Banco
API Rest

Índice

1. Instalación de dependencias

Para proceder con el proyecto deberemos instalar npm.

```
PS C:\miscosas\01-Estudios\01-SegAsir\SGBD\Segundo_trimestre\220110_api-rest> npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (220110_api-rest)
version: (1.0.0)
description: API Rest Banco
entry point: (index.js)
test command:
git repository:
keywords:
author: Emilio Sanchez
license: (ISC)
```

Ahora procederemos a la instalacion de TypeScript, sus dependencias y un paquete adicional conocido como **rimraf** que nos permitirá hacer limpiezas de directorios.

```
PS C:\miscosas\01-Estudios\01-SegAsir\SGBD\Segundo_trimestre\220110_api-rest> npm install -D typescript
npm notice created a lockfile as package-lock.json. You should commit this file.
npm WARN 220110_api-rest@1.0.0 No repository field.

+ typescript@4.5.4
added 1 package from 1 contributor and audited 1 package in 0.958s
found 0 vulnerabilities

PS C:\miscosas\01-Estudios\01-SegAsir\SGBD\Segundo_trimestre\220110_api-rest> npm install -D @types/node

+ @types/node@17.0.5
added 1 package from 42 contributors and audited 2 packages in 0.691s
found 0 vulnerabilities

PS C:\miscosas\01-Estudios\01-SegAsir\SGBD\Segundo_trimestre\220110_api-rest> npm install -D rimraf
npm WARN 220110_api-rest@1.0.0 No repository field.
+ rimraf@3.0.2
added 12 packages from 4 contributors and audited 14 packages in 1.766s

2 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

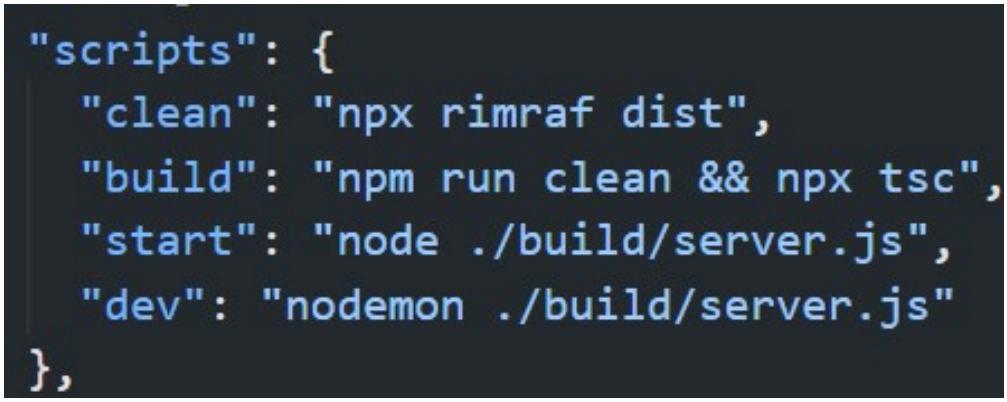
Deberemos instalar las dependencias necesarias para el uso de la base de datos junto a TypeScript.

```
PS C:\miscosas\01-Estudios\01-SegAsir\SGBD\Segundo_trimestre\220110_api-rest> npm install cors
npm WARN 220110_api-rest@1.0.0 No repository field.

+ cors@2.8.5
added 3 packages from 3 contributors and audited 17 packages in 0.746s
2 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

PS C:\miscosas\01-Estudios\01-SegAsir\SGBD\Segundo_trimestre\220110_api-rest> npm install express
npm WARN 220110_api-rest@1.0.0 No repository field.

+ express@4.17.2
added 48 packages from 37 contributors and audited 65 packages in 1.791s
4 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

PS C:\miscosas\01-Estudios\01-SegAsir\SGBD\Segundo_trimestre\220110_api-rest> npm install mongoose
npm WARN 220110_api-rest@1.0.0 No repository field.

+ mongoose@6.1.4
added 26 packages from 24 contributors and audited 92 packages in 7.697s

8 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

PS C:\miscosas\01-Estudios\01-SegAsir\SGBD\Segundo_trimestre\220110_api-rest> npm install morgan
npm WARN 220110_api-rest@1.0.0 No repository field.

