Javascript

TB022 - Esteban - Riesgo - Kristal

Qué es Javascript

Javascript es un lenguaje de programación multiparadigma (estructurado, funcional, orientado a objetos), de tipado dinámico, débilmente tipado.

Fue creado en 1995 en un lapso de 10 días y fue específicamente diseñado para ser usado en el navegador Netscape (el más popular de la época) y luego adaptado por Internet Explorer donde se hizo inmensamente popular.

Hoy en día es usado por todos los navegadores y es el lenguaje default para agregar comportamiento del lado del cliente, el Frontend.

Hay muchas alternativas para escribir código: Typescript, Elm, Clojurescript pero todos compilan a JS in the end. No hay escapatoria.

Javascript

Vamos a ver lo mínimo e indispensable, en otras palabras vamos a ver:

Instalación

Para poder escribir código en Javascript, no hay que instalar Javascript, hay que instalar NodeJS.

Dejamos las mismas recomendaciones que con Python: no instalar directamente en el sistema, sino usar un manejador de versiones que nos permite tener ambientes virtuales y poder cambiar la versión usada según necesitemos.
Para ello vamos a instalar nvm

```
$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.1/install.sh | bash
```

Y ahora vamos a instalar la última versión con

```
$ nvm install node
```

Además de NodeJS, nos intalará `npm`, el manejador de paquetes. Con éste instalaremos todas las dependencias qué necesitemos.

Sintaxis Básica: variables

Podemos declarar variables con `let`

```
let variable = "valor";
```

O podemos declarar constantes con `const`

```
const constante = "valor";
```

Sí, hay que poner los `;` al final.

Sintaxis Básica: funciones

```
function nombreFuncion(arg1, arg2) {
    return arg1 + arg2
}
```

Se usa `camelCase` para el nombre de funciones y variables.

Otra forma de definir funciones es con la sintaxis de *arrow functions*

```
let nombreFuncion = (arg1, arg2) => {
    arg1 + arg2
}
```

Las arrow functions tienen auto-return.

El último valor evaluado por la función será devuelto.

Se puede usar `return` para terminar antes la ejecución.

Sintaxis Básica: arreglos

```
let arr = ["Hidrógeno", "Helio", "Litio", "Berilio"];

arr[0] // Hidrógeno

arr[arr.length-1] // Berilio

arr.push(“Oxígeno”)

for (let elemento of arr) {
    ...
}
```

Podemos iterar el arreglo con un far similar al de Python (hay otras maneras también pero esta es la más fácil)

Sintaxis Básica: objetos

Los "objetos" en JS funcionan como los diccionarios de Python:

```
obj = {
    "claveA": 1,
    "claveB": 2,
}
obj["claveA"] // 1
```

```
obj = {
    claveA: 1,
    claveB: 2,
}
```

Otra forma de definir el mismo objeto

pero además, podemos acceder directamente por el nombre del campo si la clave es string

```
obj.claveA // 1
```

Sintaxis Extra: objetos

Es muy común asignar una variable a un atributo de un objeto con el mismo nombre.

Tan común que esto:

```
nombre = "Fede"

obj = {

    nombre: nombre,

}
```

se puede escribir de manera más corta:

```
nombre = "Fede"

obj = {

    nombre,

}
```

Sintaxis Extra: string interpolation

```
nombre = "Fede"

saludo = `Hola ${nombre}`     // Hola Fede
```

Funcionalidad Básica: “print” by JS

```
nombre = "Fede"

console.log(nombre)

console.log(`Hola ${nombre}`)
```

Javascript en HTML

Podemos correr código Javascript en nuestros archivos HTML. Los navegadores web interpretarán el código JS y lo ejecutarán.

```
<script>
    let variable = "Variable";
    alert(variable);
</script>
```

Escribir código Javascript "vainilla" no es muy fácil.
Hoy en día existen muchas librerías y frameworks que facilitan el desarrollo FE:

- Angular (y AngularJS)
- React
- NextJS
- VueJS
- Svelte

Hay uno nuevo por semana. Nosotros vamos a usar **Svelte**.

Svelte

Es un framework que compila a Javascript y nos permite manejar código del lado del cliente más fácilmente.

