SISTEMAS WEB

SKILLS

PARTE 2: BACK END

GRUPO 10: MIKEL FAJARDO, EUKEN SÁEZ, UNAI LEÓN

Repositorio de GitHub
https://github.com/unai002/skillsG10

30/12/2024

1. INTRODUCCIÓN

Durante este informe se va a analizar la segunda parte del proyecto realizado, correspondiente a la sección de back-end. Partiendo de lo realizado en la primera parte del proyecto se va a realizar la parte correspondiente al control de usuarios, evidencias y privilegios de administrador.

En primer lugar el proyecto utilizará la herramienta MongoDB, la cual es una base de datos no relacional diseñada para almacenar grandes volúmenes de datos. Los datos se almacenan en documentos similares a los JSON, ideales para trabajar con JavaScript y páginas web, que no siguen una estructura fija permitiendo una gran flexibilidad a la hora de almacenar datos. En segundo lugar la división de trabajo está basada en rutas y controladores que serán los que nos permitan gestionar las peticiones POST y GET, además de permitirnos movernos por los archivos ejs.

2. MONGODB

IMPORTANTE: El proyecto está configurado para trabajar con una base de datos MongoDB local en el puerto 27017 y con el nombre **skillsDB**. Se puede cambiar la configuración en el archivo app.js.

Para desarrollar la segunda parte, primero hemos definido los esquemas a utilizar para guardar la información necesaria en la base de datos MongoDB. Los esquemas se encuentran en el directorio models del proyecto.

- **badge.model.js**. Para guardar las medallas (Badge).
- **skill.model.js**. Para guardar las competencias (Skill).
- **user.model.js**. Para guardar la información de los usuarios (User).
- **userskill.model.js**. Para guardar las relaciones entre los usuarios y las competencias (UserSkill). Hemos hecho ligeros cambios respecto al enunciado: principalmente, para decidir si las tareas de una competencia están completadas utilizamos un array que indica qué tareas están completas, en lugar de un único valor booleano.

Una vez tenemos los modelos, podemos utilizarlos en la aplicación mediante funciones de la librería Mongoose para gestionar la información de la base de datos. Además, hemos añadido un script **initializeDB** que inicializa la base de datos creando las instancias de competencias (Skills) y medallas (Badges) que hay en el sistema. También añade dos usuarios: user (usuario normal) y admin (administrador), ambos con contraseña “1234” (para realizar pruebas rápidamente).

3. PLANTILLAS

Las páginas HTML definidas en la primera parte pasan a ser todas plantillas EJS. Para las que ya teníamos definidas, el funcionamiento principal sigue realizándose mediante JavaScript, sin utilizar realmente las ventajas que ofrecen las plantillas. Para las nuevas páginas, sí que se utilizan recursos como los parámetros para renderizar la información.

Las plantillas son redireccionadas desde las rutas, es decir, cuando se realiza un peticion del tipo POST o GET, se hace llamada a las rutas que a través del método render redireccionará la ruta actual a la deseada dependiendo de la llamada realizada desde el cliente. En resumen, esa llamada a las rutas genera otra llamada al correspondiente controlador encargado de, a parte de tramitar la petición, renderizar una nueva vista haciendo llamada a la ruta relativa del archivo EJS.

A continuación se define la función de cada vista:

- **login.ejs**: encargada de iniciar sesión mediante un nombre de usuario y contraseña. Esta es la página de inicio principal.
- **register.ejs**: encargada de la creación de nuevos usuarios por medio de un nombre y contraseña.
- **main.ejs**: es la página principal donde aparecen todas las skills a completar por los usuarios o editar y crear por parte de los administradores. Los hexágonos que representan las skills se han alterado para diferenciar entre usuarios y administradores. En la parte de usuario solo se ha habilitado la sección de formulario para poder completar las skills además de mantener las notificaciones ya implementadas. Por parte del administrador, aparece un dashboard en la parte superior que va a permitir al administrador tanto modificar como borrar medallas, como crear una nueva skill que será añadida y editar las skills existentes.
- **admindashboard.ejs**: esta vista será proporcionada únicamente al usuario tipo administrador. Desde esta tendremos la redirección a las vistas relativas con la modificación y borrado de medallas, la creación de nuevas skills y la lista de usuarios con la opción de cambiar la contraseña de dichos usuarios.
- **editbadge.ejs**: vista encargada de la modificación de medallas.
- **editskill.ejs**: vista encargada de la modificación de competencias.
- **adminbadges.ejs**: vista encargada de desplegar las medallas y las acciones posibles para el usuario tipo administrador.
- **createskill.ejs**: vista encargada de la creación de nuevas skills.
- **usercontroller.ejs**: vista encargada del listado y de la posibilidad de modificación de las contraseñas de usuarios.
- **skillspecifics.ejs**: la página donde se muestra dinámicamente la información de un solo skill, las tareas a realizar y las evidencias enviadas.

