SISTEMAS WEB

SKILLS

PARTE 1: FRONT END

GRUPO 10: MIKEL FAJARDO, EUKEN SÁEZ, UNAI LEÓN

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

17/11/2024

1. INTRODUCCIÓN

El objetivo del proyecto es implementar una plataforma de aprendizaje gamificado que fomente el aprendizaje colaborativo sobre electrónica. El aprendizaje se realiza completando tareas para obtener capacidades (skills) referentes al tema. Por cada skill, el usuario gana puntos con el que consigue medallas. El objetivo final de la plataforma es conseguir la última medalla. También contendrá otras funcionalidades como la visualización del progreso y paneles de control.

En esta primera parte nos centraremos en implementar la parte del front end, con la que el usuario interactuará. En la segunda parte la completaremos con las funcionalidades del back end.

Ejercicios

Antes de comenzar, hemos preparado en entorno del proyecto instalando las herramientas que utilizaremos durante el desarrollo: NodeJS y Express. Después creamos el proyecto con la estructura básica de un proyecto Express, y realizamos toda la vinculación al repositorio de GitHub para trabajar manteniendo todas las versiones unificadas. Todo el desarrollo del proyecto se está realizando en el IDE IntelliJ.

Obtener la información de las skills

El primer paso es obtener la información (id, texto e icono) de cada skill. Utilizaremos la información de la página web https://tinkererway.dev/web_skill_trees/electronics_skill_tree. Para ello, implementamos un script que lo realice automáticamente (scraper.js).

Utilizaremos la biblioteca Puppeteer para hacer el web scraping (extraer información de la página web). En el script definimos una función extraer(). Configuramos el objeto Puppeteer y le indicamos la página a la que debe navegar. Después la abrimos e implementamos la función evaluate().

```
const browser :Browser  = await puppeteer.launch();
const page :Page  = await browser.newPage();

await page.goto(url,  options: {waitUntil: 'networkidle0'});
```

Para llegar a los objetos skill donde está la información, primero obtenemos el container mediante un querySelector; y después utilizamos otro querySelector para obtener los elementos dentro del container. Creamos un array skills en el que devolveremos la información resultante.

```
const container :Element  = document.querySelector( selectors '.svg-container')
const skillElements :NodeListOf<Element>  = container.querySelectorAll( selectors '.svg-wrapper');
const skills :any[]  = [];
```

Para cada elemento skill, recorremos sus hijos para obtener los datos. El id se extrae del atributo de cada elemento <svg>, y el texto se obtiene concatenando los textos de los elementos <tspan>. Separamos cada línea con "\n" para identificar los saltos de línea. Para el nombre del icono, separamos los segmentos de la ruta a la imagen, y obtenemos el último (sabemos que será el nombre de la imagen, el nombre del icono).

```
skillElements.forEach( callbackfn: skill : Element => {
    const id : string = skill.getAttribute( qualifiedName: 'data-id');
    console.log(id);
    const tspans : NodeListOf<SVGTSpanElement> = skill.querySelectorAll( selectors: 'tspan')
    const text : string = Array.from(tspans).map(tspan : SVGTSpanElement => tspan.innerHTML.trim()).join('\n')
    const iconPath : string = skill.querySelector( selectors: 'image').getAttribute( qualifiedName: 'href');
    const iconName : string = iconPath.split( separator: '/').pop();

    skills.push({
        id: parseInt(id),
        text: text,
        icon: iconName
    });
});
```

Descarga de las imágenes

Creamos otro script para descargar los iconos correspondientes a cada skill (download_icons.js).

Primero definimos una función, getIconUrls(), para obtener las rutas a las imágenes, siguiendo un proceso similar al script anterior con Puppeteer.

Después definimos otra función, downloadIcon() para descargar un icono concreto. Le pasamos la página de Puppeteer donde buscar las imágenes, el enlace a la imagen a descargar y el índice. Convertimos la imagen a un objeto Buffer, para poder tratarla y guardarla en un archivo en la ruta indicada.

```
async function downloadIcon(page, url, index) : Promise<void> { Show usages  Unai Leon Plaza *
    try {
        const response = await page.goto(url);
        if (!response.ok) throw new Error(`Error ${response.status}: ${response.statusText}`);

        // Convertimos la respuesta a un objeto Buffer porque es una imagen (información binaria)
        const iconData = await response.buffer();
        const filePath : string = path.join(saveDirectory, `icon${index + 1}.svg`);

        // El objeto Buffer podemos guardarlo en un archivo
        await fs.writeFile(filePath, iconData);
        console.log(`Icono ${index + 1} descargado con éxito: ${filePath}`);

    } catch (error) {
        console.error(`Error descargando el icono ${index + 1} desde ${url}: ${error.message}`);
    }
}
```