Tiene una librería muy pequeña pero es igual de poderoso que otros frameworks más complejos.

Svelte en sí mismo se usa como una biblioteca que se puede embeber en otros frameworks. En nuestro caso vamos a usar SvelteKit, que es un framework completo con todo lo que necesitamos tener.

Vamos a ver lo mínimo y necesario para poder armar el sitio que necesitamos para el TP.

Sveltekit: inicializar un proyecto

Para crear un nuevo proyecto debemos correr lo siguiente en el directorio donde queramos crearlo:

```
npx sv create <dir_name>
```

y debemos seguir las instrucciones interactivas con lo que queramos. En nuestro caso, elegiremos SvelteKit minimal.

Ya podemos correr nuestro servicio posicionandonos en el directorio y corriendo

```
npm run dev
```

Sveltekit: components

Una aplicación hecha en Svelte está compuesta de uno o más components. Un componente es un bloque de código que combina HTML, CSS y Javascript en un archivo `.svelte`

Cada ruta (página distinta de nuestra aplicación) tendrá su directorio en `/routes` y un archivo `+page.svelte` que se renderizará al ir a esa ruta.

Por defecto, el archivo `+page.svelte` que no está en ningún subdir es el home.

Creemos un par de páginas distintas en nuestra aplicación.

Sveltekit: rutas

Agreguemos una página para mostrar alumnos y una para mostrar grupos de TP.

Una vez que agregamos los archivos de rutas le ponemos algún contenido (no es importante qué contenido tienen por ahora, por ejemplo podría ser

`<h1>Esta es la página de ...</h1>` para materias y para grupos

Ya podemos acceder a estas páginas si agregamos el path en la URL.

Pero estaría mejor que haya *algo* en el Home (y en las demás páginas) que nos permita navegar entre ellas.

Agreguemos una *nav bar*.

Sveltekit: rutas

Una nav bar (navigation bar) es comúnmente utilizada como un recurso que nos permite navegar entre las distintas páginas de nuestro sitio.

Puede ser de cualquier tipo y forma. Siempre y cuando usemos el tag HTML <nav>

```
<nav>
    <ul>
        <li>
            <a href="/">Home</a>
        </li>
        <li>
            <a href="/alumnos">Alumnos</a>
        </li>
        <li>
            <a href="/grupos">Grupos</a>
        </li>
    </ul>
</nav>
```

El tag <a> define hyperlinks pueden llevar a páginas del mismo sitio o a cualquier otro sitio en Internet.

Nuestra nav bar es una lista con tres elementos.
Si hacemos click en alguno nos llevará a la página correspondiente.

El problema ahora es como volver. Estaría bueno que la nav bar esté en todas las páginas para hacer la navegación más fácil y placentera.

Qué podemos hacer para no copiar y pegar en cada página?

Sveltekit: reusable components

Podemos crear un componente que sea nuestra nav bar y simplemente utilizarlo desde todas las páginas.

Debemos crear un componente en algún lugar (podemos ponerlo en el mismo root dir de routes donde está definido el home).

Mover el contenido que querramos ahí.

Y después importarlo en cada página y...

No ganamos demasiado. Seguimos teniendo que ponerlo a mano.

Queremos que nuestro nav bar esté presente en todas las páginas.

Vamos a definir entonces un componente especial llamado `+layout.Svelte`

Sveltekit: +layout

Define como es el layout común de todo el sitio.

```
<script>
   import Header from './Header.svelte';
   import Footer from './Footer.svelte';

</script>

<div class="app">
   <Header />

   <main>
       <slot />
   </main>

   <Footer />
</div>
```

Este es el típico layout de una aplicación cualquiera.

Tenemos un header (que incluye el nav bar y puede incluir otras cosas).

Un footer (generalmente con información del sitio, contactos, copyright, etc.).

Y en el medio, el contenido.

Estamos diciendo al framework que queremos tener primero siempre un header, luego el contenido de cada página, y finalmente el footer, siempre.

Sveltekit: estilo

Ahora que nuestra nav bar funciona como queríamos, nos falta darle estilo.