- **leaderboard.ejs**: la página con la información de las medallas y el dónde se encuentra cada usuario registrado según su puntuación.
- **username-display.ejs** (partial): la cabecera (menú) que se añade a todas las páginas. Permite navegar fácilmente entre las páginas principales.

4. AUTENTIFICACIÓN

El proyecto también consta de la gestión de usuarios, los cuales pueden ser tanto administradores, que no podrán ser registrados y están añadidos de manera local, y los usuarios que deberán realizar un formulario de registro para poder acceder al resto de funcionalidades.

- **Administrador**: este usuario es capaz de modificar y crear skills, además de modificar contraseñas de otros usuarios, modificar y eliminar medallas y aceptar evidencias directamente (sin necesidad de ser aceptada tres veces).

- **Usuario**: este usuario tendrá acceso a una serie de skills que deberá superar. Para ello, por cada skill existe un formulario a completar a la espera de ser aceptado tanto por un administrador como por 3 usuarios.

Desde la plantilla perteneciente al login hacemos llamada al login.js que genera una llamada POST que será tratada desde el controlador de users:

```
try {
    const response :Response = await fetch( input: 'http://localhost:3000/users/login', init: {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify( value: { username, password }),
    });

    const data = await response.json(); // Parsear la respuesta JSON
```

Tenemos dos puntos a analizar: el login y el register.

El register va a crear un nuevo usuario creando un objeto similar a un JSON que añade a la base de datos un nuevo usuario recogiendo los elementos introducidos utilizando la librería bcrypt para cifrar las contraseñas de manera segura.

```
exports.register = async (req, res) : Promise<...> => {
    const { username, password } = req.body;

    if (!username || !password) {
        return res.status(400).json({
            status: 'error',
            message: 'Se requieren username y password',
        });
    }

    try {
        // Check if the user already exists
        const existingUser : Query<...> & ObtainSchemaGeneric<module:mon... = await User.findOne( filter: { username });
        if (existingUser) {
            return res.status(400).json({
                status: 'error',
                message: 'El usuario ya está registrado',
            });
        }

        // Hash the password
        const hashedPassword = await bcrypt.hash(password, salt: 10);
        console.log("register ", hashedPassword)

        // Determine if the user is the first user (admin)
        const isFirstUser : boolean = (await User.countDocuments( filter: {})) === 0;

        // Create a new user
        const newUser : HydratedDocument<InferSchemaType<...> = new User( doc: {
            username,
            password: hashedPassword,
            admin: isFirstUser
        });

        // Save the user to the database
        await newUser.save();

        return res.status(201).json({
            status: 'success',
            message: 'Usuario registrado exitosamente',
        });
    } catch (error) {
        return res.status(500).json({
            status: 'error',
            message: 'Error al registrar el usuario',
        });
    }
};
```

El login por otra parte accede a la base de datos en busca del usuario introducido comparando nombre y usuario. Utilizando de nuevo bcrypt para la codificación de contraseñas.

```
exports.login = async (req, res) : Promise<...> => {
    const { username, password } = req.body;

    if (!username || !password) {
        return res.status(400).json({
            status: 'error',
            message: 'Se requieren el username y el password',
        });
    }

    try {
        // Find the user in the database
        const user : Query<...> & ObtainSchemaGeneric<module:mon... = await User.findOne( filter: { username });
        if (!user) {
            return res.status(401).json({
                status: 'error',
                message: 'El usuario no existe.',
            });
        }

        // Compare the provided password with the hashed password in the database
        const isMatch = await bcrypt.compare(password, user.password);
        console.log("login ", user.password)

        if (!isMatch) {
            return res.status(401).json({
                status: 'error',
                message: 'La contraseña es incorrecta.',
            });
        }

        req.session.username = username;
        req.session.admin = user.admin;
```

5. ADMINISTRADOR

Como se ha mencionado anteriormente el usuario del tipo administrador va a ser capaz de modificar los elementos de la propia página. Para ello con el panel de control de la parte superior de la pantalla se accede a una serie de opciones únicas de este tipo de usuarios donde incluye el dashboard.

