PRUEBAS UNITARIAS

HERRAMIENTAS UTILIZADAS

MOCHA:

MOCHA es un framework de pruebas para Node JS que puede ejecutarse desde la consola o desde un navegador. Como su propia web indica, permite realizar pruebas asíncronas de manera sencilla y divertida. Al ejecutar los test, permite la presentación de informes flexibles y precisos.

En Mocha tenemos los siguientes conceptos:

- **describe()**, se utiliza para agrupar los tests o suite de tests.
- **context()**, es un alias para describe, solo provee otra forma de agrupación de tests .
- **it()**, representa un test.
- **before()**, la función dentro de before se va a ejecutar antes del primer test dentro del describe o context.
- **after()**, la función dentro de after se va a ejecutar después del último test dentro del describe o context.
- **beforeEach()**, la función dentro de beforeEach se va a ejecutar antes de cada test dentro del describe o context.
- **afterEach()**, la función dentro de afterEach se va a ejecutar después de cada test dentro del describe o context.

SUPERTEST:

SuperTest

HTTP assertions made easy via superagent.

Supertest es un módulo de Node.js con interface Fluent API que nos provee un nivel alto de abstracción para probar servicios http.

Por debajo utiliza superagent como cliente http.

Lo que aporta supertest son las aserciones vía Fluent API.
Como aserciones tenemos diferentes sobrecargas de la función expect:

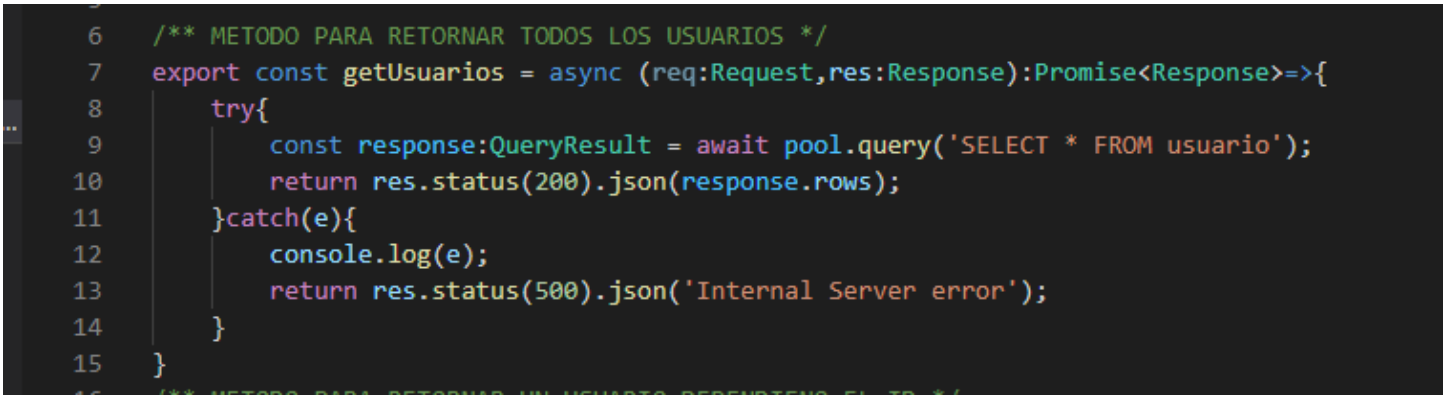
- **expect(status[, fn])**, verifica el código de estado de la respuesta.
- **expect(status, body[, fn])**, verifica el código de estado de la respuesta y el body.
- **expect(body[, fn])**, verifica el body con un string, expresión regular o también un objeto parseado.
- **expect(field, value[, fn])**, verifica una cabecera con un string o expresión regular.
- **expect(function(res) {})**, función de verificación personalizada, recibe el objeto de respuesta. Si la respuesta esta ok debería devolver false o no devolver nada. Si la verificación falla debemos devolver un error o un string especificando porque ha fallado la verificación.

