Angular

Open Bank

Índice

1. Introducción técnica

El siguiente proyecto trata de una aplicación realizada en Angular junto a Bootstrap y Material. Dicho proyecto realiza de frontend de la aplicación, preparada para el uso de por parte de un usuario administrador, no de un cliente ni de cualquier empleado (como por ejemplo un limpiador).

Esta App se utiliza principalmente en el entorno de Heroku, aunque puede ser usada en local. Se priorizará el uso a través de la red por lo que no se darán demasiadas instrucciones sobre su funcionamiento en local.

La App realiza una conversación frontend-backend con una API de creación propia, que a su vez se conecta con una base de datos en MongoDB Atlas. Un servicio en la nube utilizado de manera gratuita (y por tanto limitada para el proyecto).

En este documento se explicarán las nociones básicas del funcionamiento de ambos, empezando por la API y terminando por la aplicación en Angular, y la conversación entre ambos.

2. Pasos previos

2.1. API REST

Para el uso de nuestro proyecto necesitaremos los siguientes paquetes.

2.1.1. Paquetes

```
  "devDependencies": {
    "@types/cors": "^2.8.12",
    "@types/express": "^4.17.13",
    "@types/morgan": "^1.9.3",
    "@types/node": "^17.0.5",
    "nodemon": "^2.0.15",
    "rimraf": "^3.0.2",
    "typescript": "^4.5.4"
  },
  "dependencies": {
    "cors": "^2.8.5",
    "express": "^4.17.2",
    "mongoose": "^6.1.4",
    "morgan": "^1.10.0"
  },
```

2.2. Angular

2.2.1. Paquetes

```
  "dependencies": {
    "@angular/animations": "~13.2.0",
    "@angular/cdk": "^13.2.2",
    "@angular/common": "~13.2.0",
    "@angular/compiler": "~13.2.0",
    "@angular/core": "~13.2.0",
    "@angular/forms": "~13.2.0",
    "@angular/material": "^13.2.2",
    "@angular/platform-browser": "~13.2.0",
    "@angular/platform-browser-dynamic": "~13.2.0",
    "@angular/router": "~13.2.0",
    "angular-highcharts": "^13.0.1",
    "bootstrap": "^5.1.3",
    "express": "^4.17.2",
    "highcharts": "^9.3.3",
    "highcharts-angular": "^3.0.0",
    "ngx-toastr": "^14.2.1",
    "rxjs": "~7.5.0",
    "tslib": "^2.3.0",
    "zone.js": "~0.11.4"
  },
  "devDependencies": {
    "@angular-devkit/build-angular": "~13.2.0",
```

```
    "@angular/cli": "~13.2.0",
    "@angular/compiler-cli": "~13.2.0",
    "@types/jasmine": "~3.10.0",
    "@types/node": "^12.11.1",
    "jasmine-core": "~4.0.0",
    "karma": "~6.3.0",
    "karma-chrome-launcher": "~3.1.0",
    "karma-coverage": "~2.1.0",
    "karma-jasmine": "~4.0.0",
    "karma-jasmine-html-reporter": "~1.7.0",
    "typescript": "~4.5.2"
  }
```

2.2.2. Server.js

Para poder compilar y utilizar nuestra app fuera de un entorno local y llevarlo a Heroku será necesario crear dicho archivo.

```
const express = require('express')
const app = express()
app.use(express.static('./dist/banco-app'));
app.get('/*', function(req, res) {
 res.sendFile('index.html', {root: 'dist/banco-app/'}
 );
});
const port = 3500;
app.listen(process.env.PORT || 3500), () => {
 console.log(`Example app listening at http://localhost:${port}`)
}
```

2.2.3. Importaciones

Para el uso de los distintos paquetes en Angular deberemos importarlos de la siguiente manera.

```
  imports: [
    BrowserModule,
    AppRoutingModule,
    ReactiveFormsModule,
    BrowserAnimationsModule,
    ToastrModule.forRoot(),
    HttpClientModule,
    HighchartsChartModule,
    MatFormFieldModule,
    MatInputModule,
    MatButtonModule
  ],
```

