Práctica de Back-end

Crear servicio web REST

En esta práctica concluiremos la aplicación que hemos estado desarrollando en las demás prácticas sobre la gestión de usuarios. Esta vez toca desarrollar el back-end, crearemos con node nuestra REST API que permitirá registrar, modificar, editar, consultar y borrar a los usuarios, para facilitar las cosas guardaremos la información en archivos json y realizaremos una autenticación muy básica. Para validar nuestro servicio web lo validaremos únicamente con solicitudes de HTTP para realizar operaciones de altas, bajas, cambios y consultas usando Rest Client o postman. Si al final deseas integrar esta práctica con tu práctica 3 podrás hacerlo.

Descripción de la práctica

Crear una REST API para nuestra aplicación de gestión de usuarios. Lo cual implicará realizar lo siguiente.

1. Autenticación básica de un usuario (regresar token)
2. Registro de nuevos usuarios (con validaciones)
3. Filtro de usuarios (obtener todos o algunos según los parámetros)
4. Consultar usuario en especifico
5. Editar información de un usuario
6. Consultar usuarios
7. Paginado
8. Borrar algún usuario
9. Manejo de errores de respuesta de HTTP

Endpoint	**Método**	**headers**	**body**	**Comentarios**
/api/login	POST	content-type: application/json	correo (del usuario) y password (texto plano)	Esto es de un usuario registrado (primero debes registrar un usuario). Genera un token de usuario. (se mandará por el header x-user-token) Error 401 si no envía los datos correctos 200 en caso de éxito.
/api/users	GET	x-user-auth		Regresar el listado de usuarios. (debe ser usuario autenticado) Hay posibilidad de buscar con ?nombre=valor (buscará los usuarios que contengan ese valor en su nombre), apellido=valor (igual que con nombre) paginado? page=3&limit=4 (por default el límite es 5) filtrar por fecha ?date=valor (valor es la fecha en formato YYYY-MM-DD) Estados HTTP: 401: no autenticado 200: caso exitoso
	POST	content-type: application/json	nombre, apellido, correo, password,	Si se manda la url vacía se autogenera. El sexo debe ser **H** (hombre) o **M** (mujer). cuando se crea el usuario **regresar código 201.**

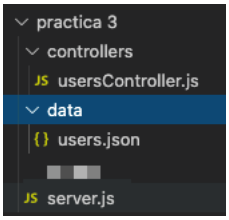
			url(opcional), sexo, fecha	Código 400 si falta algún atributo e indicar cuales faltan. **401 no autenticado.**
/api/users/:email	GET	x-user-auth		Regresar todos los datos del usuario en específico. Regresar también el id Estados HTTP: 401: no autenticado 200: caso exitoso
	PUT	x-user-auth content-type: application/json	Igual que el post en users	No realizará cambios en el correo, ni sexo (para no recalcular la imagen), ni otros identificadores internos. Sí se puede cambiar el nombre, apellido, password, fecha. La url es opcional (si no manda url no se toma en cuenta si la manda sí se toma en cuenta). Estados HTTP: 401: no autenticado 400: parametros incorrectos 200: caso exitoso
	DELETE	x-user-auth		Busca y borra Estados HTTP: 401: no autenticado 200: caso exitoso

Antes de iniciar

Crea una carpeta que se llame practica3 en la consola pon **npm init --yes** para crear el package.json. Por lo pronto solo **instala como dependencia express**, más delante instalaremos lo que se necesite. Crea un directorio con el nombre **data** y dentro crea un archivo **users.json** donde guardaremos la información de los usuarios inicialmente que tenga un arreglo vacío **[ ]**.
Crea un archivo **server.js** e **importa express y fs**.
Carga de forma global el middleware que permite parsear el json (**express.json()**) y **express.urlencoded({ extended: true })**
Crea una ruta raíz que solo regresa el mensaje "*Users app práctica 3*"
Levanta el servicio en el puerto 3000.
¡Prueba que te funciona!

Para la parte de generar las rutas recuerda utilizar en un archivo externo el router para las rutas.
Es importante que tengas en mente que no se espera que se guarde en base de datos. Lo único que se pretende que se haga es que modifiques el arreglo.