+ morgan@1.10.0
added 5 packages from 3 contributors and audited 97 packages in 1.556s
```

Ahora crearemos el archivo **tsconfig.json** manualmente y añadiremos las siguientes lineas.

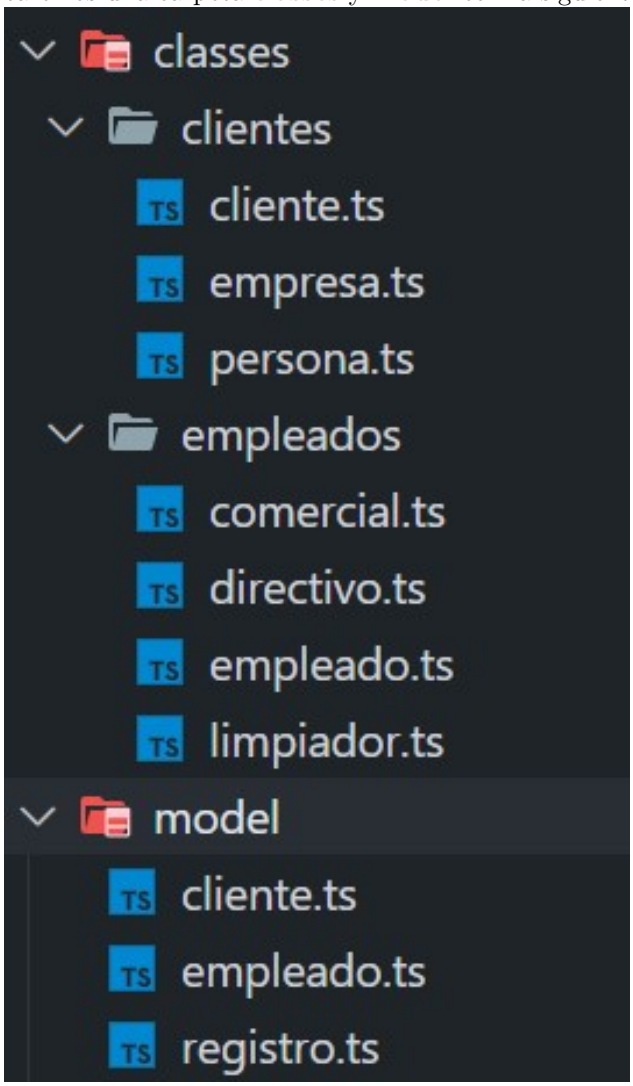
```
{
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs",
    "declaration": true,
    "outDir": "./build",
    "rootDir": "./src",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "skipDefaultLibCheck": true,
    "skipLibCheck": true
  }
}
```

Añadiremos los siguientes scripts a nuestro archivo **package.json** para poder compilar y ejecutar más rápido.



2. Clases de la base de datos

En la carpeta de **src** crearemos una carpeta **classes** y **model** con la siguiente estructura.



2.1. Clase de clientes

Esta es la clase padre de los clientes.

```
export abstract class Cliente {
    private _id: string; //DNI del cliente o CIF de la empresa
    private _nombre: string; //del cliente o la empresa
    private _telefono: string;
    protected _direccion: {numero: string, calle: string};
    private _capital: number;
    private _ingresos: number; //ingresos anuales

    constructor(id: string,
        nombre: string,
        telefono: string,
        direccion: {numero: string, calle: string},
        capital: number,
        ingresos: number) {
        this._id = id,
        this._nombre = nombre,
        this._telefono = telefono,
        this._direccion = direccion,
        this._capital = capital,
        this._ingresos = ingresos
    }
```

Dentro encontramos dos subclases:

- Personales

```
export class Persona extends Cliente {
    private _comercial: string; //comercial asignado a él

    constructor(id: string,
        nombre: string,
        telefono: string,
        direccion: {numero: string, calle: string},
        capital: number,
        ingresos: number,
        comercial: string) {
        super(id, nombre, telefono, direccion, capital, ingresos)
        this._comercial = comercial
    }
```

- Empresas

```
export class Empresa extends Cliente {
    private _plan: string; //plan financiero (1, 2 ó 3)

    constructor(id: string,
        nombre: string,
        telefono: string,
        direccion: {numero: string, calle: string},
        capital: number,
        ingresos: number,
        plan: string) {
        super(id, nombre, telefono, direccion, capital, ingresos)
        this._plan = plan
    }
```

2.2. Clase de empleados

Esta es la clase padre de los empleados.