Finalmente implementamos la función downloadAllIcons(), que utiliza las anteriores para descargar todos los iconos. Utiliza Puppeteer para abrir la página y con la función getIconUrls() obtiene los enlaces a los iconos. Por cada uno de ellos, llama a la función downloadIcon() con la información correspondiente para descargar el icono. En un principio hacíamos las solicitudes paralelamente, pero eso provocaba una sobrecarga en el servidor y dejaba de responder correctamente. Por eso añadimos una pausa de un segundo entre descargas.

2. IMPLEMENTACIÓN DE LA SECCIÓN FRONT-END

Durante este punto utilizaremos la lista de JSON generadas en el punto anterior para añadir de manera dinámica los hexágonos que representan las skills a superar, además de darle un estilo a la página mediante un página de estilo CSS proporcionada en el enunciado. Vamos a añadir en el HTML la referencia tanto al CSS como al script de JavaScript que añadirá y creará los hexágonos.

Ejercicios

La idea como se ha mencionado antes es añadir de manera dinámica, es decir, en vez de crear uno a uno los hexágonos, crearlos a través de iterar sobre el array de JSONs que representan los skills existentes. Cada JSON tiene dentro un id, un texto y un icono.

Para iterar, accederemos usando la ruta relativa donde están ubicados los JSON y por cada elemento crearemos un SVG que estará compuesto de un contenedor donde se agruparan todos los elementos: el texto, el icono y el propio hexágono.

En primer lugar creamos el SVG y el contenedor donde añadiremos los elementos. A continuación también creamos el hexágono.

```
// Contenedor
const svgWrapper :HTMLDivElement = document.createElement( tagName: 'div');
svgWrapper.classList.add('svg-wrapper');
svgWrapper.setAttribute( qualifiedName: 'data-id', skill.id);
svgWrapper.setAttribute( qualifiedName: 'data-custom', value: 'false');

// SVG
const svg :SVGSVGElement = document.createElementNS( namespaceURI: 'http://www.w3.org/2000/svg', qualifiedName: 'svg');
svg.setAttribute( qualifiedName: 'width', value: '100');
svg.setAttribute( qualifiedName: 'height', value: '100');
svg.setAttribute( qualifiedName: 'viewBox', value: '0 0 100 100');

// Hexágono
const polygon :SVGPolygonElement = document.createElementNS( namespaceURI: 'http://www.w3.org/2000/svg', qualifiedName: 'polygon');
polygon.setAttribute( qualifiedName: 'points', value: '50,5 95,27.5 95,72.5 50,95 5,72.5 5,27.5');
polygon.classList.add('hexagon');
svg.appendChild(polygon); // Lo metemos dentro del SVG
```

En segundo lugar, para el texto vamos a crear el objeto donde indicamos las características que va a tener el cuadro de texto. Luego vamos a crear un objeto tspan para cada línea de texto que tiene cada skill separadas por el salto de línea, para luego añadir dicho objeto al texto que será añadido al SVG..

```
// Objeto texto
const text :SVGTextElement = document.createElementNS( namespaceURI: 'http://www.w3.org/2000/svg', qualifiedName: 'text');
text.setAttribute( qualifiedName: 'x', value: '50%');
text.setAttribute( qualifiedName: 'y', value: '20%');
text.setAttribute( qualifiedName: 'text-anchor', value: 'middle');
text.setAttribute( qualifiedName: 'fill', value: 'black');
text.setAttribute( qualifiedName: 'font-size', value: '9.5');
```

Por último, de la misma manera en la que hemos accedido al texto de los JSON, obtendremos el nombre del icono que representa la skill que estamos ilustrando por medio del SVG, para utilizarlo para acceder al icono correcto de la carpeta icons, dónde están los iconos almacenados utilizando la ruta relativa correcta. Creamos el objeto y lo añadimos al SVG que será a su vez añadido al contenedor para ser añadido a la página.

```
svg.appendChild(text); // Metemos el texto dentro del SVG

// Icono
const image :SVGImageElement = document.createElementNS( namespaceURI: 'http://www.w3.org/2000/svg', qualifiedName: 'image');
image.setAttribute( qualifiedName: 'x', value: '35%');
image.setAttribute( qualifiedName: 'y', value: '60%');
image.setAttribute( qualifiedName: 'width', value: '30');
image.setAttribute( qualifiedName: 'height', value: '30');
image.setAttribute( qualifiedName: 'href', value: `../electronics/icons/${skill.icon}`);
svg.appendChild(image); // Metemos el icono en el SVG

svgWrapper.appendChild(svg); // Metemos el SVG en el contenedor
svgContainer.appendChild(svgWrapper);
```

Este proceso será repetido por cada JSON hasta crear todas las skills.

