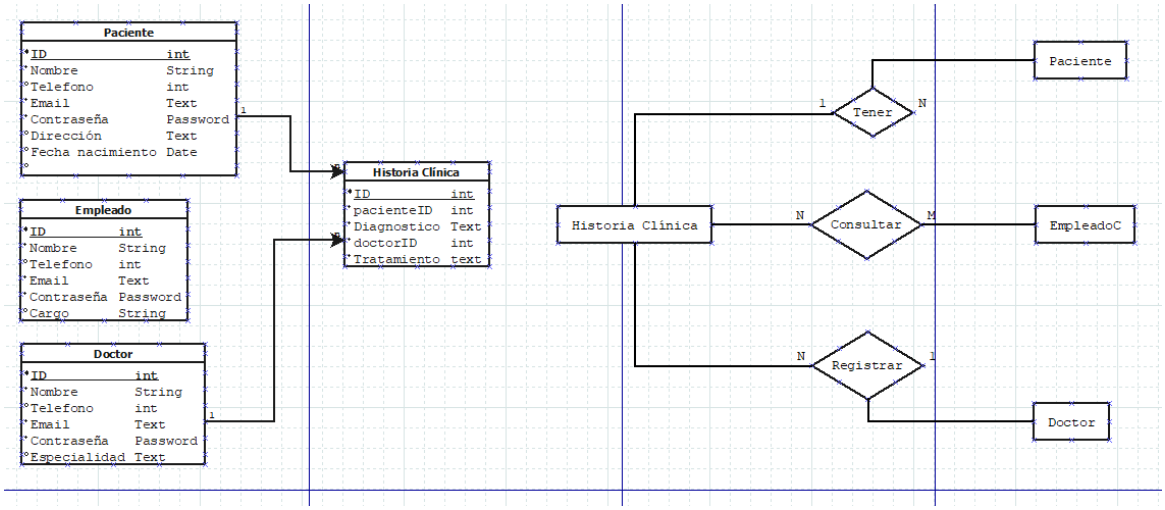
Documentación

El siguiente proyecto presenta un CRUD con Node, Sequelize y Express. Node se utiliza como entorno de ejecución para el servidor, Sequelize para las conexiones con la base de datos, Express para la definición de rutas y manejo de solicitudes HTTP. Para comenzar se utilizó el siguiente modelo y diagrama.



El modelo relacional y el diagrama E-R proporciona las cardinalidades necesarias para modelar eficientemente la próxima base de datos. Como se ve, la entidad/tabla historia clínica se relaciona con cada y una de las entidades/tablas.

Luego se pasó a crear la conexión con la base de datos.

```
const Sequelize = require ('sequelize');

const sequelize = new Sequelize( 'historia_clinica', 'postgres', ' password', {
    host: 'localhost',
    dialect: 'postgres'
});



module.exports = {
    sequelize};
```

Los ajustes que se usan para la conexión, sugerida por la documentación oficial de Sequelize, muestra una instancia de Sequelize donde se guardan los argumentos nombre de la base de datos, usuario de la db, y su contraseña. Además, proporciona donde estará escuchando el servidor y el tipo de base de datos. Luego de ahí se procede a crear los modelos.

```
const {DataTypes } = require('sequelize');
const {sequelize} = require('../Database/db.js');
const HistoriaClinica = require ('./HistoriaClinica.js')


const Paciente = sequelize.define('paciente', {

    id:{
        type: DataTypes.UUID,
        defaultValue: DataTypes.UUIDV4,
        primaryKey: true
    },
    nombre:{
        type: DataTypes.STRING
    },

    telefono:{
        type: DataTypes.INTEGER
    },

    email:{
        type: DataTypes.TEXT
    },

    contraseña: {
        type: DataTypes.TEXT
    },

    direccion:{
        type: DataTypes.TEXT
    },

    fecha_nacimiento: {
        type: DataTypes.DATE,
        defaultValue: DataTypes.NOW
    }
}, {
    timestamps: false,
});

```

El siguiente modelo presenta la tabla paciente con sus respectivos campos, los solicitados. Cada campo tiene asignado su tipo de dato para darle mayor seguridad al servidor. Los campos ID y fecha de nacimiento presentan dos valores por defectos, lo cual si el usuario no los registra, ya el servidor lo hace de forma automática.

```
Paciente.hasOne(HistoriaClinica,{
    foreingKey: 'pacienteID',
    sourceKey: 'id'
});

HistoriaClinica.belongsTo(Paciente, {
    foreingKey: 'pacienteID',
    targetId: 'id'
});


module.exports = Paciente;
```