3. Clases

3.1. Empleados

Crearemos una clase padre para los empleados, dicha clase tendrá la siguiente estructura:

```
export abstract class Empleado {
  private _id: string;
  private _nombre: string;
  private _telefono: { movil: string, fijo: string | null };
  private _direccion: Array<direccion>;
  private _iban: string;
  private _sueldo: number;
  private _fecha: Date;
```

Y los siguientes métodos:

```
todo() {
      return `id: ${this._id}
              Nombre: ${this._nombre}
              Telefono/s: ${this._telefono}
              Direccion/es: ${this._direccion}
              IBAN: ${this._iban}
              Sueldo base: ${this._sueldo}
              Fecha de incorporacion: ${this._fecha}`
  }

  salario(): number {
      let salario: number
      let salarioBase: number = this._sueldo
      let fecha: Date = this._fecha
      let fecha1: Date = new Date('January 1, 2015')
      let fecha2: Date = new Date('January 1, 2000')

      if (fecha >= fecha1) {
          salario = salarioBase * 1.02
      } else {
          if (fecha > fecha2) {
              salario = salarioBase * 1.03
          } else {
              salario = salarioBase * 1.05
          }
      }

      return Math.round(salario)
  }
```

Esta clase contará con tres subclases:

- Directivo

  Las diferencias de esta clase con la padre son:

```
export class Directivo extends Empleado {
  private _nivel: string; //Niveles: A1, A2, B1, B2, C1, C2

salario(): number {
let salario: number = super.salario()
let nivel: string = this._nivel

if (nivel == "A1") {
  salario = salario + 110
} else if (nivel == "A2") {
  salario = salario + 130
} else if (nivel == "B1") {
  salario = salario + 220
} else if (nivel == "B2") {
  salario = salario + 240
} else if (nivel == "C1") {
  salario = salario + 350
} else if (nivel == "C2") {
  salario = salario + 500
}

return Math.round(salario)
}

```

- Comercial

  Las diferencias de esta clase con la padre son:

```
export class Comercial extends Empleado {
private _horas: number;

salario(): number {
let salario: number = super.salario()
let horas: number = this._horas
let horasExtra: number

if (horas < 160) {
  salario = salario - (salario * 0.01)

} else {
  horasExtra = 160 - horas
  salario = salario + (horasExtra * 12)
}

return Math.round(salario)
}

```

- Limpiador

  Las diferencias de esta clase con la padre son:

```
      export class Limpiador extends Empleado {
        private _empresa: string;

      salario(): number {
      let salario: number = super.salario()
      let empresa: string = this._empresa

      if (empresa == "Powlowski Group") {

      } else if (empresa == "Mitchell and Sons") {

      }

      return Math.round(salario)
      }

```

3.2. Clientes

Los clientes tendrán su propia clase padre. Será de la siguiente forma:

```
export abstract class Cliente {
  private _id: string;
  private _nombre: string;
  private _telefono: string;
  protected _direccion: { numero: string, calle: string };
  private _capital: number;
  private _ingresos: number;

renta(): number {
      let capital = this._capital
      let ganancias: number

      if (capital < 100000) {
          ganancias = capital - (capital * 0.97)
      } else if (capital < 500000) {
          ganancias = capital - (capital * 0.98)
      } else {
          ganancias = capital - (capital * 0.99)
      }

      return ganancias
  }
```

Contará con dos clases hijas:

- Personales

  Se diferencian en:

```
      export class Persona extends Cliente {
        private _comercial: string;

      renta(): number {
      let ganancias: number = super.renta()
      let ingresos: number = super.ingresos

      if (ingresos > 20000) {
        ganancias = ganancias - (ganancias * 0.01)
      }

      return ganancias
      }

```

- Empresariales

  Se diferencian en:

```
export class Empresa extends Cliente {
  private _plan: string;

  renta(): number {
  let ganacias: number = super.renta()
  let plan: string = this._plan

  if (plan == "1") {
    ganacias = ganacias + (ganacias * 0.01)
  } else if (plan == "2") {
    ganacias = ganacias + ((ganacias * 0.05) / 2)
  } else if (plan == "3") {
    ganacias = ganacias - 100000
  }

  return ganacias
  }

```

3.3. Prestamos

Adicionalmente crearemos otra clase prestamo en la App para ciertos cálculos posteriores.

```
export class Registro {
  public _idComercial: string;
  public _idCliente: string;
  public _capitalCliente: number;
  public _prestamo: number;
  public _interes: number;
  public _plazo: Date;
```

4. Rutas

En la API hemos creado distintas rutas con las que llamar a la base de datos y operar. Estas son las funciones utilizadas.