3. MEDALLAS Y PUNTOS

En este punto obtendremos la información y los iconos correspondientes a las medallas que el usuario ganará mediante bitpuntos durante el aprendizaje. Las obtenemos de la página https://github.com/Obijuan/digital-electronics-with-open-FPGAs-tutorial/wiki#listado-de-rangos.

Ejercicios

Lo primero que haremos será obtener la información de las medallas (rango, bitpuntos necesarios y nombre del icono) y guardarla en un archivo badges.json. Dicha información se encuentra dentro de diferentes tablas en la página. Para extraerla, definimos el rango de índices de las tablas en las que está la información y una función extractTablesInRange().

Utilizamos Puppeteer para navegar por los elementos de la página. A la función page.evaluate() le pasamos un array vacío donde guardar todos los datos de las diferentes tablas y una variable para contar los rangos de bitpuntos. Dentro, seleccionamos todas las tablas y obtenemos el rango que necesitamos donde se encuentra la información de las medallas.

```
const { tablesData :(FlatArray<...>)[] , updatedBitpointCount :number } = await page.evaluate(
    pageFunction: (startIndex :U , endIndex :U , bitpointCount :number ) :{tablesData: ..., updatedBitpointCount: any} => {
    const tables :NodeListOf<HTMLTableElement> = document.querySelectorAll( selectors: 'table');

    // De todas las tablas, seleccionamos las que están en el rango que queremos
    const selectedTables :HTMLTableElement[] = Array.from(tables).slice(startIndex, endIndex + 1);
```

Después obtenemos la información necesaria de cada tabla. Seleccionamos las filas y quitamos la primera, que es la cabecera de la tabla. Por cada fila, generamos un objeto JSON llamado rowData con la información que requiere. El rango lo obtenemos de la columna de índice 2 de la tabla; los bitpoints máximos y mínimos los calculamos con la variable bitpointCount; y el nombre del icono lo obtenemos de la ruta a la imagen, separándola y obteniendo el último segmento. Al final actualizamos la variable bitpointCount para que los siguientes rangos sigan sumando puntuaciones cada 10 puntos.

```
const tablesData :(...)[] = selectedTables.map((table :HTMLTableElement ) :({...})[] => {

    // Seleccionamos las filas de la tabla
    const rows :HTMLTableRowElement[] = Array.from(table.querySelectorAll( selectors: 'tr'));
    rows.shift(); // Quitamos la cabecera de la tabla (primera fila)
    const tableData :({...})[] = rows.map((row :HTMLTableRowElement ) :{...} => {
        const cells :(T)[] = Array.from(row.querySelectorAll( selectors: 'td, th'));
        // Devolvemos los datos necesarios
        const rowData :{...} = {
            "rango": cells[2].innerText.trim(),
            "bitpoints-min": bitpointCount.toString(),
            "bitpoints-max": (bitpointCount + 9).toString(),
            "png": cells[1].querySelector('img').getAttribute( qualifiedName: 'src').split( separator: '/').pop()
        };
        bitpointCount += 10;
        return rowData;
    });
    return tableData;
});
```

Descargar iconos

La función también debe descargar las versiones minimizadas de los iconos para cada medalla. Definimos las funciones downloadBadge() y downloadAllBadges() siguiendo una lógica similar a cuando descargamos los iconos de las competencias en el primer punto.

Para garantizar que descargamos las versiones minimizadas, añadimos el sufijo “-min” a los nombres de las imágenes, que forman la ruta a la versión minimizada de cada imagen. También añadimos una pausa entre solicitudes de descarga para no sobrecargar el servidor.