Si queremos usar estilos que sólo se apliquen a ese componente, podemos definir una sección `<style>` en el componente que queremos y sólo va a ser válido allí.
Si queremos definir y usar estilos más genéricos podemos crear archivos .css, ubicarlos donde deseemos (si es usado por todos los componentes de una ruta, en el dir de la ruta; por varias rutas, en el home) e importarlos

```
import '<path_to_file>.css';
```

y se aplicarán los estilos según las reglas que ya vimos.

Sveltekit: cargar datos de una API

Ahora sí, por fin. Vamos a usar el backend que ya teníamos andando, para conseguir información y mostrarla en nuestro FE.

Debemos usar la función fetch de Javascript que nos permite pasarle una URL, body, argumentos y demás y nos devolverá en un objeto la response.

Javascript: fetch

```
fetch('https://<url>?param1=value&param2=value2', {
    method: '<method>',
    body: JSON.stringify({
        ...
    })
})
```

Un poco más prolijo para armar la URL

```
let url = new URL('https://<url>.com')

let params = {param1:value, param2:value2}
url.search = new URLSearchParams(params).toString();
url.headers = {...}

fetch(url, {
    method: 'GET'
})
```

Javascript: fetch, cómo usarla

Volviendo a nuestro ejemplo, debemos "pegarle" a nuestro servidor local para obtener los alumnos y poder mostrarlos.

Podríamos llamar directamente a fetch desde el componente (el archivo .svelte) pero en este caso no, porque fetch es una función especial.

`fetch` no devuelve el contenido directamente, sino que lo que devuelve es una *Promise*. Javascript se encargará de ejecutar el llamado y traer el resultado eventualmente, es una promesa de que va a ejecutar ese código que le digamos. Para poder leer el resultado debemos *esperar* a que esté listo, y luego continuar.

Esto lo logramos con la sentencia `await`. Como no podemos poner un await directamente en un componente de svelte, debemos tener un archivo .js aparte para poder lograrlo.

Svelte: +page.server

Junto a nuestro archivo `+page.svelte` de cada ruta que deseemos tener, podemos tener un archivo extra que debe llamarse `+page.server.js`el cual se ejecutará por defecto al cargar esa ruta.

```
export async[1] function load() {
    let url = new URL('http://localhost:8000/alumnos/')[2]
    const response = await fetch(url);[3]
    if (!response.ok) {
        error(`Response status: ${response.status}`);
    }[4]

    let alumnos = await response.json();[5]

    return {
        alumnos: alumnos
    };[6]
}
```

1. Definimos y exportamos una función y le agregamos el denominador `async` para poder usar fetch.
2. Armamos la URL
3. Esperamos a que se ejecute el fetch
4. Chequeamos que no haya habido un error
5. Esperamos a poder mostrar el contenido del body de la response
6. Devolvemos un objeto con los alumnos (podríamos hacer más cosas y devolver un objeto más completo si quisiéramos).

Forma de manejar errores.
Hay que agregar el siguiente import:

```
import { error } from '@sveltejs/kit';
```

Svelte: +page.server

Debemos ahora mostrar la información en nuestro componente

`+page.svelte`

```
<table>
    <thead>
        <tr>
            <th>Padron</th>
            <th>Nombre</th>
            <th>Apellido</th>
            <th>Edad</th>
        </tr>
    </thead>
    <tbody>
        {#each data.alumnos as alumno}
            <tr>
                <td>{alumno.padron}</td>
                <td>{alumno.nombre}</td>
                <td>{alumno.apellido}</td>
                <td>{alumno.edad}</td>
            </tr>
        {/each}
    </tbody>
</table>
```

Pero nos falta algo, debemos decirle a svelte dónde cargar la información que estamos devolviendo de nuestro `+page.server.js`
Debemos agregarle al componente lo siguiente:

```
<script>
    let { data } = $props();
</script>
```

El each es propio de Svelte, no es de JS

Si llegamos a tener un problema acá, chequear la diapositiva de CORS.

Forms

Todos conocemos lo que es un formulario en un sitio: llenamos los datos que queremos completar y al "enviarlo" ocurre algo.

Los Forms son la forma que se definió que se iba a interactuar con usuarios cuando se les iba a pedir información para mandar al servidor.