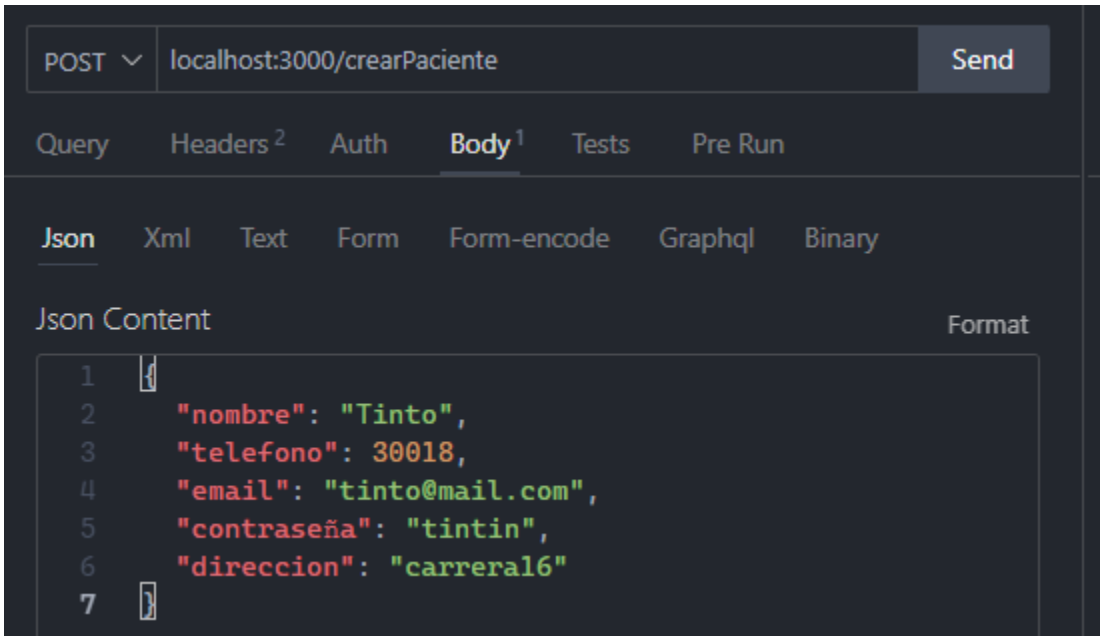
En la parte inferior del modelo paciente se realizan las relaciones, entre Paciente e Historia Clínica. Un paciente solo puede ser registrado por una historia clínica, y una historia clínica solo puede registrar un paciente. Pasamos a crear los controladores CRUD para el modelo paciente.

```
const Paciente = require ('../../Models/Paciente.js');


const mostrarPaciente = async (req, res) => {

    try {
        const traerPaciente =  await Paciente.findAll();

        res.json(traerPaciente);

    } catch (error) {
        console.log(error);
    }
}


module.exports = mostrarPaciente;
```

Para hacer el controlador de tipo GET para Paciente necesitamos traer el modelo del mismo. Y luego por medio de una función asíncrona, se utilizan los request y response para traer los pacientes que se encuentren en la base de datos por medio del método findAll de sequelize.

```
const crearPaciente = async (req, res) => {

    try {
        const {nombre, telefono, email, contraseña, direccion, fecha_nacimiento} = req.body;

        const nuevoPaciente = await Paciente.create({
        nombre,
        telefono,
        email,
        contraseña,
        direccion,
        fecha_nacimiento
    });
        res.json(nuevoPaciente);

    } catch (error) {
        console.log(error);
    }
}

module.exports = crearPaciente;
```

Para el método POST, se traen los campos del modelo paciente y los destructuran en un request de tipo body. Luego por medio del método créate de Sequelize se crea el nuevo paciente y recibe un response mostrándolo en pantalla.



Demostración

Para el tipo PUT se utilizan se traen los campos que se van a actualizar, en este caso la contraseña, y por medio del método findByPk se filtra el paciente que se va a actualizar.

```
const editarPaciente = async (req, res) => {

    const {id} = req.params;
    const {contraseña} = req.body;

    try {
        const paciente = await Paciente.findByPk(id);

        await paciente.update({
            contraseña
        });

        await paciente.save();

       res.send (paciente);


    } catch (error) {
        console.log(error);
    }

}
```

Se observa que se una el request params para atrapar el id, y el método update para actualizar el campo señalado. Pasamos a mostrar el método DELETE.

```
const borrarPaciente = async (req, res) => {

    const {id} = req.params;

    try {
        const paciente = await Paciente.findByPk(id);

        await paciente.destroy();

       res.send (paciente);


    } catch (error) {
        console.log(error);
    }

}
```