Ahora crea un directorio con el nombre de **controllers** dentro de la carpeta practica3 crea un archivo que se llame usersController.js
Deberías tener una estructura de archivos de esta forma:



Dentro del archivo de **usersController.js** Copia y pega el siguiente código.

```
const fs = require('fs');
const USERS_DB = require('../data/users.json');
let CURRENT_ID = 0;
```

```
let uids = USERS_DB.map((obj)=>{return obj.uid});
CURRENT_ID = Math.max(...uids)+1;
console.log(`Current id: ${CURRENT_ID}`);
// console.table(USERS_DB);

class UsersController {


    generateId(){
        let id = CURRENT_ID;
        CURRENT_ID++;
        fs
        return id;
    }
    insertUser(user){
        user.uid = this.generateId();
        USERS_DB.push(user);
        return user;
    }
    updateUser(user){
        let index = USERS_DB.findIndex(element => element.uid === user.uid);
        if(index>-1){
            USERS_DB[index] = Object.assign(USERS_DB[index],user);
            return user;
        }else{
            return undefined;
        }
    }

    deleteUser(user){
        let index = USERS_DB.findIndex(element => element.uid === user.uid);
        if(index>-1){
            USERS_DB.splice(index,1);
            return true;
        }else{
            return false;
        }
    }

    getList(){
        return USERS_DB;
    }
    getUserByCredentials(email, password){
        let users = USERS_DB.filter((item,index,arr)=>{
            if( item.password.toLowerCase()=== password.toLowerCase() &&
                item.email.toLowerCase() === email.toLowerCase()){
                return true;
            }
            return false;
        });
        return users[0];
```

```
    }
    getUniqueUser(name,lastname,email){
        let users = USERS_DB.filter((item,index,arr)=>{
            if(item.nombre.toLowerCase()=== name.toLowerCase() &&
                item.apellidos.toLowerCase()=== lastname.toLowerCase() &&
                item.email.toLowerCase() === email.toLowerCase()){
                return true;
            }
            return false;
        });
        return users[0];
    }
    getUser(id){
        let user = USERS_DB.find(ele=>ele.uid ===id);
        return user;
    }
    getUserByEmail(email){
        let user = USERS_DB.find(ele=>ele.email ===email);
        return user;
    }
}

module.exports = UsersController;
```

Como observaras carga un archive que se llama users.json. Deberás crear este archivo users.json dentro de la carpeta data. Te recomiendo que crees un arreglo de usuarios con la estructura que hemos estado usando. Por favor usar el mismo objeto del enpoint de la practica2.

POST /api/users

Lo primero que realizaremos será trabajar con el registro de un usuario para ello debemos hacer varias cosas:

1. Validar que en el body se encuentren los atributos que nos interesan en caso contrario regresar **bad request** e indicar en el mensaje de respuesta cuales atributos faltan. Recuerda que puedes utilizar los métodos del controlador.
2. Validar que no exista otro usuario con el mismo correo o con el mismo nombre y apellidos. Regresar **bad request** en caso de existir.
3. Si no hay usuario genera un usuario con base en la información enviada y con un nuevo id auto-incremental, que debe guardarse en alguna variable global y que debe asignarse tan pronto se lea el archivo de usuarios o guardarlo en otro archivo json y leerlo al inicio. Te recomiendo implementar esta lógica dentro de **UsersController.js. Recuerda que puedes calcular el último ID basándote en le máximo del arreglo de usuarios.**

4. Guardar el nuevo usuario en el arreglo y regresar el status 201 y como mensaje el JSON del usuario creado (todos los datos del usuario creado)

POST /api/login

Para simular la creación del token instala una librería que te permite generar cadenas aleatorias **randomatic** https://www.npmjs.com/package/randomatic **instálalo y no olvides importarlo (**`const randomize = require('randomatic');`**).** Crea el endpoint **/api/login** (lo puedes dejar dentro de server.js) con el método **POST** que debe recibir un **correo y contraseña** como body.

1. Si falta alguno de los atributos regresar el error correspondiente con el mensaje de lo que falta. (400)
2. Obten el valor del correo y del password y busca que alguno de los usuarios lo tenga de lo contrario regresa un error (401) con un mensaje.
3. En caso de que todo este correcto
    a. Revisa si el usuario tiene un token, si no genera con randomatic un token guardando el resultado de **randomize('Aa0','10')+"-"+userId**, añadelo al atributo token del usuario y guárdalo en el arreglo de usuarios regresa el token como respuesta. La respuesta puede ser un JSON como {token: valorToken}

El token generado será requerido para poder acceder a las siguientes operaciones de nuestro servicio web

Middleware de autenticación

1. Crear un middleware llamado **athentication** que verifica que exista el header x-auth-user con el token. Mandar error 401 en caso de que no existan "usuario no autenticado".
2. Revisa que el token exista para el id indicado al final del token (puedes usar split("-") y pop() para quedarte con el último valor)
    a. Si no manda otro error.
    b. En caso de existir guarda el id del usuario en el request como req.uid = idDelUsuario y **ejecuta next()**

GET /api/users

Esta es la parte más extensa de la práctica ya que hay muchas cosas que se podrán hacer con el listado de usuarios como filtros, paginado y búsquedas. Antes de continuar no olvides **poner el middleware de autenticar**.
Iremos resolviendo por partes poco a poco.