INSTALACION DE MOCHA Y SUPERTEST

$ npm install mocha supertest –save

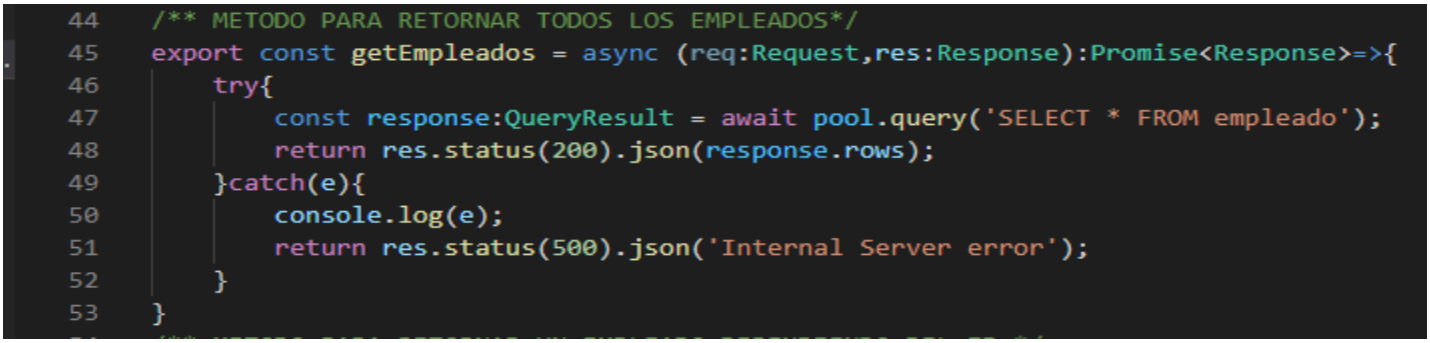
Código que se utilizó para realizar el test

Bloque de código que implementa la función de retornar todos los usuarios en la base de datos, si todo es correcto envía un ***estado 200*** de respuesta.



```
/** METODO PARA RETORNAR TODOS LOS USUARIOS */
export const getUsuarios = async (req:Request,res:Response):Promise<Response>=>{
    try{
        const response:QueryResult = await pool.query('SELECT * FROM usuario');
        return res.status(200).json(response.rows);
    }catch(e){
        console.log(e);
        return res.status(500).json('Internal Server error');
    }
}
```

Bloque de código que implementa la función de retornar todos los empleados en la base de datos, si todo es correcto envía un ***estado 200*** de respuesta.



```
/** METODO PARA RETORNAR TODOS LOS EMPLEADOS*/
export const getEmpleados = async (req:Request,res:Response):Promise<Response>=>{
    try{
        const response:QueryResult = await pool.query('SELECT * FROM empleado');
        return res.status(200).json(response.rows);
    }catch(e){
        console.log(e);
        return res.status(500).json('Internal Server error');
    }
}
```

Bloque de código que implementa la función de crear un reporte en la base de datos, si el cuerpo que se envía a través de un endpoint no cumple la estructura adecuada el servidor responderá con un ***estado 400*** y un ***mensaje de "No se creó el reporte"***;

```
/** GUARDAR REPORTE */
export const crearReporte = async (req:Request,res:Response)=>{
    try{
    const {zona,fechaReporte,horaReporte,fechaProblema,horaProblema,descripcion,idTipoProblema,idUsuario} = r
    const response:QueryResult = await pool.query('INSERT INTO reporte (zona,fechaReporte,horaReporte,fechaPr
                                                  +'VALUES($1,$2,$3,$4,$5,$6,$7,$8)',
    [zona,fechaReporte,horaReporte,fechaProblema,horaProblema,descripcion,idTipoProblema,idUsuario]);
    const retornar = await pool.query('SELECT MAX(idReporte) as idReporte FROM reporte');
    //console.log(retornar.rows[0].idreporte);
    const idReporte = retornar.rows[0].idreporte;
    return res.json({
        idReporte:idReporte,
        estado:true});
    }catch(e){
        return res.status(400).json("No se creo el reporte");
    }

}
```