```
export abstract class Empleado {
    private _id: string; //DNI del empleado
    private _nombre: string;
    private _telefono: { movil: string, fijo: string | null };
    private _direccion: Array<direccion>;
    private _iban: string; //Tipo string ya que no es un valor con el que se operará
    private _sueldo: number; //Sueldo base
    private _fecha: Date; //Fecha de incorporacion

    constructor(id: string,
        nombre: string,
        telefono: { movil: string, fijo: string | null },
        direccion: Array<direccion>,
        iban: string,
        sueldo: number,
        fecha: Date) {
        this._id = id;
        this._nombre = nombre;
        this._telefono = telefono;
        this._direccion = direccion;
        this._iban = iban;
        this._sueldo = sueldo;
        this._fecha = fecha;
    }
```

Dentro encontramos tres subclases:

- Directivos

```
export class Directivo extends Empleado {
    private _nivel: string; //Niveles: A1, A2, B1, B2, C1, C2
    constructor(id: string,
        nombre: string,
        telefono: { movil: string, fijo: string | null },
        direccion: Array<direccion>,
        iban: string,
        sueldo: number,
        fecha: Date,
        nivel: string) {
        super(id, nombre, telefono, direccion, iban, sueldo, fecha);
        this._nivel = nivel;
    }
```

- Limpiadores

```
export class Limpiador extends Empleado {
    private _empresa: string; //Subcontrata de los limpiadores
    constructor(id: string,
        nombre: string,
        telefono: { movil: string, fijo: string | null },
        direccion: Array<direccion>,
        iban: string,
        sueldo: number,
        fecha: Date,
        empresa: string) {
        super(id, nombre, telefono, direccion, iban, sueldo, fecha);
        this._empresa = empresa;
    }
```

- Comerciales

```
export class Comercial extends Empleado {
    private _horas: number;
    constructor(id: string,
        nombre: string,
        telefono: { movil: string, fijo: string | null },
        direccion: Array<direccion>,
        iban: string,
        sueldo: number,
        fecha: Date,
        horas: number) {
        super(id, nombre, telefono, direccion, iban, sueldo, fecha);
        this._horas = horas;
    }
```

2.3. Modelos

Necesisateremos también un modelo por cada clase padre.

- Cliente

  Creamos restricciones en los ingresos, que deberán de ser de un mínimo de cero y en el plan, que solo podrá contener 1, 2 ó 3.

```
const clienteSchema = new Schema({
  _id: {
    type: String,
    required: true,
    unique: true
  },
  _tipoObjeto: {
    type: String,
  },
  _nombre: {
    type: String,
  },
  _telefono: {
    type: String,
  },
  _direccion: {
    type: {numero: String, calle: String},
  },
  _capital: {
    type: Number
  },
  _ingresos: {
    type: Number,
    minimun: 0
  },
  _plan: {
    type: String,
    required: [ "1", "2", "3" ]
  },
  _comercial: {
    type: String,
  }
});
export const Cli = model("clientes", clienteSchema);
```

- Empleado

  En empleado no permitiremos que los sueldos sean negativos.

```
const empleadoSchema = new Schema({
  _id: {
        type: String,
        required: true,
        unique: true
    },
  _tipoObjeto: {
    type: String,
  },
  _nombre: {
    type: String,
  },
  _telefono: {
    type: { movil: String, fijo: String },
  },
  _direccion: {
    type: Array,
    default: []
  },
  _iban: {
    type: String,
  },
  _sueldo: {
    type: Number,
    minimun: 0
  },
  _fecha: {
    type: Date,
  },
  _nivel: {
    type: String,
  },
  _empresa: {
    type: String,
  },
  _horas: {
    type: Number,
  },
  _minimo: {
    type: Boolean, //mide si han cumplido el minimo de horas mensuales de trabajo
  }
});
export const Emp = model("empleados", empleadoSchema);
```

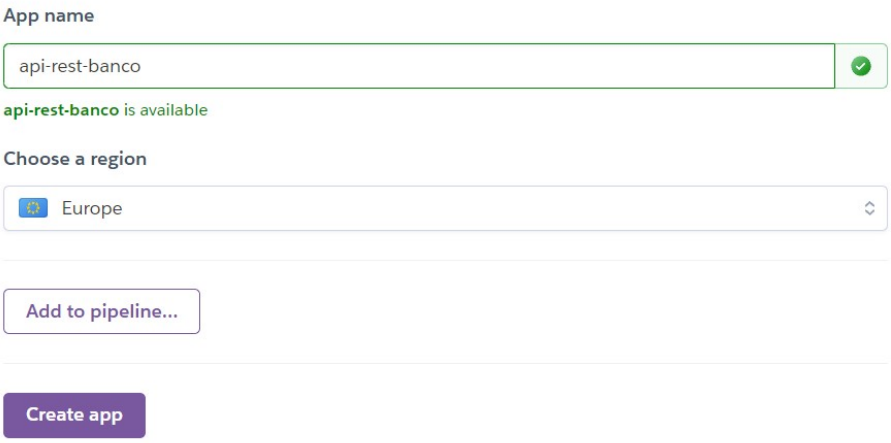
- Registro