Para el tipo DELETE se usa casi la misma sintaxis solo que cambia el método destroy para eliminar el paciente con el id señalado. Cada controlador tiene su ruta para mostrar cada tabla.

```
router.get('/paciente', mostrarPaciente);
router.get('/doctor', mostrarDoctor);
router.get('/empleado', mostrarEmpleado);
router.get('/historia', mostrarHistoriaClinica);

router.post('/crearPaciente', crearPaciente);
router.post('/crearDoctor', crearDoctor);
router.post('/crearEmpleado', crearEmpleado);
router.post('/crearHistoriaClinica', crearHistoria);

router.put('/editarPaciente/:id', editarPaciente);
router.put('/editarDoctor/:id', editarDoctor);
router.put('/editarEmpleado/:id', editarEmpleado);

router.delete('/eliminarPaciente/:id',borrarPaciente);
router.delete('/eliminarDoctor/:id', borrarDoctor);
router.delete('/eliminarEmpleado/:id', borrarEmpleado);
```

Como se ve, cada controlador presenta su propia ruta, y la muestra – las de tipo get – en un modelo json en el localhost 3000.


```
[
  ▾ {
        id: "ef88ba24-6dd3-49d2-ac25-dc437932a776",
        nombre: "Luis",
        telefono: 30012,
        email: "luis@mail.com",
        "contraseña": "luisito12",
        direccion: "calle 1",
        fecha_nacimiento: "2023-02-26T23:14:43.963Z"
    },
  ▾ {
        id: "3f3a3a70-d553-49f2-8e5b-2bced0b01d6c",
        nombre: "Tinto",
        telefono: 30018,
        email: "tinto@mail.com",
        "contraseña": "tintin",
        direccion: "carrera16",
        fecha_nacimiento: "2023-02-27T01:13:44.055Z"
    }
]
```

Se muestra un ejemplo también de la tabla historia clínica para observar su relación por ID.


```
[
  {
    id: "0f0a17ab-a63c-4f80-9849-de0f64825d39",
    diagnostico: "Dolor de muela",
    tratamiento: "Tomar antibiótico",
    doctorId: "0eb8d482-7ee8-4216-93dd-227f04e94d3d",
    pacienteId: "ef88ba24-6dd3-49d2-ac25-dc437932a776"
  },
  {
    id: "f41f161a-4923-4d90-a32a-47024e46c5c8",
    diagnostico: "Dolor de zobaco",
    tratamiento: "Aplicar pomada",
    doctorId: "61b354b1-ef65-4d5b-9115-a02b0a18081f",
    pacienteId: "3f3a3a70-d553-49f2-8e5b-2bced0b01d6c"
  }
]
```


Véase que tiene la tabla historia clínica tiene dos llaves foráneas que surgen de las tablas paciente y doctor. Observe el id del pacienteId del primer registro concuerda con el id del primer registro del paciente.

Para finalizar, los demás modelos también presentan su tipo de modelaje y sus controladores. Se omitieron en la documentación para hacer más compacto y estética la documentación. Anexo tipo POST para las demás tablas como guía.

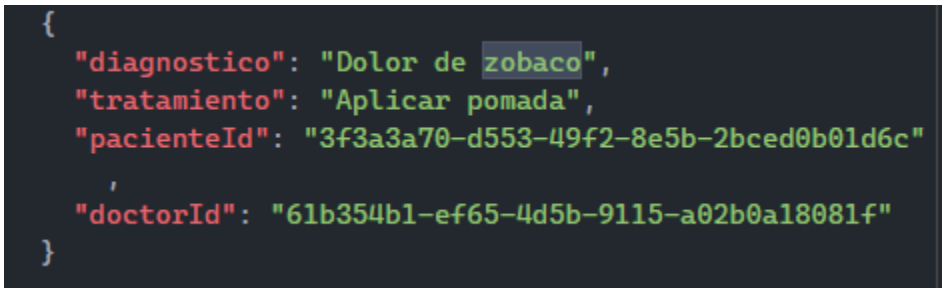
Json Content

```
{
  "nombre": "Pedro",
  "telefono": 30016,
  "email": "pedro@hospital.com",
  "contraseña": "pedrito",
  "especialidad": "Dentista"
}
```

Doctor

```
{
  "nombre": "Pablo",
  "telefono": 30017,
  "email": "pablo@hospital.com",
  "contraseña": "pablito",
  "cargo": "Logista"
}
```

Empleado



Historia clínica