid > constructor

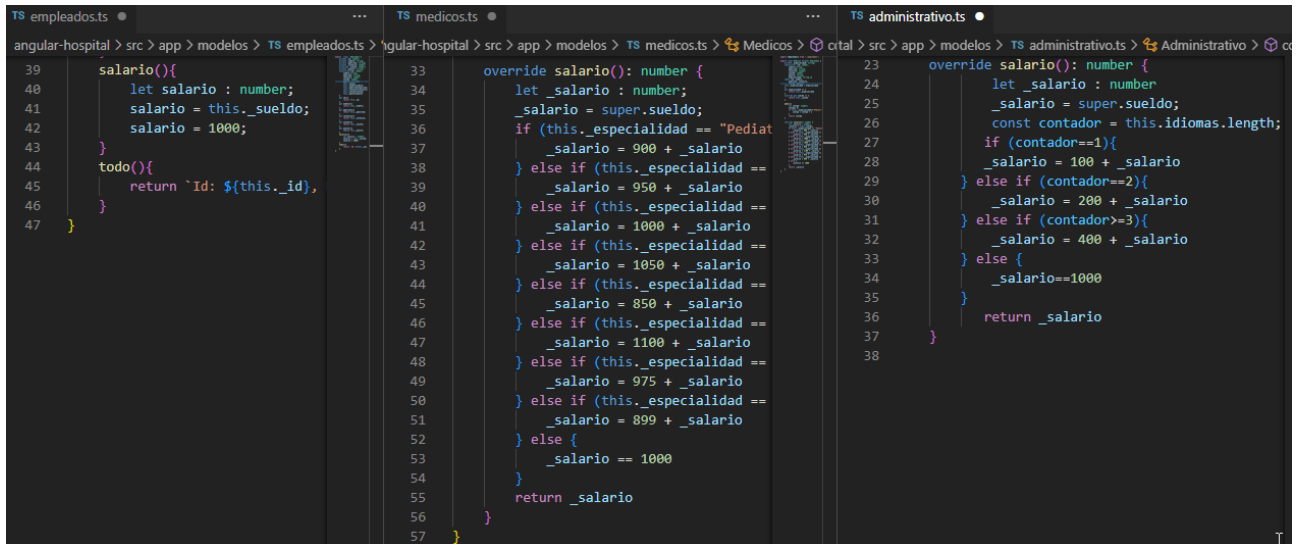
1 export class Pacientes {
2   private _id: number;
3   private _nombre: string;
4   private _apellido1: string;
5   private _apellido2: string;
6   private _edad: number;
7   private _dni: string;
8   private _telefono: number;
9   private _medico: string;
10  protected _urgencia: string;
11  private _tipo: string;
12  constructor(id: number,
13    nombre: string,
14    apellido1: string,
15    apellido2: string,
16    edad: number,
17    dni: string,
18    telefono: number,
19    medico: string,
20    urgencia: string,
21    tipo: string, ){
22    this._id=id;
23    this._nombre=nombre;
24    this._apellido1=apellido1;
25    this._apellido2=apellido2;
26    this._edad=edad;
27    this._dni=dni;
28    this._telefono=telefono;
29    this._medico=medico;
30    this._urgencia=urgencia;
31    this._tipo=tipo;
32  }
33 }

1 import {Pacientes} from "../pacientes";
2
3 export class Urgencias extends Pacientes {
4   private _pruebas: Array<string>;
5   constructor (id: number,
6     nombre:string,
7     apellido1: string,
8     apellido2: string,
9     edad: number,
10    dni: string,
11    telefono:number,
12    medico: string,
13    urgencia:string,
14    tipo: string,
15    pruebas: Array<string>){
16     super (id, nombre, apellido1,
17       apellido2, edad, dni,
18       telefono, medico, urgencia, tipo)
19     this._pruebas = pruebas
20   }
21   get pruebas(){
22     return this._pruebas
23   }
24   get urgencia(){
25     return this._urgencia
26   }
27 }

1 import {Pacientes} from "../pacientes";
2
3 export class Covid extends Pacientes {
4   private _test: string;
5   constructor (id: number,
6     nombre:string,
7     apellido1: string,
8     apellido2: string,
9     edad: number,
10    dni: string,
11    telefono:number,
12    medico: string,
13    urgencia:string,
14    tipo: string, test:
15    string){
16     super (id, nombre, apellido1,
17       apellido2, edad,
18       dni, telefono,
19       medico, urgencia, tipo)
20     this._test = test
21   }
22   get test(){
23     return this._test
24   }
25   get urgencia(){
26     return this._urgencia
27   }
28 }
```

## 6.2-Métodos

Se ha creado un método `salario()` en la clase `empleados`, mientras en gracias al polimorfismo en las clases `médicos` y `administrativos`, se **reescribe** el método ajustándolo a los campos exclusivos de dichas subclases.



The screenshot displays three TypeScript files in a code editor, illustrating the implementation of the `salario()` method across different classes. The first file, `TS empleados.ts`, shows the base method in the `empleados` class. The second file, `TS medicos.ts`, shows the `override salario()` method in the `Medicos` class, which uses a series of `if` statements to calculate the salary based on the `especialidad` (specialty). The third file, `TS administrativo.ts`, shows the `override salario()` method in the `Administrativo` class, which calculates the salary based on the `idiomas` (languages) property.

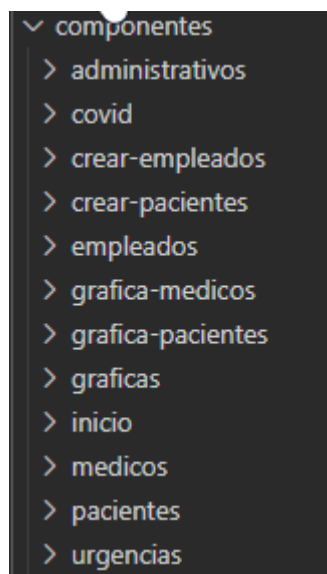
```
TS empleados.ts
39  salario(){
40    let salario : number;
41    salario = this._sueldo;
42    salario = 1000;
43  }
44  todo(){
45    return `Id: ${this._id},
46  }
47  }
```