```
const registroSchema = new Schema({
  _idComercial: {
      type: String
  },
  _idCliente: {
      type: String
  },
  _capitalCliente: {
      type: Number
  },
  _prestamo: {
      type: Number
  },
  _interes: {
      type: Number
  },
  _plazo: {
      type: Date
  }
});
export const Reg = model("registros", registroSchema);
```

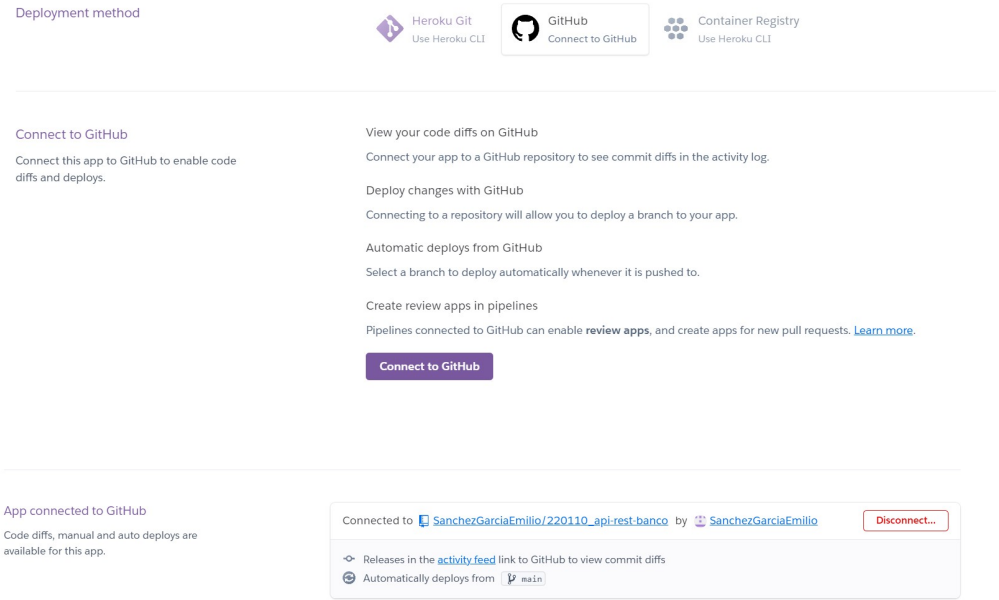
El modelo de registro corresponde a una colección nueva no incluida dentro de las clases que se crea al hacer préstamos dentro del banco.

3. Subida a Heroku

EL objetivo del proyecto será subirlo a Heroku para que la app pueda ser usada.
Para ello crearemos un proyecto en dicha página:



Ahora conectaremos la cuenta con la de GitHub y buscaremos nuestro proyecto.



4. Rutas

Dentro de nuestra API encontraremos distintas rutas para realizar búsquedas, subidas de datos, actualizaciones e incluso borrado. Estas son las rutas:

- Index

  Crearemos una página inicial donde tendremos un pequeño título y un link a nuestro GitHub, de esta forma podremos encontrar fácilmente la documentación con estas rutas, así sabremos cuales usar y como.

  Esta ruta será por defecto, se encontrará en / por lo que no será necesario buscarla. Usaremos una búsquedad de tipo GET.