Finalmente, llamamos a la función downloadAllBadges() dentro de la función extractTablesInRange(), pasándole la información obtenida entre la que se incluyen los nombres de los iconos a descargar.

```
allTablesData = allTablesData.concat(tablesData);
bitpointCount = updatedBitpointCount;

console.log(JSON.stringify(allTablesData,  replacer: null,  space: 2));

await browser.close();

await fs.writeFile(path.resolve(__dirname, '..', 'public', 'badges', 'badges.json'),
    JSON.stringify(allTablesData,  replacer: null,  space: 2));
console.log('Se han guardado los datos en public/badges/badges.json');

await downloadAllBadges(allTablesData);
```

Página para mostrar la información

Para mostrar la información obtenida, creamos una página leaderboard.html. Dentro de la misma creamos una estructura básica donde mostraremos la tabla, pero se cargará de forma dinámica para obtener la información del archivo badges.json y los iconos descargados previamente.

La carga dinámica la definimos en una función loadLeaderboard() en el archivo leaderboard.js, que insertaremos como source en la cabecera de la página. Le pasamos la información de las medallas que hemos guardado antes, y de ella construimos el interior de la tabla. Por cada columna de la tabla (llamada cell en el código, cada celda de una fila), creamos un elemento <td> donde insertamos la información. En cada fila guardaremos el nombre del rango, la imagen de la medalla y el rango de bitpuntos para conseguirla. Cada elemento fila <tr> lo insertamos en la tabla de la página con appendChild(), después de haber insertado cada celda <td> dentro.

```
const tbody = document.getElementById('leaderboard-body');
data.forEach((item, index) => {
    const row = document.createElement('tr');

    const rangoCell = document.createElement('td');
    rangoCell.textContent = item.rango;
    row.appendChild(rangoCell);

    const badgeCell = document.createElement('td');
    const img = document.createElement('img');
    img.src = `badges/${item.png.replace('.png', '-min.png')}`;
    badgeCell.appendChild(img);
    row.appendChild(badgeCell);

    const bitpointsCell = document.createElement('td');
    if (index === data.length - 1) {
        bitpointsCell.textContent = `${item['bitpoints-min']} <`;
    } else {
        bitpointsCell.textContent = `${item['bitpoints-min']} - ${item['bitpoints-max']}`;
    }
    row.appendChild(bitpointsCell);

    tbody.appendChild(row);
});
```

La imagen la obtenemos de la carpeta badges, donde hemos descargado los iconos previamente. Para los bitpuntos, hacemos un tratamiento especial cuando se trata del último elemento de la tabla (la última fila), para no mostrar el rango de puntos cerrado sino un rango abierto por arriba (para obtener la última medalla basta con superar el mínimo de puntos).

4. CONTROL DE USUARIOS

Para simular el acceso a la aplicación de diferentes usuarios, hemos creado un sistema simple de control de usuarios utilizando el localstorage. Para poder probar a visualizar la aplicación desde perspectivas de diferentes usuarios (sin necesidad de hacer login), hemos implementado un panel de control en la pantalla principal. El panel cuenta con la información de cinco usuarios. Los primeros cuatro son usuarios estándar (rol “user”), alumnos que utilizarán la plataforma; y el quinto es un administrador (rol “admin”).

```html
<div class="user-panel">
    <label for="userSelect">Select User:</label>
    <select id="userSelect">
      <option value="user1" data-role="user">User 1 (User)</option>
      <option value="user2" data-role="user">User 2 (User)</option>
      <option value="user3" data-role="user">User 3 (User)</option>
      <option value="user4" data-role="user">User 4 (User)</option>
      <option value="user5" data-role="admin">User 5 (Admin)</option>
    </select>
    <button id="applyUser">Apply</button>
    <span id="welcomeMessage"></span>
</div>
```

El código para el funcionamiento del panel está implementado en userPanel.js. Al iniciar la aplicación, si no hay ningún usuario seleccionado, se determina que se ha “iniciado sesión” como el usuario “user1”.

Después configuramos un event listener para el botón del panel, donde obtenemos la información de la lista desplegable y la guardamos en localstorage.

```javascript
// Event listener cuando se aplique el usuario escogido
applyUserButton.addEventListener( type: 'click', listener: () : void => {
    const selectedUser = userSelect.options[userSelect.selectedIndex];
    const userName = selectedUser.value;
    const userRole : string = selectedUser.getAttribute( qualifiedName: 'data-role');

    // Guarda en localstorage y alerta del cambio
    localStorage.setItem('currentUser', userName);
    localStorage.setItem('currentUserRole', userRole);

    alert(`Usuario cambiado a ${userName} con el rol ${userRole}`);
    updateWelcomeMessage();
    location.reload();
});
```

De esta forma, guardaremos el localstorage el nombre del usuario actual y su rol. Esta información será utilizada en la implementación de los puntos posteriores.