1. Regresar todos los usuarios: Solamente aclarar que no regresamos todos los atributos solo algunos esto lo puedes hacer con un map para quitar atributos como password, sexo, fecha, token.
2. Búsquedas para ello tenemos la opción por nombre o apellido a través del query param ?name=mar si tiene este parámetro entonces buscar aquellos usuarios que en su nombre o lastname incluya esta palabra usar **cadena.toUpperCase()** y también **cadena.includes()**
3. Paginado: si existe el query parameter page=numeroPagina, limit =elementosPorPágina, crea una formulita que te diga el id correspondiente al primer elemento que debe ir en esa página y obtén los siguientes. Si no está el limit tomar el default (5)
4. Filtrar por año solamente es quedarse con aquellos usuarios que cumplan con la condición.

GET /api/users/:email

Crea un route para el endpoint de detalle; regresa todos los atributos del usuario en específico. **No olvides poner el middleware de autenticar**

PUT /api/users/:email

Revisa que existan los atributos obligatorios de lo contrario marca error (parecido a post de usuarios). Alguien si el usuarios no existe (404). Si todo está en orden añade las modificaciones. No olvides actualizar el archivo de usuarios. **No olvides poner el middleware de autenticar**

DELETE /api/users/:email

Busca el usuario indicado si no existe manda un 404, en caso de existir borrarlo y actualizar el archivo. **No olvides poner el middleware de autenticar**

Integrar con front-end (opcional)

Esta parte opcional implica unir la práctica 2 y 1 con esta práctica. Para ello se crea una carpeta “public” dentro de la carpeta de practica4. Debes utilizar el middleware para cargar los archivos estáticos y editar las peticiones

Persistir el arreglo de usuarios y de Ids (Opcional)

Esta parte opcional implica que utilices los métodos de fs para guardar en sistema de archivos para que cuando se reinicie la aplicación el arreglo se recree.

Evaluación

En la entrega debes mandar todos tus archivos (excepto la caperta de node_modules, si la incluyes restaré 20 puntos) y documento PDF en que incluyas tus conclusiones de la práctica y una tabla donde indicas cuales requerimientos has cumplido. Sube un archivo .zip con el nombre de archivo: apellidos_nombre_expediente.zip

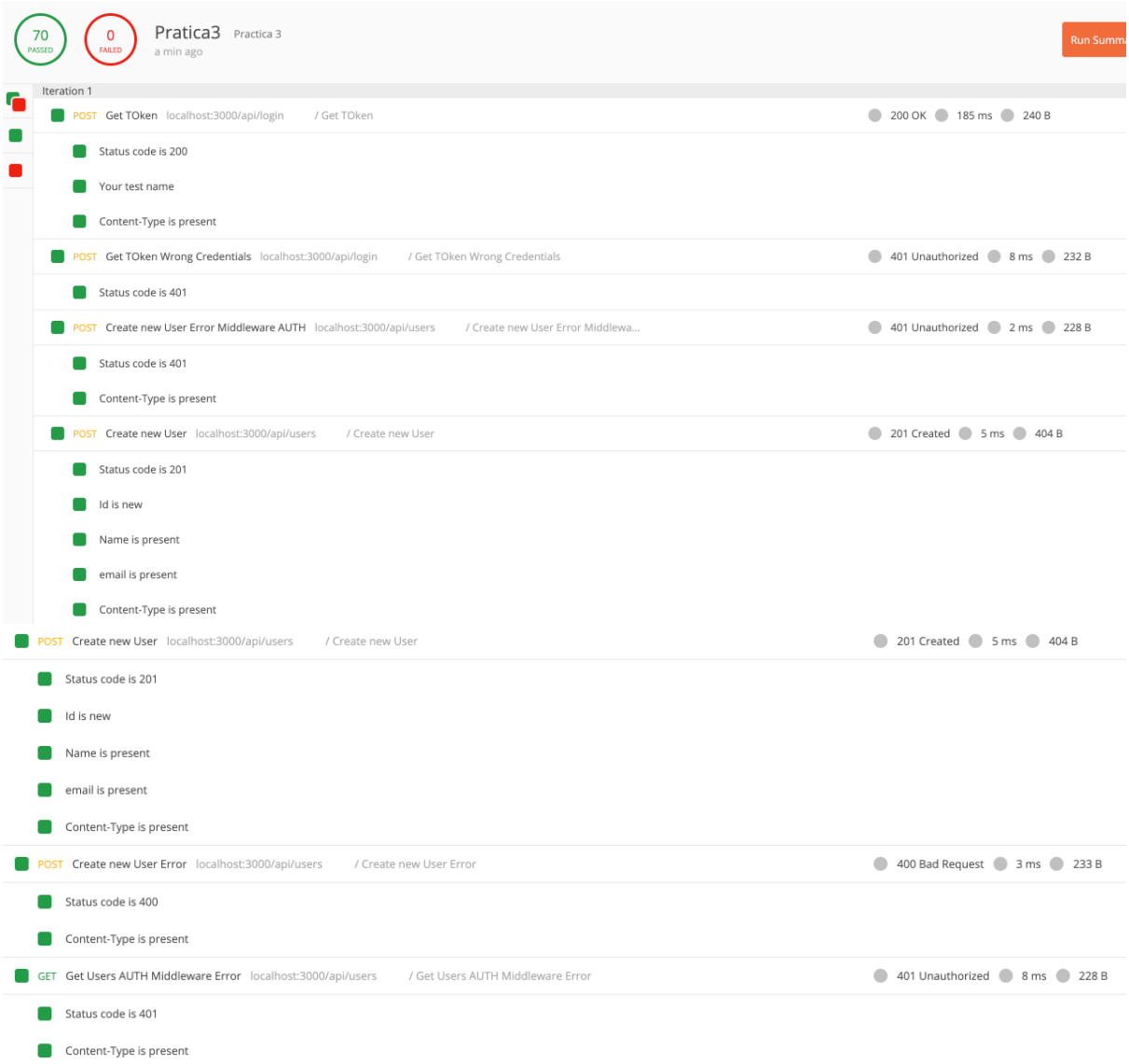
NOTA: si utilizas otros nombres de path, inclusive de propiedades el script marcará error por lo que lo tomaré como si no hubieras hecho ese ejercicio.