```
TS medicos.ts
33  override salario(): number {
34    let _salario : number;
35    _salario = super.sueldo;
36    if (this._especialidad == "Pediat
37    _salario = 900 + _salario
38  } else if (this._especialidad ==
39    _salario = 950 + _salario
40  } else if (this._especialidad ==
41    _salario = 1000 + _salario
42  } else if (this._especialidad ==
43    _salario = 1050 + _salario
44  } else if (this._especialidad ==
45    _salario = 850 + _salario
46  } else if (this._especialidad ==
47    _salario = 1100 + _salario
48  } else if (this._especialidad ==
49    _salario = 975 + _salario
50  } else if (this._especialidad ==
51    _salario = 899 + _salario
52  } else {
53    _salario == 1000
54  }
55  return _salario
56  }
57  }
```

```
TS administrativo.ts
23  override salario(): number {
24    let _salario : number
25    _salario = super.sueldo;
26    const contador = this.idiomas.length;
27    if (contador==1){
28      _salario = 100 + _salario
29    } else if (contador==2){
30      _salario = 200 + _salario
31    } else if (contador>=3){
32      _salario = 400 + _salario
33    } else {
34      _salario==1000
35    }
36    return _salario
37  }
38  }
```

## 7-Componentes

Un componente representa cada una de las pantallas que se nos muestra en nuestro proyecto angular, cada uno de dichas páginas es un componente:



The screenshot shows a list of components in an Angular application. The list is titled "componentes" and includes the following items:

- > administrativos
- > covid
- > crear-empleados
- > crear-pacientes
- > empleados
- > grafica-medicos
- > grafica-pacientes
- > graficas
- > inicio
- > medicos
- > pacientes
- > urgencias

Tenemos varios componentes, un componente por cada lista que queremos mostrar, en este caso tenemos una por cada clase.(se vera en la exposición)

Por otro lado tenemos un componente inicio, que sera el componente que se ejecutará en primer lugar al arrancar la aplicación, más adelante veremos el por que.

Disponemos de dos componentes para crear nuevos documentos en nuestra base de datos, uno para empleados y otro para pacientes, además de poder eliminarlos, ya que existe un botón de borrado que emplea el método delete de la api-rest para eliminar los empleados que queramos.

Por último pero no menos importante, disponemos de dos componentes que emplea gráficos para representar datos de la base de datos empleando Highcharts, que nos permite gracias a su página disponer de ejemplos que podemos utilizar para alimentar con nuestros datos dichas gráficas.

Cada componente dispone de tres archivos principales:

Poniendo como ejemplo el componente graficas:

Dispone de graficas.component.css( donde se puede determinar el estilo que queramos que tengan las diferentes etiquetas y secciones de nuestro componente), graficas.component.html(es el código html que es lo que vamos a ver por pantalla, dicho archivo hará uso de los métodos que hallamos definido en el último archivo) y graficas.component.ts (en el se declaran todas las variables, todos las importaciones y todos los métodos que vamos a utilizar en el componente, el cual tirará de los servicios que definamos).

## **8.Problemas surgidos**

Durante el proyecto Heroku y github tubieron problemas de conexión por lo que hubo que subirlo directamente a Heroku(la res-api) por otro lado, de momento no se ha conseguido subir la aplicación angular a heroku, por lo que probablemente se hara la exposición en local.

Otro problema ha sido a la hora de editar documentos en la base de datos, permitia cambiar los datos pero no ver los datos que cambiabas, por lo que se ha desechado ese componente de editar-empleado.component y ver-empleados componente, así como sus versiones para pacientes.

No se ha podido hacer un login, se han seguido diferentes tutoriales y ejemplos pero a la hora de tocar el componente y servicio creado para el login, el `app.route.module`, aunque existia en el la ruta, dejaba de dar acceso a ese componente, por lo que se ha desechado.

Por últimos, en las gráficas se han usado datos fijos cogidos de la base de datos, se ha intentado de diferentes formas la implementación de métodos en estas gráficas, pero al parecer aunque no se detecte erro debe haber algún fallo en el método que no se ha encontrado y ese es el motivo por el cual se han usado datos fijos.

## **9. Conclusiones**

En este proyecto he comprendido mucho mejor los conceptos que se han desarrollado durante el curso y no terminaba de entender, sin embargo existen algunas cosas que desconozco por que no funcionan y en ocasiones me a desesperado realmente, sin embargo los conceptos y elementos básicos estudiados sobre angular y la api-rest han quedado claros.