- Listar empleados

  Para listar los empleados usaremos el siguiente código:

```
    private listarEmpleados = async (req: Request, res: Response)
=> {
    await db.conectarBD()
      .then(async (mensaje) => {
        console.log(mensaje)
        const query = await Emp.find();
        res.json(query)
      })
      .catch((mensaje) => {
        res.send(mensaje)
      })

    db.desconectarBD()
    }

```

- Listar directivos

  Para listar los directivos usaremos el siguiente código:

```
    private listarDirectivos = async (req: Request, res: Response)
=> {
      await db.conectarBD()
        .then(async (mensaje) => {
          const valor = req.params.id
          console.log(mensaje)
          const query = await Emp.find({ _tipoObjeto: { $eq: "
Directivo" } });
          res.json(query)
        })
        .catch((mensaje) => {
          res.send(mensaje)
        })

      db.desconectarBD()
    }

```

- Listar limpiadores

  Para listar los limpiadores usaremos el siguiente código:

```
    private listarLimpiadores = async (req: Request, res: Response) => {
      await db.conectarBD()
        .then(async (mensaje) => {
          const valor = req.params.id
          console.log(mensaje)
          const query = await Emp.find({ _tipoObjeto: { $eq: "Limpiador" } });
          res.json(query)
        })
        .catch((mensaje) => {
          res.send(mensaje)
        })

      db.desconectarBD()
    }

```

- Listar comerciales

  Para listar los comerciales usaremos el siguiente código:

```
    private listarComerciales = async (req: Request, res: Response) => {
      await db.conectarBD()
        .then(async (mensaje) => {
          const valor = req.params.id
          console.log(mensaje)
          const query = await Emp.find({ _tipoObjeto: { $eq: "Comercial" } });
          res.json(query)
        })
        .catch((mensaje) => {
          res.send(mensaje)
        })

      db.desconectarBD()
    }

```

- Buscar empleado

  Para buscar por empleado usaremos el siguiente código:

```
  private buscarEmpleado = async (req: Request, res: Response) => {
    await db.conectarBD()
      .then(async (mensaje) => {
        const valor = req.params.id
        console.log(mensaje)
        const query = await Emp.aggregate(
          [{ $match: { _id: valor } }]
        );
        res.json(query)
      })
      .catch((mensaje) => {
        res.send(mensaje)
      })

    db.desconectarBD()
  }

```

- Listar empresas

  Para listar las empresas usaremos el siguiente código:

```
  private listarEmpresas = async (req: Request, res: Response) => {
    await db.conectarBD()
      .then(async (mensaje) => {
        const valor = req.params.id
        console.log(mensaje)
        const query = await Cli.find({ _tipoObjeto: { $eq: "Empresarial" } });
        res.json(query)
      })
      .catch((mensaje) => {
        res.send(mensaje)
      })

    db.desconectarBD()
  }

```

- Listar personas

  Para listar las personas usaremos el siguiente código:

```
    private listarPersonas = async (req: Request, res: Response) => {
      await db.conectarBD()
        .then(async (mensaje) => {
          const valor = req.params.id
          console.log(mensaje)
          const query = await Cli.find({ _tipoObjeto: { $eq: "Personal" } });
          res.json(query)

        })
        .catch((mensaje) => {
          res.send(mensaje)
        })

      db.desconectarBD()
    }

```

- Buscar cliente

  Para buscar por cliente usaremos el siguiente código:

```
    private buscarClientes = async (req: Request, res: Response) => {
      await db.conectarBD()
        .then(async (mensaje) => {
          const valor = req.params.id
          console.log(mensaje)
          const query = await Cli.aggregate(
            [{ $match: { _id: valor } }]
          );
          res.json(query)
        })
        .catch((mensaje) => {
          res.send(mensaje)
        })

      db.desconectarBD()
    }