5. COMPETENCIAS Y TAREAS

Por cada skill de la página, debe aparecer un lápiz y un cuaderno cuando el ratón esté encima de este, aparte de un cuadrado informativo amarillo en la parte inferior de la pantalla. Dicho cuaderno, al hacer click en este, no va a mostrar un formulario que

representa las tareas a superar para dar como completada dicha skill. En este formulario debe aparecer el nombre de la skill como título, una descripción, el icono y la lista de tareas a realizar. Además de esto, al seleccionar todas las tareas aparecerá un cuadro de texto para rellenar con recursos relacionados con la competencia y un botón que almacene dicha información. Al hacerlo también se ha creado una animación de confeti. Es la mejor parte.

Añadir el cuaderno y el lápiz al contenedor, y el cuadrado amarillo

Utilizando el contenedor donde se han añadido los elementos de la skill, vamos a añadir un nuevo elemento a este. En primer lugar vamos a hacer fetch de los emojis utilizando la página: https://www.reshot.com, y vamos a obtener los emoticonos correspondientes.

```
window.onload = function() : void {
    // Promesa para el icono del lápiz
    const fetchPencil : Promise<string> = fetch( input: 'https://www.reshot.com/preview-assets/icons/U3A6CNXBDH/pencil-U3A6CNXBDH.svg')
        .then(response : Response => {
            if (!response.ok) {
                throw new Error(`HTTP error! status: ${response.status}`);
            }
            return response.text();
        });

    // Promesa para el icono del cuaderno
    const fetchNotebook : Promise<string> = fetch( input: 'https://www.reshot.com/preview-assets/icons/UVG3NADPR2/note-book-UVG3NADPR2.svg')
        .then(response : Response => {
            if (!response.ok) {
                throw new Error(`HTTP error! status: ${response.status}`);
            }
            return response.text();
        });

    Promise.all( values: [fetchPencil, fetchNotebook]) Promise<...>
        .then(([pencilSvg : string , notebookSvg : string ]) : void => {
            appendEmoji(pencilSvg, className: 'emojiLapiz'); // Insertamos el lápiz en todos los hexágonos
            appendEmoji(notebookSvg, className: 'emojiCuaderno'); // Insertamos el cuaderno en todos los hexágonos
            eventManager(); // Después llamamos a eventManager, para garantizar que el cuaderno está cargado
        }) Promise<void>
        .catch(error => {
            console.error('Error al cargar los iconos de edición e información.', error);
        });
```

Una vez obtenidos los emoticonos, hay que añadirlos en los contenedores de todos los hexágonos.

```
// Para insertar un emoji dentro de todos los hexágonos
function appendEmoji(svgContent, className) : void {  Show usages  euken13 +1
    const containers : NodeListOf<Element> = document.querySelectorAll( selectors: '.svg-wrapper');
    containers.forEach( callbackfn: (cont : Element ) : void => {
        const tempDiv : HTMLDivElement = document.createElement( tagName: 'div');
        tempDiv.innerHTML = svgContent;
        const svgElement : ChildNode = tempDiv.firstChild;
        svgElement.classList.add('emojiContainer');
        svgElement.classList.add(className);
        cont.appendChild(svgElement);
    });
}
```

Como último añadido, la barra amarilla inferior debe ser creada y añadida.

```
// Para crear la barra donde se visualiza la descripción de la skill
function createLowerBanner() : void  {  Show usages  Unai Leon Plaza
    const descriptionBanner :HTMLDivElement  = document.createElement( tagName: 'div');
    descriptionBanner.id = 'description-banner';
    descriptionBanner.classList.add('description-banner');
    document.body.appendChild(descriptionBanner);
}
```

Una vez creados los elementos que van a interactuar con el ratón, hay que implementar dicha funcionalidad. Para ello, hacemos uso de un event manager, que detectara (mouseover) cuando el ratón esté sobre el elemento concreto y cambiará el display de la barra y la mostrará junto con el id de la skill en la que está posando. Cuando deje de detectar el ratón (mouseout), esconde la barra de nuevo. En el caso del cuaderno y el lapiz, se ha incluido un :hover en la página de diseño CSS donde agranda el icono y muestra los emojis cambiando su display a visible.

```
// Para que aparezca la descripción al pasar el ratón por los skills
document.querySelectorAll( selectors: '.svg-wrapper').forEach( callbackFn: wrapper : Element  => {
    wrapper.addEventListener( type: 'mouseover',  listener: () : void  => {
        let banner : Element  = document.querySelector( selectors: '.description-banner');
        banner.style.display = 'block';
        banner.innerHTML = `Descripción de la tarea: ${wrapper.getAttribute( qualifiedName: 'data-id')}`;
    });
    wrapper.addEventListener( type: 'mouseout',  listener: () : void  => {
        document.querySelector( selectors: '.description-banner').style.display = 'none';
    });
});
```