Uno de esos elementos mencionados son los skills. El administrador será capaz de añadir una nueva skill visible para todos los demás usuarios añadiendo a su vez a la base de datos. Para ello dentro de las opciones del dashboard está la de la creación de competencias. En este nos redirige a la plantilla correspondiente que nos permitirá completar un formulario que añadirá la nueva skill a la base de datos.

Para ello hay que saber un primer lugar el id que va a tener la nueva skill, para ello buscamos el último elemento añadido y le asignamos el número posterior para a continuación, añadir tanto el nombre como la descripción y las tareas a realizar. Añadir un icono sería opcional. Para añadir el icono utilizamos la librería Multer, y configuramos el directorio donde están todos los iconos como la localización de almacenamiento.

```
        try {
            // Fetch the last skill to determine the next id
            const lastSkill = await Skill.findOne().sort( arg: {id: -1});
            const nextId : any | number  = lastSkill ? lastSkill.id + 1 : 1;

            // Create a new skill document
            const newSkill : HydratedDocument<InferSchemaType<...>  = new Skill( doc: {
                id: nextId,
                text: text,
                description: description,
                tasks: tasks ? tasks.split('\n') : [],
                resources: resources ? resources.split('\n') : [],
                score: parseInt(score,  radix: 10),
                icon: icon,
                set: skillTreeName
            });

            // Save the new skill to the database
            await newSkill.save();

            console.log('New skill added:', newSkill);
            res.redirect(`/skills/${skillTreeName}`);
        } catch (error) {
            console.error('Error adding new skill:', error);
            res.status(500).send('Internal server error');
        }
    });
};
```

Desde el propio dashboard existe también la opción de administrar los usuarios existentes y dentro de este la opción de cambiar la contraseña de esos usuarios. Desde esta opción debemos acceder a una lista completa de los usuarios alojados en la base de datos y desplegarlos en una vista donde junto a ello se habilita la opción de cambio de contraseña que será la que hará la llamada al controlador.

Primero necesitamos acceder al render que nos proporcionará la lista de usuarios obtenidos de la base de datos.

```
exports.getUserList = async (req, res) : Promise<void>  => {
    const username = req.session.username;
    const isAdmin = req.session.admin;
    try {
        const users : Query<...> & ObtainSchemaGeneric<module:mon...  = await User.find();
        console.log('Fetched users:', users);
        res.render('usercontroller', { users, username, admin: isAdmin });
    } catch (error) {
        console.error('Error fetching users from database:', error);
        res.status(500).render('error', { message: 'Internal Server Error', error });
    }
};
```

Y posteriormente se podrá hacer la llamada para realizar el cambio de contraseña. Desde donde se obtendrá el nombre del usuario al cual se va a realizar la acción para buscar dicho usuario en la base de datos para realizar el cambio de contraseña, cifrada de nuevo con bcrypt.

```
exports.changePassword = async (req, res) : Promise<...> => {
    const { userId, newPassword } = req.body;

    // Validate that both userId and newPassword are provided
    if (!userId || !newPassword) {
        return res.status(400).json({ error: 'Both userId and newPassword are required.' });
    }

    try {
        // Find the user by their ID
        const user : Query<...> & ObtainSchemaGeneric<module:mon... = await User.findById(userId);
        if (!user) {
            return res.status(404).json({ error: 'User not found.' });
        }

        // Hash the new password
        const hashedPassword = await bcrypt.hash(newPassword, salt: 10);

        // Update the user's password
        user.password = hashedPassword;
        await user.save();

        // Send a JSON response with the result of the change
        res.redirect('/admin/dashboard');
    } catch (error) {
        console.error('Error changing password:', error);
        res.status(500).json({ error: 'Internal Server Error' });
    }
};
```

Como última opción de administrador tenemos la edición y borrado de medallas. Para este caso se despliega una lista de medallas existentes accedidas mediante la base de datos, donde por cada una de las medallas tenemos dos opciones: borrar la medalla o editar la misma.