Bloque de código que implementa la función de crear un mensaje en la base de datos, si el cuerpo que se envía a través de un endpoint no cumple la estructura adecuada el servidor responderá con un ***estado 400*** y un ***mensaje de "No se creó el mensaje"***;

```
239 export const crearMensaje = async(req:Request,res:Response)=>{
240     try{
241         const{descripcion,idReporte,idEmpleado} = req.body;
242         const response:QueryResult = await pool.query('INSERT INTO mensaje(descripcion,idReporte,idEmpleado)
243                                                        [descripcion,idReporte,idEmpleado]);
244         return res.json({
245             message:"Mensaje Creado",
246             body:{
247                 mensaje:{
248                     descripcion,idReporte,idEmpleado
249                 }
250             }
251         });
252     }catch(e){
253         return res.status(400).json("No se creo el mensaje");
254     }
255
256
257 }
```

Implementando los test

Como primer paso creamos una carpeta dentro de nuestro proyecto con nombre **Test** y en esta carpeta **Test** creamos un archivo ***test.js***.

Contenido de ***test.js***

Requerimos nuestro modulo Supertest y nuestra app para acceder a nuestros métodos a testear.

```
Test > JS test.js > ...
1   const request = require("supertest");
2
3   const app = require("../dist/index");
4
```

Métodos para testear los endpoints de tipo post para retornar usuarios y empleados.

```
/**
 * Testing endpoint get que devuelve todos los usuarios
 */
describe("GET /usuarios", () => {
  it("Responde con un json que contiene todos los usuarios", (done) => {
    request(app)
      .get("/usuarios")
      .set("Accept", "application/json")
      .expect("Content-Type", /json/)
      .expect(200, done);
  });
});

/**
 * Testing endpoint get que devuelve todos los empleados
 */
describe("GET /empleados", () => {
    it("Responde con un json que contiene todos los usuarios", (done) => {
      request(app)
        .get("/empleados")
        .set("Accept", "application/json")
        .expect("Content-Type", /json/)
        .expect(200, done);
    });


  });
```

Explicando el código

La función describe() se utiliza para agrupar los test y a la cual le pasamos una descripción del método a testear.

La función it() representa un test al cual le podemos pasar una descripción, dentro de este método requerimos nuestra app con request(app) y especificamos el tipo de endpoint a testear .get(“/usuarios”) con su respectiva url, además de especificar unos encabezados con .set(“Accept”,”aplication/json”), además verificamos las distintas respuestas que esperamos del endpoint con expect(“Content-Type”,/json/) con esto especificamos que esperamos un json, y .expect(200,done) especificamos que esperamos una respuesta 200 del endpoint, esto basado en las respuestas que retorna nuestro endpoint “/usuarios” el cual implementamos anteriormente, si esto se cumple nuestro test será exitosos en todo caso el test no pasará. Esto se hace tanto como para el enpoint de usuarios como el de empleados.

Siguientes test aplicados a los enpoint para almacenar reportes y usuario.

```
/**
 * Testing endpoint post para un reporte
 */
 describe("POST /reporte",() =>{
   it("Responder con un 400 si no se creo el reporte", (done) => {
        const reporte = {
            zona:"",
            fechaReporte:"",
            horaReporte:"",
            fechaProblema:"",
            horaProblema:"",
            descripcion:"",
            idTipoProblema:"",
            idUsuario:""
        };
        request(app)
          .post("/reporte")
          .send(reporte)
          .set("Accept", "application/json")
          .expect("Content-Type", /json/)
          .expect(400)
          .expect('"No se creo el reporte"')
          .end((err) => {
            if (err) return done(err);
            done();
          });
      });
 })
```

```
/**
 * Testing endpoint post para un mensaje
 */

 describe("POST /mensaje",() =>{
   it("Responder con un 400 si no se creo el mensaje", (done) => {
        const reporte = {
            descripcion:"",
            idReporte:"",
            idEmpleado:""
        };
        request(app)
          .post("/mensaje")
          .send(reporte)
          .set("Accept", "application/json")
          .expect("Content-Type", /json/)
          .expect(400)
          .expect('"No se creo el mensaje"')
          .end((err) => {
            if (err) return done(err);
            done();
          });
      });
 })