Casi todos los frameworks modernos tienen soporte nativo para ello.

Agreguemos un form a nuestro sitio.

Para ello primero agreguemos una vista para grupos (igual a la de alumnos, que los liste) y luego, debajo, un form para crear uno nuevo.

Forms

```
<form method="POST" action="?/create">
    <label>
        Nombre:
        <input
            name="nombre"
            autocomplete="off"
        />
    </label>
    <button>Crear</button>
</form>
```

Al crear un grupo, por ahora, pasémosle sólo el nombre, por lo tanto el form tendrá un sólo campo.

La acción que elijamos reflejará qué se ejecutará al completar el form. Debemos crear esta acción en el archivo `+page.server.js` de esta ruta.

Forms

```
export const actions = {
    create: async ({ cookies, request }) => {
        const data = await request.formData();[1]

        let url = new URL('http://localhost:8000/grupos/')
        const response = await fetch(url, {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },[2]
            body: JSON.stringify({nombre: data.get("nombre")})[3]
        });
        if (!response.ok) {
            error(`Response status: ${response.status}`);
        }
    },
}
```

1. Primero obtenemos los datos que fueron cargados en el form.
2. Al armar la URL, debemos poner el tipo del contenido que enviamos, porque envíamos un body.
3. Debemos armar el body según la especificación del BE. Ojo con cómo se está obteniendo la información de la `data`.

Al terminar de ejecutarse la página se recargará automáticamente y cargará del BE la lista de grupos con el nuevo grupo.

Forms: enhanced

Podemos agregarle lógica a nuestros forms para actuar en base al éxito o fracaso de un submit

use:enhance nos permite definirle comportamiento según el resultado de la operación

Recibe dos funciones:

- la primera es para el caso exitoso
- la segunda para el caso no exitoso

En base a esto podemos hacer lo que queramos en la página para reflejar el resultado. Una de las operaciones más comunes es la de mostrar un banner o mensaje indicando el resultado de la operación.

Para usarlo debemos importarlo

```
import { enhance } from '$app/forms';
```

Forms: enhanced

Cuando creemos un grupo, queremos ver un banner de que fue creado exitosamente

Modificamos el form para agregarle el

`use:enhance`

```
<form
   class="grupo-form"
   method="POST"
   action="?/create"
   use:enhance={() => {
       mostrarBannerGrupoCreado();
   }}
>
```

Agregamos la función en la sección de script

```
function mostrarBannerGrupoCreado() {
   exito = true;
   setTimeout(() => {
       exito = false;
   }, 3000);
}
```

Y agregamos una sección en el HTML para mostrar el banner

```
{#if exito}
 <div class="banner">✅ Grupo creado con éxito</div>
{/if}
```

Ruta con identificadores dinámicos

Por ahora estuvimos mostrando el listado de alumnos, grupos, pero también nos gustaría tener una página para ver uno solo de estos recursos y quizás ver más detalles.

Propuesta: en la vista de grupos, al hacer click en cada uno debería llevarme a una página donde pueda ver el detalle del mismo y poder agregar o quitar alumnos al mismo.

Creemos una ruta de la siguiente manera:

```
/grupos
    /[id]
        +page.server.js
        +page.svelte
```

Y en cada grupo de la tabla, agreguemos un link a esta página:

```
<a href="/grupos/{grupo.id}">
```

En el archivo de servidor, cargamos el detalle del grupo con la función `load({params})` que en este caso va recibir en sus params el ID del grupo en cuestión.

Finalmente en el componente, mostramos la información.

Rutas con identificador dinámico

Definamos el componente

```
<script>
    import '../../table-styles.css';

    let { data } = $props();
</script>

<h1>{data.grupo.nombre}</h1>
<h2>Integrantes</h2>
<table>
    <tbody>
        {#each data.grupo.integrantes as integrante}

        {/each}
    </tbody>
</table>
```

```
<tr>
    <td>
        {integrante.alumno.padron}
    </td>
    <td>
        <span>{integrante.alumno.nombre} {integrante.alumno.apellido}</span>
    </td>
    <td>
        {integrante.nota}
    </td>
</tr>
```

Rutas con identificador dinámico

Y tambien el page server

```
import { error } from '@sveltejs/kit';

export async function load({ params }) {
    let url = new URL(`http://localhost:8000/grupos/${params.id}`)
    const response = await fetch(url);
    if (!response.ok) {
        error(response.status)
    }

    let grupo = await response.json();

    return {
        grupo
    };
}
```

Pero me gustaría poder agregar alumnos también, y ponerles nota!