Creación de la nueva página donde muestra el formulario

Para ello, hemos creado un nuevo HTML el cual hará uso de la página de diseño CSS proporcionada para la creación de la interfaz de los hexágonos. Para los scripts, en cambio, hemos decidido crear un nuevo código JavaScript donde implementaremos las funcionalidades.

Como comienzo, necesitamos recopilar la información de la skill en la que el formulario se centra. Para ello, hemos creado dos funciones que van a proporcionarnos tanto el icono a mostrar en la página como el nombre de la skill. Haciendo uso del localStorage, almacenamos el SVG de la skill en la que hemos hecho click y la añadimos a la página. Para mostrar el título en cambio, hemos almacenado la ID de la skill para, usando la ruta relativa adecuada, acceder a los JSON donde buscaremos el título al que dicha ID hace referencia (.find), para añadirla a la página sustituyendo los salto de línea por huecos en blanco.

```
function loadSkillHexagon() : void { Show usages  Unai Leon Plaza
    const skillHexagon : string  = localStorage.getItem( key: 'skillHexagon');
    if (skillHexagon) {
        const svgContainer : HTMLDivElement  = document.createElement( tagName: 'div');
        svgContainer.classList.add('svg-container');
        svgContainer.innerHTML = skillHexagon;
        document.querySelector( selectors: '.hexagon-container').appendChild(svgContainer);
    }
}

// Función para cargar la información de la skill dada en la página
function loadSkillInformation(skillId) : void { Show usages  Unai Leon Plaza +1
    fetch( input: 'electronics/skills.json') Promise<Response>
        .then(response : Response  => response.json()) Promise<any>
        .then(skills => {
            const skillFind = skills.find(item => (item.id).toString() === skillId);
            if (skillFind) {
                let text = (skillFind.text).replace(/\n/g, " ");
                document.getElementById( elementId: 'title').innerText = 'Skill: ' + text;
            } else {
                console.log("ERROR: no se ha encontrado la información de la competencia a cargar.");
            }
        });
}
```

En dicha página debemos añadir también un botón y un cuadro de texto que están ocultos hasta que el usuario ha seleccionado todas las tareas de la competencia. Para ello hay que entender los checkboxes como listas en donde todas ellas tienen que tener el atributo checked activo, de esta forma cada vez que se cambia el estado de uno de ellos, se hace una comprobación para saber si todas han sido seleccionadas o no, para en caso afirmativo, mostrar los elementos ocultos, y realizar la animación del confeti obtenida usando los scripts de canvas-confetti.

```
// Para llamar cada vez que se marque o desmarque una tarea (checkbox)
function checkBoxVerify() : void { Show usages  Unai Leon Plaza +1
    const checkboxes : NodeListOf<Element>  = document.querySelectorAll( selectors: '.checkbox');
    const allCheck : boolean  = Array.from(checkboxes).every(checkbox : Element  => checkbox.checked);
    const textBoxTittle : HTMLElement  = document.getElementById( elementId: 'textBoxTitle');
    const textBox : HTMLElement  = document.getElementById( elementId: 'textBox');
    const buttonSubmit : HTMLElement  = document.getElementById( elementId: 'buttonSubmit');

    // Si todas están marcadas mostramos el formulario + confetti
    if (allCheck) {
        textBoxTittle.style.display = 'block';
        textBox.style.display = 'block';
        buttonSubmit.style.display = 'block';
        confetti({
            particleCount: 150,
            spread: 70,
            origin: { y: 0.6 }
        });
    } else {
        // Si hay alguna sin marcar lo ocultamos
        textBoxTittle.style.display = 'none';
        textBox.style.display = 'none';
        buttonSubmit.style.display = 'none';
    }
```