```
private index = async (req: Request, res: Response) => {
    res.send(html)
}
```

- Búsqueda

  La segunda ruta será la de búsqueda. Para acceder a esta ruta deberemos añadir a nuestra aplicación: **/buscar/[id]** sustituyendo el **[id]** por la id del **comercial** que deseemos buscar. Usaremos una búsqueda de tipo GET.

  La aplicación nos devolverá un documento en formato json de nuestra búsqueda.

```
private buscarComercial = 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()
}
```

- Crear cliente

  La siguiente función será de tipo POST y en ella crearemos un cliente. Para acceder a esta ruta deberemos añadir **/crearCliente**

```
private crearCliente = async (req: Request, res: Response) => {
    const { id,  tipoObjeto,  nombre,  telefono, direccion, capital, ingresos} = req.body
    await db.conectarBD()
            const dSchema = {
                _id: id,
                _tipoObjeto: tipoObjeto,
                _nombre: nombre,
                _telefono: telefono,
                _direccion: direccion,
                _capital: capital,
                _ingresos: ingresos,
                _comercial: null,
            }
            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()
}
```

Para ello deberemos usar el body para enviar los datos, lo enviaremos desde un documento json (aunque no es la única opción).

- Actualizar cliente

  Usaremos un tipo PUT para esta opción. En ella actualizaremos el comercial asignado al cliente, ya que al crearlo no podemos asignarle uno.

  En la ruta deberemos añadir **/actualizar/[id]/[comercial]** sustituyendo el [**id**] por el DNI del cliente y el [**comercial**] por el DNI del comercial.

- Eliminar cliente

  Usaremos un tipo DELETE para esta opción. Eliminaremos un cliente que hayamos seleccionado en la URL.

  En la ruta deberemos añadir **/borrar/[id]** sustituyendo el **[id]** por el DNI del cliente que deseamos borrar.

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

    const id = req.params.id
    await Cli.findOneAndDelete({ _id: id })
    .then(() => console.log('\nEliminado Correctamente'))
    .catch((err: any) => console.log('\nError: ' + err))

    await db.desconectarBD()
}
```

- Calcular salario empleado

  Usaremos un tipo POST para esta opción. Buscaremos un comercial con su ID y nos devolverá su sueldo.

  En la ruta deberemos añadir **/salario/[id]** sustituyendo el **[id]** por el DNI del empleado que deseamos averiguar su salario.

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

    const id = req.params.id
    let tmpEmpleado: Empleado
    const query = await Emp.findOne({ _id: id })

    if (query._tipoObjeto == "Directivo") {
        tmpEmpleado = new Directivo(query._id,
            query._nombre,
            query._telefono,
            query._direccion,
            query._iban,
            query._sueldo,
            query._fecha,
            query._nivel)
        let salario = tmpEmpleado.salario().toString()
        res.send(salario)

    } else if (query._tipoObjeto == "Limpiador") {
        tmpEmpleado = new Limpiador(query._id,
            query._nombre,
            query._telefono,
            query._direccion,
            query._iban,
            query._sueldo,
            query._fecha,
            query._empresa)
        let salario = tmpEmpleado.salario().toString()
        res.send(salario)

    } else if (query._tipoObjeto == "Comercial") {
        tmpEmpleado = new Comercial(query._id,
            query._nombre,
            query._telefono,
            query._direccion,
            query._iban,
            query._sueldo,
            query._fecha,
            query._horas)
        let salario = tmpEmpleado.salario().toString()
        res.send(salario)
    }
    await db.desconectarBD()
}
```

- Calcular renta cliente

  De la misma forma que calculamos el salario, usaremos un tipo POST para esta opción. Buscaremos un cliente con su ID y nos devolverá su renta.

  En la ruta deberemos añadir **/renta/[id]** sustituyendo el **[id]** por el DNI del cliente que deseamos calcular su renta.

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

    const id = req.params.id
    let tmpCliente: Cliente
    const query = await Cli.findOne({ _id: id })

    if (query._tipoObjeto == "Personal") {
        tmpCliente = new Persona(query._id,
                                 query._nombre,
                                 query._telefono,
                                 query._direccion,
                                 query._capital,
                                 query._ingresos,
                                 query._comercial)
        let renta = tmpCliente.renta().toString()
        res.send(renta)

    } else if (query._tipoObjeto == "Empresarial") {
        tmpCliente = new Empresa(query._id,
                                 query._nombre,
                                 query._telefono,
                                 query._direccion,
                                 query._capital,
                                 query._ingresos,
                                 query._plan)
        let renta = tmpCliente.renta().toString()
        res.send(renta)
    }

    await db.desconectarBD()
}
```