Al hacer la llamada se accedera a las medallas haciendo búsqueda en la base de datos de las medallas con en nombre Badge que serán desplegadas por medio de la vista.

```
exports.getAdminDashboard = (req, res) : void => {
    const username = req.session.username;
    const isAdmin = req.session.admin;
    res.render('admindashboard', { username, isAdmin });
};

exports.getAdminBadges = async (req, res) : Promise<void> => {
    try {
        const badges : Query<...> & ObtainSchemaGeneric<module:mon... = await Badge.find().sort( arg: { bitpoints_min: 1 });
        const username = req.session.username;
        const isAdmin = req.session.admin;

        if (req.headers.accept && req.headers.accept.includes( value: 'application/json')) {
            res.json(badges);
        } else {
            res.render('adminbadges', { username, admin: isAdmin, badges });
        }
    } catch (err) {
        console.error('Error fetching badges from database:', err);
        res.status(500).render('error', { message: 'Internal Server Error', error: err });
    }
};
```

En primer lugar está el borrado de medallas. Este hará un llamada al controlador el cual será el encargado de borrar la medalla de la base de datos. La medalla será buscada en la base de datos por su id y eliminada.

```
exports.deleteBadge = async (req, res) : Promise<...> => {
    try {
        const { id } = req.params;
        const deletedBadge : Query<...> & ObtainSchemaGeneric<module:mon... = await Badge.findOneAndDelete( filter: { name: id });

        if (!deletedBadge) {
            const notFoundError : Error = new Error('Badge not found');
            notFoundError.status = 404;
            return res.status(404).render('error', { message: 'Badge not found', error: notFoundError });
        }

        res.redirect('/admin/badges');
    } catch (err) {
        console.error('Error deleting badge:', err);
        res.status(500).render('error', { message: 'Error deleting badge', error: err });
    }
};
```

Aparte de la posibilidad de borrar las medallas existe la posibilidad de modificar dichas medallas. Existe la vista correspondiente que será desplegada con la llamada al controlador que una vez obtenida la medalla a modificar se indicará mediante un formulario los elementos que se quiera modificar. Una vez completado (o no) dicho formulario se hará la correspondiente llamada que indicará el cambio de medalla y así se registrará en la base de datos.

En resumen en primer lugar se obtiene la medalla a modificar y se establece la vista.

```
exports.getEditBadge = async (req, res) : Promise<...> => {
    try {
        const badgeName = req.params.id;
        const badge : Query<...> & ObtainSchemaGeneric<module:mon... = await Badge.findOne( filter: { name: badgeName });

        if (!badge) {
            const notFoundError : Error = new Error('Badge not found');
            notFoundError.status = 404;
            return res.status(404).render('error', { message: 'Badge not found', error: notFoundError });
        }

        const username = req.session.username;
        const isAdmin = req.session.admin;
        res.render('editbadge', { badge, username, admin: isAdmin });
    } catch (err) {
        console.error('Error fetching badge from database:', err);
        res.status(500).render('error', { message: 'Internal Server Error', error: err });
    }
};
```

Y luego se aplican las modificaciones realizadas en el correspondiente formulario.

```
exports.updateBadge = async (req, res) : Promise<...> => {
    try {
        const { id } = req.params;
        const { badgeName, minBitpoints, maxBitpoints, imageUrl } = req.body;

        const updatedBadge : Query<...> = await Badge.findOneAndUpdate(
            filter: { name: id },
            update: {
                name: badgeName,
                bitpoints_min: minBitpoints,
                bitpoints_max: maxBitpoints,
                image_url: imageUrl
            },
            options: { new: true }
        );

        if (!updatedBadge) {
            return res.status(404).send('Badge not found');
        }

        res.redirect('/admin/badges');
    } catch (err) {
        console.error('Error updating badge:', err);
        res.status(500).send('Error updating badge');
    }
};
```

6. EVIDENCIAS Y COMPROBACIÓN DE COMPETENCIAS

La lógica principal para el envío y comprobación de las evidencias ya estaba implementada en la parte anterior, pero utilizando objetos JSON guardados en localstorage. Para la segunda parte, hemos adaptado dicha lógica para obtener los objetos de la base de datos en lugar del localstorage; y la gestión de los datos se realiza desde los archivos controller en vez de en los archivos JavaScript.