```

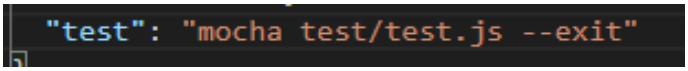
Explicando el código

La función describe() se utiliza para agrupar los test y a la cual le pasamos una descripción del método a testear.

La función it() representa un test al cual le podemos pasar una descripción, declaramos una constante ***mensaje*** a la cual le asignamos la estructura de un json simulando lo que se va a enviar al endpoint, dentro de este método requerimos nuestra app con request(app) y especificamos el tipo de endpoint a testear .post("/mensaje") con su respectiva url, .send(reporte) es lo que le vamos a enviar a nuestro servidor, además de especificar unos encabezados con .set("Accept","aplication/json"), además verificamos las distintas respuestas que esperamos del endpoint con expect("Content-Type",/json/) con esto especificamos que esperamos un json, y .expect(400) especificamos que esperamos una respuesta 400 del endpoint si no se creó el mensaje, además de un .expect("No se creó el mensaje") que es la respuesta requerida que esperamos, esto basado en las respuestas que retorna nuestro endpoint "/reportes" el cual implementamos anteriormente cuando no se pudo crear nuestro mensajes, si esto se cumple nuestro test será exitosos en todo caso el test no pasará. Esto se hace tanto como para el enpoint de reportes como el de mensajes.

Probando nuestro test

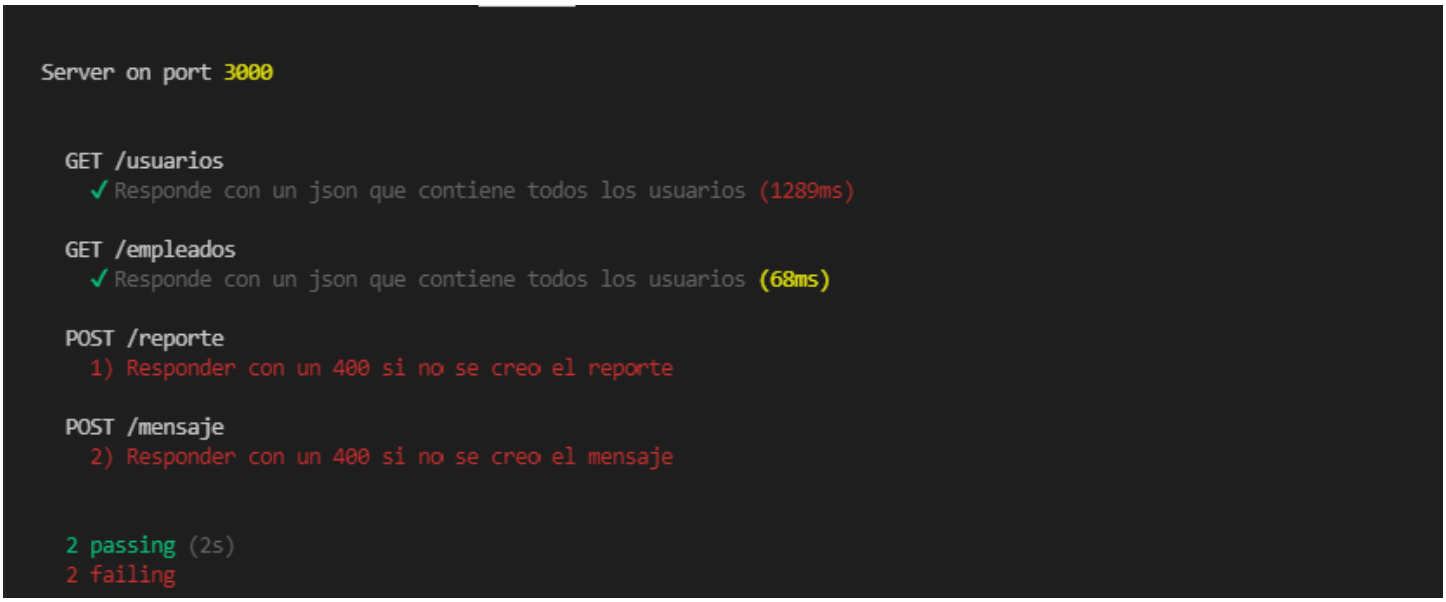
Para facilitar ejecutar nuestro test, crearemos un nuevo script en nuestro archivo ***package.json***, como el que se puede ver en la imagen.