Acciones por elemento

En nuestro listado de alumnos, podemos agregar una acción para que el usuario pueda remover un alumno de un grupo.

Debemos seguir los pasos que ya hicimos antes:

- Cada fila de la tabla tendrá un form con la información a mandar al BE.
- El `page.server` debe tener la lógica en una acción custom para hacer el request.

Cambiemos el componente para que cada fila tenga lo siguiente:

```
<form method="POST" action="?/desinscribir">
    <input type="hidden" name="padron" value={integrante.alumno.padron} />
    <input type="hidden" name="id" value={data.grupo.id} />
    <button>Remover</button>
</form>
```

El atributo `type="hidden"` hace que no se muestre al usuario el campo.
Necesitamos pasar tanto el id del grupo y el padrón del alumno para hacer la request.

Acciones por elemento

```
export const actions = {
   desinscribir: async ({ request }) => {
       const data = await request.formData();

       let url = new URL(`http://localhost:8000/grupos/${data.get('id')}/integrantes/${data.get('padron')}`)

       const response = await fetch(url, {
           method: 'DELETE',
           headers: { 'Content-Type': 'application/json' }
       });
       if (!response.ok) {
           error(response.status, 'Algo falló');
       }
   },
}
```

Inscribir alumno

Nos falta una forma de poder agregar alumnos a un grupo

Agregamos el form en el componente

```
<h2>Inscribir Alumno</h2>
<form method="POST" action="?/inscribir">
    <input type="hidden" name="id" value={data.grupo.id} />
    <label for="padron">Padrón:</label>
    <input type="number" id="padron" name="padron" min="1" required />
    <button type="submit">Inscribir</button>
</form>
```

Agregamos el action en el page server

```
inscribir: async ({ request }) => {
    const data = await request.formData();
    let url = new URL(
`http://localhost:8000/grupos/${data.get('id')}/integrantes`)
    const response = await fetch(url, {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({padron: data.get("padron")})
    });
    if (!response.ok) {
        error(response.status, response);
    }
},
```

Inscribir alumno

Necesito acordarme el padrón de cada alumno para saber cuál agregar, no es muy cómodo.

Podemos implementar un buscador que me permita elegir el alumno de una lista, de un desplegable.

Pero sigue sin ser muy cómodo si tengo muchos elementos.

Podemos implementar el buscador de manera que nos permita tipear el nombre o parte del nombre de lo que estoy buscando y me muestre una lista reducida a medida que voy escribiendo (como Google).

Esto se llama *typeahead*

Typeahead

Comúnmente se denomina a las barras de búsqueda que nos permiten buscar y filtrar elementos al mismo tiempo *typeahead*. Porque a medida que uno tipea, va buscando según lo que se ingresó.

Podemos hacer un componente que funcione de esta manera nosotros mismos, pero también podemos usar alguno que ya esté hecho.

Nos interesa que lo que hagamos se comporte de esta manera:

- a medida que tipeo haga un request al BE filtrando por nombre (y/o apellido) del alumno y me vaya mostrando el listado reducido
- al seleccionar un alumno utilice luego la información del mismo (el padrón puntualmente) para el form que hicimos recién y poder inscribir al alumno al grupo

Podemos buscar en el directorio de `npm` a ver qué encontramos: https://www.npmjs.com/

Typeahead propio

Si no encontramos nada, lo podemos hacer nosotros relativamente fácil.