```

- Listar prestamos

  Para listar los prestamos usaremos el siguiente código:

```
    private listarPrestamos = async (req: Request, res: Response)
=> {
      await db.conectarBD()
        .then(async (mensaje) => {
          console.log(mensaje)
          const query = await Reg.find();
          res.json(query)

        })
        .catch((mensaje) => {
          res.send(mensaje)
        })

      db.desconectarBD()
    }

```

- Registrar persona

  Para registrar las personas usaremos el siguiente código:

```
    private registrarPersona = async (req: Request, res: Response)
=> {
      const { id, nombre, telefono, calle, numero, capital,
ingresos, comercial } = req.body
      await db.conectarBD()
      const dSchema = {
        _id: id,
        _tipoObjeto: "Personal",
        _nombre: nombre,
        _telefono: telefono,
        _direccion: { calle: calle, numero: numero },
        _capital: capital,
        _ingresos: ingresos,
        _comercial: comercial,
      }
      console.log(dSchema)
      const oSchema = new Cli(dSchema)
      await oSchema.save()
        .then((doc: any) => res.send('Has guardado el archivo:\n' +
 doc))
        .catch((err: any) => res.send('Error: ' + err))

      db.desconectarBD()
    }

```

- Registrar empresariales

  Para registrar las empresas usaremos el siguiente código:

```
    private registrarEmpresa = async (req: Request, res: Response) => {
      const { id, nombre, telefono, calle, numero, capital, ingresos, plan } = req.body
      await db.conectarBD()
      const dSchema = {
        _id: id,
        _tipoObjeto: "Empresarial",
        _nombre: nombre,
        _telefono: telefono,
        _direccion: { calle: calle, numero: numero },
        _capital: capital,
        _ingresos: ingresos,
        _plan: plan,
      }
      const oSchema = new Cli(dSchema)
      await oSchema.save()
        .then((doc: any) => res.send('Has guardado el archivo:\n' + doc))
        .catch((err: any) => res.send('Error: ' + err))

      db.desconectarBD()
    }

```

- Crear prestamo

  Para crear los prestamos usaremos el siguiente código:

```
    private crearRegistro = async (req: Request, res: Response) =>
{
      const { idComercial, idCliente, capitalCliente, prestamo } =
req.body
      let interes: number
      let plazo: Date
      let fecha: Date = new Date()
      if (prestamo < 10000) {
        interes = 0.05
        fecha.setMonth(fecha.getMonth() + 6)
        plazo = fecha

      } else if (prestamo < 50000) {
        interes = 0.07
        fecha.setFullYear(fecha.getFullYear() + 2)
        plazo = fecha

      } else {
        interes = 0.09
        fecha.setFullYear(fecha.getFullYear() + 10)
        plazo = fecha
      }

      await db.conectarBD()
      const dSchema = {
        _idComercial: idComercial,
        _idCliente: idCliente,
        _capitalCliente: capitalCliente,
        _prestamo: prestamo,
        _interes: interes,
        _plazo: plazo,
      }
      const oSchema = new Reg(dSchema)
      await oSchema.save()
        .then((doc: any) => res.send('Has guardado el archivo:\n' +
 doc))
        .catch((err: any) => res.send('Error: ' + err))

      db.desconectarBD()
    }

```

- Actualizar cliente

  Para actualizar los clientes usaremos el siguiente código:

```
    private actualizarCliente = async (req: Request, res: Response) => {
      await db.conectarBD()
      const id = req.params.id
      const { nombre, telefono, calle, numero, capital, ingresos, comercial, plan } = req.body
      await Cli.findOneAndUpdate(
        { _id: id },
        {
          _nombre: nombre,
          _telefono: telefono,
          _direccion: { calle: calle, numero: numero },
          _capital: capital,
          _ingresos: ingresos,
          _comercial: comercial,
          _plan: plan
        },
        {
          new: true,
          runValidators: true
        }
      )
        .then((doc: any) => res.send('Has guardado el archivo:\n' + doc))
        .catch((err: any) => res.send('Error: ' + err))

      await db.desconectarBD()
    }