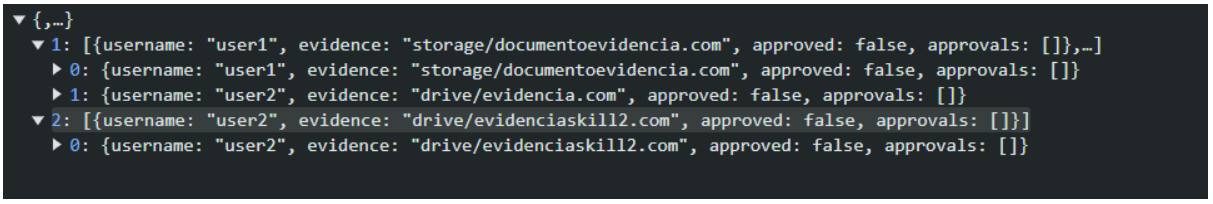
6. EVIDENCIAS Y CONFIRMACIÓN DE COMPETENCIA

Cuando el usuario haya terminado todas las tareas, aparecerá en la pantalla el formulario donde podrá escribir y enviar una evidencia.

Gestión de las evidencias enviadas en localstorage

Por ahora guardaremos la información sobre las evidencias en localstorage. Creamos un objeto JSON llamado “evidencias”, que guardará un array de evidencias para cada competencia. Dentro de dicho array, guardaremos la información que necesitaremos sobre la evidencia: el usuario que la envió, el texto en sí, un atributo booleano que indique si está aprobada o no y un número de confirmaciones de otros usuarios.

En la siguiente imagen se muestra un ejemplo de como se organiza la estructura descrita. Para la skill con id 1, se han enviado dos evidencias. Una de ellas es del usuario “user1” y la otra del usuario “user2”. Para la skill con id 2, solo se ha enviado una evidencia; en este caso, por el “user2”.



Punto de notificación en las competencias de la pantalla principal

Cuando se envíe una evidencia para una competencia, deberá aparecer un círculo rojo en su hexágono. Dentro del círculo aparecerá el número de evidencias que se han enviado para la competencia. Cuando la evidencia que un usuario ha enviado sea aprobada, se considerará que ha obtenido la competencia; coloreando el hexágono y punto de información correspondientes en verde.

Para lograr esto, definimos la función loadNotificationDots() en main.js. Obtiene la lista de evidencias guardadas en el localstorage y dibuja los puntos de notificación en los hexágonos de las skills que contengan alguna evidencia.

```
// Si hay alguna evidencia, se crean los círculos de notificación
if (evidenceCount > 0) {
    const svgWrapper :Element = document.querySelector( selectors: `.svg-wrapper[data-id="${skillId}"]`);
    if (svgWrapper) {
        const redDot :HTMLDivElement = document.createElement( tagName: 'div');
        const greenDot :HTMLDivElement = document.createElement( tagName: 'div');
        redDot.classList.add('redNotification');
        greenDot.classList.add('greenNotification');
        redDot.innerText = evidenceCount;
        greenDot.innerText = evidenceCount;
        svgWrapper.appendChild(redDot);
        svgWrapper.appendChild(greenDot);
```

La configuración de estilo de los círculos se realiza en style.css, utilizando las clases redNotification y greenNotification. Inicialmente, el punto rojo es visible y el punto verde no lo es, utilizando la propiedad “display”.

En la misma función, comprobamos si hay alguna evidencia que el usuario actual haya enviado y que haya sido aprobada. En ese caso, colorearemos el hexágono de la competencia en verde.

```
// Si hay alguna evidencia que el usuario actual ha publicado y está aprobada,
// llamamos a la función approveEvidence que coloreará el hexágono en verde
const userEvidence = evidenceList.find(evidence => evidence.username === currentUser);
if (userEvidence && userEvidence.approved) {
    approveEvidence(skillId);
}
```

Para colorear el hexágono hemos definido otra función, approveEvidence(). Pasándole un identificador de competencia, obtiene el hexágono correspondiente y lo pinta de verde utilizando la propiedad de estilo “fill”. Después cambia la visibilidad de los círculos para ocultar el círculo rojo (cambiando la propiedad “display” a “none”) y mostrar el círculo verde (cambiando la propiedad “display” a “flex”).

```
// Para colorear en verde una skill y cambiar el punto rojo por otro verde (cuando esté aprobada)
function approveEvidence(skillId) : void  {  Show usages  Unai Leon Plaza
    const svgWrapper : Element  = document.querySelector( selectors: `.svg-wrapper[data-id="${skillId}"]`);
    const hex : Element  = svgWrapper.getElementsByClassName( classNames: 'hexagon')[0];
    if (hex) {
        hex.style.fill = '#2afa2a';
    }
    const redDot : Element  = svgWrapper.getElementsByClassName( classNames: 'redNotification')[0];
    const greenDot : Element  = svgWrapper.getElementsByClassName( classNames: 'greenNotification')[0];

    // Utilizamos la propiedad de estilo "display" para hacer los elementos aparecer y desaparecer
    if (redDot) {
        redDot.style.display = 'none'; // No aparece
    }

    if (greenDot) {
        greenDot.style.display = 'flex'; // Se posiciona en la pantalla
    }
}
```