Hacemos por separado un input:

```
<input
    class="typeahead-input"
    type="text"
    bind:value={query}
    oninput={onInput}
    onblur={handleBlur}
    placeholder="Buscar por nombre..."
    autocomplete="off"
/>
```

Y luego el "dropdown" que me va a mostrar las opciones:

```
{#if showDropdown}
   <div class="dropdown">
       {#each results as result}
           <div class="dropdown-item" role="button" tabindex="0"
onmousedown={() => selectResult(result)}>
               ({result.padron}) {result.nombre}, {result.apellido}
           </div>
       {/each}
   </div>
{/if}
```

Typeahead propio

Luego en la sección de <script> agregamos la lógica para que funcione como un typeahead

```
let query = $state('');
let padron = $state(0)
let results = $state([]);
let showDropdown = $state(false);
let timeout;

function onInput() {
    clearTimeout(timeout);
    if (query.length < 2) {
        results = [];
        showDropdown = false;
        return;
    }
    timeout = setTimeout(fetchResults, 300);
}

function selectResult(result) {
    query = '';
    padron = result.padron;
    showDropdown = false;
}
```

Necesitamos actualizar el padrón elegido

```
async function fetchResults() {
    let url = new URL(`http://localhost:8000/alumnos/`)
    let params = { nombre: query }
    url.search = new URLSearchParams(params).toString();

    const response = await fetch(url, {
        method: 'GET',
        headers: { 'Content-Type': 'application/json' },
    });
    if (!response.ok) {
        error(response.status, response);
    }

    results = await response.json()
    showDropdown = results.length > 0;
}

function handleBlur() {
    setTimeout(() => showDropdown = false, 150);
}

onDestroy(() => clearTimeout(timeout));
```

Typeahead propio

Me falta una cosa más: mandar el padrón elegido en el dropdown en el form

Ahora el input del padrón pasa a ser hidden

```
<input hidden=true type="number" id="padron" name="padron" min="1" bind:value={padron} />
```

Y el valor del mismo lo "bindeamos" a la variable padrón.

Cada vez que se actualice la variable porque se hizo una selección en el dropdown, se actualizará la información a mandar en el form.

Typeahead como reusable component

Nuestro typeahead lo podemos reutilizar

Podemos mover nuestro typeahead a un componente reutilizable y hacer genéricos ciertos parámetros del mismo de manera que quien usa el componente los defina: el endpoint a usar, el texto placeholder, tiempo de espera, y la acción a tomar cuando se elige un resultado del dropdown.

El componente puede tener parámetros con un valor por default si quisiéramos.

Podemos ver el mismo Typeahead que hicimos recién como componente reutilizable en el repositorio de sveltekit-example en la branch alumnos-grupos.

Forms con recursos anidados

Nos hace falta una cosa más para tener la aplicación completa: poder crear un grupo ya indicándole los integrantes y la nota de cada uno.

Necesitamos un form, como cualquier otro, pero el mismo va a ser "dinámico".
Además de los campos del Grupo, vamos tener que mostrar una lista variable de Integrantes y sus datos y hacer que todo eso se mande al backend.
Necesitamos:

- El form
- Un botón o algo similar para agregar más integrantes para contabilizar cuántos quiero agregar
- Una sección en el form por cada integrante que se quiere agregar con sus campos y un botón para eliminar dicho integrante si al final no lo quiero agregar

Forms con recursos anidados

El form ya lo tenemos, falta el resto.

Primero, agreguemos un botón para agregar integrantes:

```
<button type="button" onclick={agregarIntegrante}>
    Agregar Integrante
</button>
```

La función en sí misma, y el estado donde se van a ir guardando

```
let grupo = $state({
    nombre: '',
    integrantes: [],
})

function agregarIntegrante() {
    grupo.integrantes.push({ nombre: '', apellido: '', edad: '' });
}
```

Al form de grupos agreguemos una sección para mostrar los integrantes:

```
<div>
    <h3>Integrantes:</h3>
    {#each grupo.integrantes as integrante, index}
        <fieldset>
            ...
        </fieldset>
    {/each}
</div>
```

A medida que agreguemos más integrantes, se actualizará el ciclo y se renderizarán más campos, para cada uno.

Forms con recursos anidados

Vamos al contenido de cada fieldset

Primero el Typeahead que podemos reutilizar

```
<legend>Alumno {index + 1}</legend>
<Typeahead
    endpoint="http://localhost:8000/alumnos/"
    placeholder="Elegir alumno..."
    on:select={(event) => {
        grupo.integrantes[index] = event.detail.result
    }}
/>
```

Esta función actualiza el integrante correspondiente al indice actual con la información que se devuelve el Typeahead.