```

- Eliminar cliente

  Para eliminar los clientes usaremos el siguiente código:

```
    private eliminarCliente = async (req: Request, res: Response) => {
      await db.conectarBD()

      const id = req.params.id
      await Cli.findOneAndDelete({ _id: id })
        .then((doc: any) => res.send('Eliminado correctamente.'))
        .catch((err: any) => res.send('Error: ' + err))

      await db.desconectarBD()
    }

```

- Calcular renta

  Para calcular la renta usaremos el siguiente código:

```
    private calcularRenta = async (req: Request, res: Response) =>
{
      console.log('test')
      await db.conectarBD()
        .then(async (mensaje) => {
          let tmpCliente: Cliente
          let dCliente: tCliente2
          let arrayRenta: Array<tRenta> = []
          const query = await Cli.find({})

          for (dCliente of query) {
            if (dCliente._tipoObjeto == "Empresarial") {
              tmpCliente = new Empresa(dCliente._id,
                dCliente._nombre,
                dCliente._telefono,
                dCliente._direccion,
                dCliente._capital,
                dCliente._ingresos,
                dCliente._plan)

              console.log(tmpCliente)

              let rentaT: number = 0
              rentaT = tmpCliente.renta()

              let dRenta: tRenta = {
                _id: null,
                _nombre: null,
                _renta: null
              }

              dRenta._id = tmpCliente.id
              dRenta._nombre = tmpCliente.nombre
              dRenta._renta = rentaT
              arrayRenta.push(dRenta)
            }
          }

          res.json(arrayRenta)
        })
        .catch((mensaje) => {
          console.log('test')
          res.send(mensaje)
        })

      await db.desconectarBD()
    }

```

Finalmente las rutas quedarán así:

```
misRutas() {

      this._router.get('/', this.index)

      this._router.get('/empleados', this.listarEmpleados)
      this._router.get('/empleados/directivo', this.listarDirectivos)
      this._router.get('/empleados/limpiador', this.listarLimpiadores)
      this._router.get('/empleados/comercial', this.listarComerciales)
      this._router.get('/empleados/:id', this.buscarEmpleado)
      this._router.get('/clientes/persona', this.listarPersonas)
      this._router.get('/clientes/renta', this.calcularRenta)
      this._router.get('/clientes/empresa', this.listarEmpresas)
      this._router.get('/clientes/:id', this.buscarClientes)
      this._router.get('/prestamos', this.listarPrestamos)

      this._router.post('/clientes/registrarPersona', this.
 registrarPersona)
      this._router.post('/clientes/registrarEmpresa', this.
 registrarEmpresa)
      this._router.post('/registro', this.crearRegistro)

      this._router.put('/clientes/actualizar/:id', this.
 actualizarCliente)

      this._router.delete('/clientes/eliminar/:id', this.eliminarCliente
 )

      this._router.get('/empleados/salario/:id', this.calcularSalario)
      this._router.get('/ganancia/:id', this.mediaGanancia)
  }
```

5. Servicios

Para conectar Angular con nuestra API deberemos hacer uso de los servicios. En este caso tendremos dos, uno que conecte con la parte de los clientes y otro la de los empleados.

Esta será la estructura de los servicios:

- ClientesService

```
export class ClienteService {
  url = 'https://api-rest-banco.herokuapp.com/clientes'
  url2 = 'https://api-rest-banco.herokuapp.com'

  constructor(private http: HttpClient) { }

  getPersonas(): Observable<any> {
    return this.http.get(this.url + '/persona')
  }

  getEmpresa(): Observable<any> {
    return this.http.get(this.url + '/empresa')
  }

  eliminarCliente(id: string): Observable<any> {
    return this.http.delete(this.url + '/eliminar/' + id)
  }

  registrarPersona(persona: tPersona): Observable<any> {
    return this.http.post(this.url + '/registrarPersona', persona
)
  }

  registrarEmpresa(empresa: tEmpresa): Observable<any> {
    return this.http.post(this.url + '/registrarEmpresa', empresa
)
  }

  obtenerCliente(id: string): Observable<any> {
    return this.http.get(this.url + '/' + id)
  }

  editarCliente(id: string, persona: tPersona | tEmpresa):
Observable<any> {
    return this.http.put(this.url + '/actualizar/' + id, persona)
  }

  getRenta(): Observable<any> {
    return this.http.get(this.url + '/renta/')
  }

  crearPrestamo(registro: tRegistro): Observable<any> {
    return this.http.post(this.url2 + '/registro', registro, {
responseType: 'json' })
  }
  getPrestamos(): Observable<any> {
```

```
        return this.http.get(this.url2 + '/prestamos')
    }