Requerimiento (incluye validaciones, regresar código de error y que funcione correctamente).	Valor	Realizado (Si/No)
1. POST /api/users	15	
2. POST /api/login	15	
3. Middleware de autenticar	10	
4. GET /api/users -Obtener todos los usuarios -Búsqueda de nombre o apellido	15	
5. GET /api/users - Filtrar por año - paginado	15	
6. GET /api/users/:email	10	
7. PUT /api/users/:email	10	
8. DELETE /api/users/:email	10	
9. Integración con front-end (opcional)	10	
10. Persistencia de usaurios (Opcional)	10	

Si obtienes una calificación mayor a 100 te quedas con 100.

Anexo

Output de pruebas POSTMAN:



GET GetUsers normal localhost:3000/api/users / GetUsers normal 200 OK 6 ms 610 B

- Status code is 200
- Size of data is correct
- nombre is defined
- appellidos is defined
- email is defined
- Only 4 props in object
- Content-Type is present

GET GetUsers Filter names localhost:3000/api/users... / GetUsers Filter names 200 OK 10 ms 289 B

- Status code is 200
- Size of data is correct
- nombre is defined
- appellidos is defined
- email is defined
- Only 4 props in object
- Content-Type is present

GET GetUsers Filter names and page limit localhost:3000/api/users... / GetUsers Filter names and page ... 200 OK 3 ms 2.091 KB

- Status code is 200
- Start id is correct
- End id is correct
- Size of data is correct
- nombre is defined
- appellidos is defined
- email is defined
- Only 4 props in object
- Content-Type is present

GET GetUsers Filter names, page limit,dates localhost:3000/api/users... / GetUsers Filter names, page limi... 200 OK 3 ms 294 B

- Status code is 200
- User id is correct
- Size of data is correct
- nombre is defined
- appellidos is defined
- email is defined
- Only 4 props in object
- Content-Type is present

GET Get user by Email localhost:3000/api/users... / Get user by Email — 200 OK · 3 ms · 404 B

- Status code is 200
- id is correct
- nombre is defined
- appellidos is defined
- email is defined
- Only 8 props in object
- Content-Type is present

GET Get user by Email AUTH localhost:3000/api/users... / Get user by Email AUTH — 401 Unauthorized · 3 ms · 228 B

- Status code is 401
- Content-Type is present

PUT Update User localhost:3000/api/users... / Update User — 200 OK · 2 ms · 417 B

- Status code is 200
- Id is new
- Name is present
- email is present
- Content-Type is present

PUT Update User Missin Args localhost:3000/api/users... / Update User Missin Args — 400 Bad Request · 2 ms · 231 B

- Status code is 400
- Content-Type is present

PUT Update User AUTH Error localhost:3000/api/users... / Update User AUTH Error — 401 Unauthorized · 2 ms · 228 B

- Status code is 401
- Content-Type is present

DELETE Delete User AUTH Error localhost:3000/api/users... / Delete User AUTH Error — 401 Unauthorized · 2 ms · 228 B

- Status code is 401
- Content-Type is present

DELETE Delete User localhost:3000/api/users... / Delete User — 200 OK · 2 ms · 228 B

- Status code is 200
- Content-Type is present

GET Get user by Email After delete localhost:3000/api/users... / Get user by Email After delete — 204 No Content · 2 ms · 152 B

- Status code is 204
- Body matches string