Ejecutamos el script ***npm test*** en nuestra consola.

Resultados:

Como podemos observar dos pruebas pasaron, específicamente las de los enpoints GET usuarios y empleados ya que los métodos retornan lo esperado por el test, mientras dos test no pasaron debido a que nuestros métodos no retornan lo esperado por el test.



Mensajes específicos sobre las pruebas fallidas:

```
  2 failing

  1) POST /reporte
       Responder con un 400 si no se creo el reporte:
     Error: expected 400 "Bad Request", got 200 "OK"
      at Context.<anonymous> (test\test.js:50:12)
      at processImmediate (internal/timers.js:461:21)
  ----
      at Test._assertStatus (node_modules\supertest\lib\test.js:296:12)
      at C:\Users\Sergio-RPR\Desktop\nodejs-restapi-postgres-ts\node_modules\supertest\lib\test.js:80:15
      at Test._assertFunction (node_modules\supertest\lib\test.js:311:11)
      at Test.assert (node_modules\supertest\lib\test.js:201:21)
      at Server.localAssert (node_modules\supertest\lib\test.js:159:12)
      at emitCloseNT (net.js:1659:8)
      at processTicksAndRejections (internal/process/task_queues.js:79:21)

  2) POST /mensaje
       Responder con un 400 si no se creo el mensaje:
     Error: expected 400 "Bad Request", got 200 "OK"
      at Context.<anonymous> (test\test.js:72:12)
      at processImmediate (internal/timers.js:461:21)
  ----
      at Test._assertStatus (node_modules\supertest\lib\test.js:296:12)
      at C:\Users\Sergio-RPR\Desktop\nodejs-restapi-postgres-ts\node_modules\supertest\lib\test.js:80:15
      at Test._assertFunction (node_modules\supertest\lib\test.js:311:11)
      at Test.assert (node_modules\supertest\lib\test.js:201:21)
      at Server.localAssert (node_modules\supertest\lib\test.js:159:12)
      at emitCloseNT (net.js:1659:8)
      at processTicksAndRejections (internal/process/task_queues.js:79:21)


npm ERR! Test failed.  See above for more details.
PS C:\Users\Sergio-RPR\Desktop\nodejs-restapi-postgres-ts>
```

En caso de que arregláramos los métodos que hacen que el test fallen:

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

> mocha test/test.js --exit

Server on port 3000


  GET /usuarios
    ✓ Responde con un json que contiene todos los usuarios (522ms)

  GET /empleados
    ✓ Responde con un json que contiene todos los usuarios (70ms)

  GET /reportes
    ✓ Responde con un json que contiene todos los usuarios (66ms)

  POST /mensaje
    ✓ Responder con un 400 si no se creo el mensaje (104ms)


  4 passing (779ms)

PS C:\Users\Sergio-RPR\Desktop\nodejs-restapi-postgres-ts>
```