```

- EmpleadosService

```
    export class EmpleadosService {
url = 'https://api-rest-banco.herokuapp.com/empleados'

constructor(private http: HttpClient) { }

getEmpleados(): Observable<any> {
  return this.http.get(this.url)
}
getDirectivos(): Observable<any> {
  return this.http.get(this.url + '/directivo')
}
getComerciales(): Observable<any> {
  return this.http.get(this.url + '/comercial')
}
getLimpiadores(): Observable<any> {
  return this.http.get(this.url + '/limpiador')
}

obtenerEmpleado(id: string): Observable<any> {
  return this.http.get(this.url + '/' + id)
}

eliminarEmpleado(id: string): Observable<any> {
  return this.http.delete(this.url + '/eliminar/' + id)
}

registrarDirectivo(directivo: tDirectivo): Observable<any> {
  return this.http.post(this.url + '/registrarDirectivo', directivo
 )
}
registrarComercial(comercial: tComercial): Observable<any> {
  return this.http.post(this.url + '/registrarComercial', comercial
 )
}
registrarLimpiador(limpiador: tLimpiador): Observable<any> {
  return this.http.post(this.url + '/registrarDirectivo', limpiador
 )
}

editarEmpleado(id: string, empleado: tComercial | tDirectivo |
 tLimpiador): Observable<any> {
  return this.http.put(this.url + '/actualizar/' + id, empleado)
}

```

6. Componentes

Para el uso de Angular necesitaremos distintas páginas, cada una de estas será un componente.

Tendremos los siguientes:

- Login

  Para el componente de **Login** haremos uso de Angular Material, que nos permite un mayor manejo de los estilos con las plantillas definidas. En el html tendremos un formulario para introducir el usuario y la contraseña.

  Haremos uso del paquete de ReactiveForms que nos permitirá un mayor control de nuestros formularios. Recibiremos cada input con un nombre y lo enviaremos a unos datos del formulario para validarlo. Después, mediante texto plano, separaremos el Login en dos partes.

```
      entrar() {
        const usuario = this.loginForm.value.usuario
        const contrasena = this.loginForm.value.contrasena
        if (usuario == "empleado" && contrasena == "1234") {
        this.router.navigate(['/home'])
        } else if (usuario == "admin" && contrasena == "admin") {
        this.router.navigate(['/empleados'])
        } else {
        this.toastr.error('El usuario o la contrasena son incorrectos', 'Login incorrecto');
        }}

```

- Listar empleados

  La ruta a la que nos llevará el usuario de admin será un listado de todos los empleados. Como es lógico, tan solo un usuario administrador debesería ser capaz de ver esto.

  En el html crearemos tres botones distintos que llamarán cada uno a una función distinta. Por defecto vendrá seleccionado y activado el primer botón, en él listará los empleados directivos de la siguiente forma:

```
      listarDirectivos() {
        let dni = ""
        dni = this.buscarForm.get('dni')?.value
        this.titulo = "Nivel"

        if (dni == "") {
          this._empleadoService.getDirectivos().subscribe(data => {
          this.listEmpleado = data
          })
        } else {
          this._empleadoService.obtenerEmpleado(dni).subscribe(data => {
          this.listEmpleado = data
          })
        }}

```

Los otros dos botones activarán las funciones de **listarComerciales** y **listarLimpiadores**, que harán lo mismo pero con las respectivas subclases.

Uno de los valores de la tabla en el HTML es variable, ya que cambiará en función de quien se esté representando. Dicho valor se representará como **titulo** , cambiando la variable titulo en cada función por el nombre que le corresponda.

- Home

  El componente **Home** será un componente sencillo en HTML que tendrá un poco de texto para dar una breve introducción a la aplicación.