Confirmación de evidencias

Cuando una competencia tiene alguna evidencia enviada, se muestran todas las evidencias que se han enviado en una tabla. La tabla se encuentra en la pantalla que muestra la información sobre una competencia concreta (skillspecifics.html). Contendrá la información de cada evidencia (usuario y texto) y una columna más para controlar la aprobación.

Si el usuario actual es un usuario estándar, únicamente verá un botón para aprobar las evidencias que se han enviado. En cambio, si el usuario es administrador, verá un botón para aprobar la evidencia y otro para rechazarla. Al lado de los botones aparecerá un texto que indique cuántos alumnos (usuarios estándar) han aprobado la evidencia. Una skill se considerará completada para un usuario cuando otros tres usuarios hayan aceptado su evidencia, o un único administrador lo haya hecho.

Para cargar las evidencias en la tabla implementamos la función loadAndDisplayEvidences() en skillspecifics.js. Obtiene las evidencias enviadas para la skill actual y crea una fila de la tabla por cada una.

En la primera columna añade el usuario, en la segunda el texto de la evidencia. En la tercera columna comprueba si el rol del usuario actual es “admin”, en cuyo caso crea dos botones (para aprobar o rechazar). En el caso contrario, si el usuario no es administrador, crea solo un botón para aprobar. Al final, siempre añade una etiqueta “span” con el número de confirmaciones.

```
row.innerHTML = `
    <td>${evidence.username}</td>
    <td>${evidence.evidence}</td>
    <td>
        ${evidence.approved ? `
            <span class="approved-label">Approved</span>
        ` : `
            ${currentUserRole === 'admin' ? `
                <button class="approve-btn" data-index="${index}">Approve</button>
                <button class="reject-btn" data-index="${index}">Reject</button>
            ` : `
                <button class="approve-btn" data-index="${index}">Approve</button>
            `}
            <span class="approval-count">(${approvalCount} approvals)</span>
        `}
    </td>
`;
evidenceTableBody.appendChild(row);
```

Después añade los event listeners correspondientes a los botones que se han definido, teniendo siempre en cuenta si el usuario es administrador o no.

```
// Añadimos event listeners a los botones creados para llamar a las funciones correspondientes
document.querySelectorAll( selectors: '.approve-btn').forEach( callbackfn: button : Element  => {
    button.addEventListener( type: 'click', handleApprove);
});

if (currentUserRole === 'admin') {
    document.querySelectorAll( selectors: '.reject-btn').forEach( callbackfn: button : Element  => {
        button.addEventListener( type: 'click', handleReject);
    });
}
```

Como vemos en los event listeners anteriores, para controlar la confirmación o rechazo de las evidencias definimos otras dos funciones: handleApprove() y handleReject().

En handleApprove() definimos que si el usuario es administrador, la evidencia quedará directamente aprobada. Si el usuario no lo es, sumaremos su confirmación al número de confirmaciones de la evidencia; y si llega a tres, entonces se aprobará la evidencia.

```
if (currentUserRole === 'admin') {
    // Si es admin directamente aprueba la evidencia
    evidencias[skillId][index].approved = true;
} else {
    // Si no es admin sumamos un aprobado, y solo cuando sean 3 se aprobará la evidencia
    if (!evidencias[skillId][index].approvals) {
        evidencias[skillId][index].approvals = [];
    }

    if (!evidencias[skillId][index].approvals.includes(currentUser)) {
        evidencias[skillId][index].approvals.push(currentUser);
    }

    if (evidencias[skillId][index].approvals.length >= 3) {
        evidencias[skillId][index].approved = true;
    }
}
```

En handleReject(), simplemente eliminamos la evidencia de localstorage.

7. CONCLUSIÓN

Finalizamos el desarrollo con el control de confirmación de competencias, terminando la parte del front end de la aplicación final. También hemos implementado una parte importante de la lógica detrás del front end: posibilita la simulación de diferentes escenarios dependiendo del usuario y facilita la adaptación del código cuando se implemente la sección del back end.