Los cambios realizados en las funciones son los siguientes.

skillspecifics.js

- **loadSkillInformation**. Utilizamos la ruta skills/electronics/info para obtener todas las skills de la base de datos y poder renderizar la información de la skill que buscamos.
- **loadUserTasks**. Se utiliza para cargar las tareas que el usuario ya ha marcado como realizadas y marcarlas en las casillas de la página, garantizando que el progreso se guarda. Utilizamos la ruta /skills/electronics/userTasks con los parámetros del usuario y la skill para encontrar las tareas.
- **checkboxVerify**. Comprueba el estado de los checkboxes y lo guarda en la base de datos mediante skills/electronics/updateTasks.
- **loadEvidence**. Carga la evidencia, si el usuario ya ha enviado una, y la escribe en la caja de texto. Utiliza la ruta skills/electronics/getEvidence y los parámetros necesarios.

- **handleEvidenceSubmission**. Utilizamos la ruta skills/electronics/submit-evidence para guardar la evidencia en la base de datos.
- **loadAndDisplayEvidences**. Utilizamos la ruta skills/electronics/getAllEvidences para obtener todas las evidencias para la skill actual y cargarlas en una tabla donde pueden aprobarse.
- **handleApprove**. Utilizamos la ruta skills/electronics/skillID/verify para verificar la evidencia en la base de datos.
- **handleReject**. Utilizamos la misma ruta que en handleApprove para rechazar la evidencia, pasando el atributo approved como falso.

main.js

- En el fetch principal, obtenemos la información de las skills desde la ruta skills/info, que devuelve todas las skills guardadas en el sistema. Así, utilizamos la lógica ya implementada de la primera parte para crear los hexágonos.
- **loadNotificationDots**. Por cada skill, tenemos que cargar en el punto rojo las evidencias que se han enviado y en el punto verde las aprobaciones que ha obtenido la evidencia enviada por el usuario actual. Para ello, utilizamos una nueva ruta skills/electronics/getAllEvidences, que devuelve una lista de objetos con la misma estructura que la implementación anterior requería. Así, podemos reutilizarla.

7. LEADERBOARD

Sobre la información de las medallas que teníamos anteriormente, hemos creado el ranking de usuarios y las medallas que tienen según los puntos que han obtenido. Siguiendo con la implementación inicial de la primera parte, hemos completado el archivo leaderboard.js para añadir, además de la tabla informativa, el leaderboard de los usuarios.

Para ello, utilizamos las nuevas rutas users/badges y users/allUsers. De esta forma, creamos una tabla por cada medalla, y dentro añadimos la información de los usuarios cuya puntuación se encuentre en el rango de puntos de la medalla.

8. CONCLUSIÓN

Como punto final hay que analizar las conclusiones obtenidas después de la realización del proyecto en conjunto. Como primer punto hay que recuperar las ideas obtenidas durante la primera parte del trabajo. Durante esa parte se llevó a cabo la parte relacionada con el front-end, es decir, la parte más visible de la página web. Para esta parte pudo recalcar en la importancia de usar en conjunto JavaScript con CSS y HTML. Cada una de las partes forman parte de la página y se ha usado como plantilla para todo el proyecto. Queda constancia de la importancia del uso de los fetch y entender cómo manejar la información obtenida mediante estos, es decir, el tratamiento de las promesas y los JSON.

En la parte dos se observó un enfoque algo diferente. Usando lo aprendido durante el primer proceso del proyecto, este se ha centrado en la parte Back-End, es decir en la parte menos visible, que hace que todo tenga un sentido mucho más claro. El uso constante de las rutas y los controladores hace claro de la importancia de las peticiones GET y POST (principalmente) en el proceso de creación web. Entender cómo renderizar las vistas mediante el uso de las mencionadas rutas y enlazar estos procesos con los scripts ha sido clave. Por último, el uso de una base de datos no relacional muy aplicable para estos casos gracias al funcionamiento de MongoDB (usa documentos similares a los JSON), se ha podido hacer un uso mucho más amplio del desarrollo web realizado.

En conjunto se ha realizado un desarrollo de una página web estable, funcional y con diversas funcionalidades que sirve como base y precedente para futuros proyectos.