- Listar personas

  Dicho componente será el encargado de listar todos los clientes de la subclase persona. El código será el siguiente:

```
1       listarPersonas() {
2         let dni = ""
3         dni = this.buscarForm.get('dni')?.value
4
5         if (dni == "") {
6           this._clientesService.getPersonas().subscribe(data => {
7           this.listPersonas = data
8
9           })
10        } else {
11          this._clientesService.obtenerCliente(dni).subscribe(data =>
    {
12          this.listPersonas = data
13          })
14        }
15        }
16
```

La variable **dni** empezará vacía, si se acaba llenando en vez de buscar en toda la base de datos buscará únicamente el id que introduzcamos. Este id será recogido en un formulario al comienzo de la página y nos permitirá hacer búsquedas en nuestra tabla.

También tendremos implementada la función de borrado de la siguiente forma:

```
1       eliminarPersona(id: any) {
2         this.listPersonas = this.listPersonas.filter((h) => h._id !==
     id)
3         this._clientesService.eliminarCliente(id).subscribe()
4         this.toastr.error('El cliente fue eliminado correctamente', '
    Cliente eliminado')
5         }
6
```

Esta se activará con un botón en forma de papelera que tendremos a la derecha de cada documento. Nos borrará tanto de la base de datos como del array de documentos que estamos visualizando.

- Registrar personas

  El siguiente componente será el de registrar clientes personales Se accederá a él mediante un botón a la derecha del anterior componente. El HTML será un formulario que mediante ReactiveForms enviará los datos introducidos. Nuestro código será el siguiente:

```
    crearPersona() {
      const CLIENTE: tPersona = {
        id: this.clienteForm.get('dni')?.value,
        nombre: this.clienteForm.get('nombre')?.value,
        telefono: this.clienteForm.get('telefono')?.value,
        numero: this.clienteForm.get('numero')?.value,
        calle: this.clienteForm.get('calle')?.value,
        capital: this.clienteForm.get('capital')?.value,
        ingresos: this.clienteForm.get('ingresos')?.value,
        comercial: this.clienteForm.get('comercial')?.value,
      }
      if (this.id !== null) {
        this._clienteService.editarCliente(this.id, CLIENTE).
subscribe()
        this.toastr.info('El cliente fue actualizado correctamente
', 'Cliente actualizado');
      } else {
        this._clienteService.registrarPersona(CLIENTE).subscribe()
        this.toastr.success('El cliente fue creado correctamente',
'Cliente creado');
        this.clienteForm.reset()
      }}

```

  Montaremos un objeto con lo que nos llegue del formulario y lo enviaremos a la API, para que cree dicho objeto. Pero antes comprobaremos si el campo id está vacio. Si está lleno significará que estamos editando un cliente (el que tenga dicho id).

  Para editar dicho cliente no queremos tener que volver a escribir todos los datos, por lo que captaremos los datos asociados a ese id y lo mostraremos en nuestra página:

```
    editarPersona() {
      if (this.id !== null) {
        this.titulo = "Editar cliente"
        this._clienteService.obtenerCliente(this.id).subscribe(data
 => {
        this.clienteForm.setValue({
          dni: data[0]._id,
          nombre: data[0]._nombre,
          telefono: data[0]._telefono,
          numero: data[0]._direccion.numero,
          calle: data[0]._direccion.calle,
          capital: data[0]._capital,
          ingresos: data[0]._ingresos,
          comercial: data[0]._comercial,
        })
        })
      }
    }

```

- Listar empresas

  Este componente funcionará de manera similar al de personas, ya que se trata de la otra subclase de cliente. Contará con las mismas funciones, solo que cambiará las rutas de la API de personales a empresariales.

- Registrar empresas

  Al igual que en **listar empresas**, este componente funcionará igual que **registrar personas** pero cambiando las rutas de la API.

- Estadísticas

  Este componente será el encargado de mostrarnos ciertas tablas que nos permitirán ver datos de una forma más visible. Utilizaremos el módulo de Highcharts que hemos tenido que instalar. En el HTML crearemos dos botones igual que antes, con uno activado por defecto. Esto nos permitirá dividir la tabla en dos partes, pudiendo aprovechar más recursos en menos espacio.

  Ambas tablas nos mostrarán las rentas de los dos tipos de clientes que tenemos.