- Calcular media de ganancia

  Usaremos un método POST para calcular la media de ganancia. Buscaremos un cliente por su ID y el comercial que tiene asignado. Luego calcularemos su renta y el sueldo gastado en el comercial asignado a él y nos devolverá un valor. Este puede ser negativo o positivo.

  En la ruta deberemos añadir **/ganancia/[id]** sustituyendo el **[id]** por el DNI del cliente sobre el que deseamos hacer la operación.

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

    const id = req.params.id
    let tmpPersona: Persona
    let tmpComercial: Comercial
    const query = await Cli.findOne({ _id: id })

    if (query._tipoObjeto == "Personal") {
        tmpPersona = new Persona(query._id,
                                 query._nombre,
                                 query._telefono,
                                 query._direccion,
                                 query._capital,
                                 query._ingresos,
                                 query._comercial)

        let query2 = await Emp.findOne({ _id: tmpPersona.comercial })
        tmpComercial = new Comercial(query2._id,
            query2._nombre,
            query2._telefono,
            query2._direccion,
            query2._iban,
            query2._sueldo,
            query2._fecha,
            query2._horas)

        let renta = tmpPersona.renta()
        let salario = tmpComercial.salario()
        let total = (renta - salario).toString()
        res.send(total)

    } else if (query._tipoObjeto == "Empresarial") {
        res.send('Solo se permiten clientes personales.')
    }
    await db.desconectarBD()
}
```

- Registro de préstamos

  Usaremos un método POST para crear un registro de préstamos. Desde la propia URL especificaremos el DNI del cliente que solicitará el préstamo y la cantidad elegida.

  En la ruta deberemos añadir **/prestamo/[id]/[cantidad]** sustituyendo el **[id]** por el DNI del cliente y **[cantidad]** por la cantidad elegida. Debe ser un número o dará error.

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

    const dniCli = req.params.id
    const prestamo = parseInt(req.params.prestamo)
    let tmpCliente: Cliente
    let dCliente: tCliente2
    let interes: number
    let fecha: Date = new Date()
    let plazo: Date
    let query: any = await Cli.find({ _id: dniCli })

    let sSchema: any
    let sSchemaReg: tRegistro = {
        _idComercial: null,
        _idCliente: null,
        _capitalCliente: null,
        _prestamo: null,
        _interes: null,
        _plazo: null,
    }

    for (dCliente of query) {
        if (dCliente._tipoObjeto == "Personal") {
            tmpCliente = new Persona(dCliente._id,
                                     dCliente._nombre,
                                     dCliente._telefono,
                                     dCliente._direccion,
                                     dCliente._capital,
                                     dCliente._ingresos,
                                     dCliente._comercial)

            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
            }

            sSchemaReg._idComercial = dCliente._comercial
            sSchemaReg._idCliente = dCliente._id
            sSchemaReg._capitalCliente = dCliente._capital
            sSchemaReg._prestamo = prestamo
            sSchemaReg._interes = interes
            sSchemaReg._plazo = plazo

            sSchema = new Reg(sSchemaReg)
            await sSchema.save()
            .then((doc: any) => res.send('Has guardado el archivo:\n' + doc))
            .catch((err: any) => res.send('Error: ' + err))
            }
    }

    await db.desconectarBD()
}
```

Finalmente tendremos todas las rutas de la siguiente forma:

```
misRutas() {
    this._router.get('/', this.index)
    this._router.get('/buscar/:id', this.buscarComercial)
    this._router.post('/crearCliente', this.crearCliente)
    this._router.put('/actualizar/:id/:comercial', this.actualizarCliente)
    this._router.delete('/borrar/:id', this.borrarCliente)
    this._router.post('/salario/:id', this.calcularSalario)
    this._router.post('/renta/:id', this.calcularRenta)
    this._router.post('/ganancia/:id', this.mediaGanancia)
    this._router.post('/prestamo/:id/:prestamo', this.crearPrestamo)
}
```