Luego puedo poner campos para mostrar la información del alumno elegido, como por ejemplo:

```
<div>
    <label for="student-name-{index}">Nombre:</label>
    <input
        id="student-name-{index}"
        type="text"
        bind:value={grupo.integrantes[index].nombre}
        required
        readonly
    />
</div>
```

Al tener el binding se actualiza solo cuando se elija un resultado del Typeahead

Y repetir lo mismo para apellido y padrón.

Forms con recursos anidados

Vamos al contenido de cada fieldset

Finalmente, un input para la nota

```
<div>
    <label for="student-nota-{index}">Nota:</label>
    <input
        id="student-nota-{index}"
        type="number"
        bind:value={grupo.integrantes[index].nota}
        min=1
        max=10
        required
    />
</div>
```

Y un botón en cada Integrante para eliminarlo si no lo quiero

```
<button type="button" onclick={() => quitarIntegrante(index)}>
    Quitar Alumno
</button>
```

Junto con su función

```
function quitarIntegrante(index) {
    grupo.integrantes.splice(index, 1);
}
```

Forms con recursos anidados

Por último, debemos enviar la data al page.server y al backend.

En el form, debemos tener un campo para la información que queremos enviar. Podemos hacer lo siguiente para guardar todos los integrantes:

```
<input type="hidden" name="integrantes"
value={JSON.stringify(grupo.integrantes)} />
```

Por último, en el page.server debemos agregar a la request la información de los integrantes:

```
create: async ({ request }) => {
    ...
    const integrantes = JSON.parse(data.get('integrantes') || '[]');
    const payload = { nombre, integrantes };
    ...
}
```

Pagination

Deberíamos paginar los resultados en nuestra tabla de Alumnos ya que el BE lo permite

Vamos a hacer un reusable component ya que la misma lógica de paginación la podríamos reusar en distintas tablas.

También vamos a aprovechar la lógica que tenemos hoy en día en el page_server en la función de load() para hacer la request al BE (aunque podríamos no reusarla).

Me gustaría tener arriba de la tabla dos botones que me permitan ir a la página anterior o a la página siguiente (Previous y Next por ejemplo).

La idea de implementación es la misma que con el Typeahead: vamos a bindear una variable que va a tener el número de página a estos botones. La diferencia va a estar en que cuando se modifique vamos a cargar la misma función de load() que ya veníamos usando antes (la URL va a pasar a tener un query param con el nro de página).

Pagination

Deberíamos paginar los resultados en nuestra tabla de Alumnos ya que el BE lo permite

El componente Pagination.svelte

```
<div class="pagination">
    {#if currentPage > 0}
        <button onclick={prevPage} class="pagination-btn">
            ← Anterior
        </button>
    {/if}

    <span class="page-info">
        Página {currentPage + 1}
    </span>

    {#if hasMore}
        <button onclick={nextPage} class="pagination-btn">
            Siguiente →
        </button>
    {/if}
</div>
```

```
<script>
    import { goto } from '$app/navigation';
    import { page } from '$app/stores';

    let { currentPage = 0, pageSize = 20, hasMore = true } = $props();

    function goToPage(newPage) {
        const url = new URL($page.url);
        url.searchParams.set('page', newPage.toString());
        goto(url.toString());
    }

    function nextPage() {
        goToPage(currentPage + 1);
    }

    function prevPage() {
        goToPage(currentPage - 1);
    }
</script>
```

Pagination

Deberíamos paginar los resultados en nuestra tabla de Alumnos ya que el BE lo permite

El componente Pagination.svelte

```
<div class="pagination">
    {#if page > 0}
        <button onclick={prevPage} class="pagination-btn">
            ← Anterior
        </button>
    {/if}

    <span class="page-info">
        Página {page + 1}
    </span>

    {#if hasMore}
        <button onclick={nextPage} class="pagination-btn">
            Siguiente →
        </button>
    {/if}
</div>
```

```
<script>
    import { goto } from '$app/navigation';
    import { page } from '$app/stores';

    let { page = 0, pageSize = 20, hasMore = true } = $props();

    function goToPage(newPage) {
        const url = new URL($page.url);
        url.searchParams.set('page', newPage.toString());
        goto(url.toString());
    }

    function nextPage() {
        goToPage(currentPage + 1);
    }

    function prevPage() {
        goToPage(currentPage - 1);
    }
</script>
```