  La de empresas hará uso de la API y sus cálculos, usando el método de **renta()** que tenemos en la clase de **Clientes**. Enviaremos los datos de sus respectivos nombres y rentas a la gráfica.

```
1       obtenerRentaEmpresa() {
2         this._clientesService.getRenta().subscribe((result: any) => {
3
4           this.listRenta = result.map((renta: any) => {
5           return new Renta(renta._id, renta._nombre, renta._renta)
6           })
7
8           const dataSeries = this.listRenta.map((x: Renta) => x.
    _renta)
9           const dataCategorias = this.listRenta.map((x: Renta) => x.
    _nombre)
10
11          this.chartOptions.title["text"] = "Ganancias de clientes
    empresariales"
12          this.chartOptions.series[0]["data"] = dataSeries
13          this.chartOptions.xAxis["categories"] = dataCategorias
14          this.chartOptions.series["name"] = "Empresas"
15
16          Highcharts.chart("renta", this.chartOptions)
17        })
18        }
19
```

Para los clientes personales hemos calculado su renta en la propia App, obligándonos a montar un objeto con ellos para poder hacer uso de sus métodos.

```
    obtenerRentaPersona() {
      this._clientesService.getPersonas().subscribe((data) => {
        let dCliente: tPersona2
        let tmpCliente: Cliente
        let renta: number
        this.arrayRenta = []

        this.listPersonas = data
        for (dCliente of this.listPersonas) {
        tmpCliente = new Persona(dCliente._id,
          dCliente._nombre,
          dCliente._telefono,
          dCliente._direccion,
          dCliente._capital,
          dCliente._ingresos,
          dCliente._comercial)

        renta = tmpCliente.renta()
        let dRenta: tRenta = {
          _id: null,
          _nombre: null,
          _renta: null
        }
        dRenta._id = tmpCliente._id
        dRenta._nombre = tmpCliente._nombre
        dRenta._renta = renta
        this.arrayRenta.push(dRenta)
        }
        this.listRenta = this.arrayRenta.map((renta: any) => {
        return new Renta(renta._id, renta._nombre, renta._renta)
        })

        const dataSeries = this.listRenta.map((x: Renta) => x._renta)
        const dataCategorias = this.listRenta.map((x: Renta) => x._nombre)

        this.chartOptions.title["text"] = "Ganancias de clientes personales"
        this.chartOptions.series[0]["data"] = dataSeries
        this.chartOptions.xAxis["categories"] = dataCategorias
        this.chartOptions.series["name"] = "Personas"

        Highcharts.chart("renta", this.chartOptions)

      })
      }

```

- Prestamos

  El último componente será el de los préstamos.

  En el HTML incluiremos tanto un formulario como una tabla que muestre datos. Crearemos una función que muestre todos los préstamos que se han realizado haciendo una llamada en la API.

```
      listarPrestamos() {
        this._clientesService.getPrestamos().subscribe(data => {
          this.listPrestamos = data
        })
        }

```

  Para crear los prestamos utilizaremos la siguiente función.

```
      crearPrestamo() {

        this.idCliente = this.prestamoForm.get('idCliente')?.value

        this._clientesService.obtenerCliente(this.idCliente).
  subscribe(data => {
          this.capital = data[0]._ingresos

          const PRESTAMO: tRegistro = {
          idComercial: this.prestamoForm.get('idEmpleado')?.value,
          idCliente: this.prestamoForm.get('idCliente')?.value,
          prestamo: this.prestamoForm.get('prestamo')?.value,
          capitalCliente: this.capital
          }
          console.log(PRESTAMO)
          this._clientesService.crearPrestamo(PRESTAMO).subscribe()
          this.toastr.success('El prestamo fue creada correctamente',
   'Prestamo creado');
          this.prestamoForm.reset()

        })
        }

```

  Para crear un préstamo la API debe recibir un id del Comercial, del cliente, la cantidad total de préstamo y cuanto capital posee dicho cliente. Utilizaremos la potencia de Angular y la API para quitarnos carga de cálculo mezclando ambos.

  Por formulario tan solo pediremos ambos id y el préstamo que se desea pedir. El propio Angular obtendrá el capital del cliente en función del id introducido y lo enviará a la API. Luego, la API calculará el plazo y el interés en función de los datos recibidos.

  Por último, mostraremos todos los registros creados.