Pagination

Deberíamos paginar los resultados en nuestra tabla de Alumnos ya que el BE lo permite

Actualizamos el componente de la página de alumnos,

agregamos el import

```
import Pagination from '$lib/components/Pagination.svelte';
```

y renderizamos el componente

```
<Pagination
   currentPage={data.currentPage}
   pageSize={data.pageSize}
   hasMore={data.hasMore}
/>
```

y actualizamos también en page_server

```
export async function load({ url }) {
    const page = parseInt(url.searchParams.get('page') || '0');
    const pageSize = 20;
    const offset = page * pageSize;

    let apiUrl = new URL(`${PUBLIC_API_URL}/alumnos/`)
    apiUrl.searchParams.set('limit', pageSize.toString());
    apiUrl.searchParams.set('offset', offset.toString());
    const response = await fetch(apiUrl);
    if (!response.ok) {
        error(`Response status: ${response.status}`);
    }
    const alumnos = await response.json();

    const hasMore = alumnos.length === pageSize;
    return {
        alumnos: alumnos,
        page: page,
        pageSize: pageSize,
        hasMore: hasMore
    };
}
```

Ejercicio

Dada esta API <TODO>

Escribir una aplicación FE que haga lo siguiente:

- Tenga una página para mostrar la lista de Todos
- Tenga una página para mostrar un Todo en detalle (haciendo click en uno de la lista) donde se muestren sus datos y su lista de asignados ("`Nombre Apellido (Email)`")
- En el detalle del Todo, tenga un form que permita asignar un usuario al mismo
- En la lista de asignados permita desasignar un usuario del Todo

Bonus Tracks

Debugging

Debuggear el código FE no es tán sencillo como debuggear la contraparte en el BE.

Tenemos muchos tipos de "código" y cada uno de va a debuggear de distintas maneras.

`console.log` AKA `print`

`console.log()` es una función de JS que permite mostrar por consola algo. Lo que sería el print en otros lenguajes.
Si lo usamos en los componentes `.svelte` se va a imprimir el contenido en la consola del navegador
Si lo usamos en los archivos `+page.server.js` se va a imprimir el contenido en la consola de la terminal

En los componentes .svelte podemos aprovechar las propiedades reactivas del framework e imprimir cada vez que una variable reactiva cambie de valor:

```
let name = $state('');
$effect(() => {
    console.log('Changed to:', name);
});
```

Debugging

Debuggear el código FE no es tán sencillo como debuggear la contraparte en el BE.

Tenemos muchos tipos de "código" y cada uno de va a debuggear de distintas maneras.

Web Browser devtools

Cada browser tiene su propio debugger que podemos usar para analizar el código de los componente Svelte. Debemos ir a la parte de "Sources" buscar el componente de Svelte que queremos debuggear y poner breakpoints allí mismo. Sólo el código que esté dentro de los tags de `<script>` va a poder ser debuggeado.

Existe también una extensión de Chrome/Firefox llamada Svelte Devtools que amplía las capacidades de debugging en el navegador mismo.

Debugging

Debuggear el código FE no es tán sencillo como debuggear la contraparte en el BE.

Tenemos muchos tipos de "código" y cada uno de va a debuggear de distintas maneras.

IDE debugger

VS Code tiene su propio debugger para el código js, que podemos usar para el código que está en los archivos de `+page.server.js`
En `Run and Debug` -> `Node.js` -> `Run script: dev`
Luego simplemente debemos poner los breakpoints y listo

CORS

Cross Origin Resource Sharing

Es un mecanismo de HTTP para poder limitar el origen de las request a un servidor.
Debemos especificar en nuestro servidor BE de dónde permitimos que entren requests.

```
origins = [
    "*",
]

app.add_middleware(
    CORSMiddleware,
    allow_origins=origins,
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

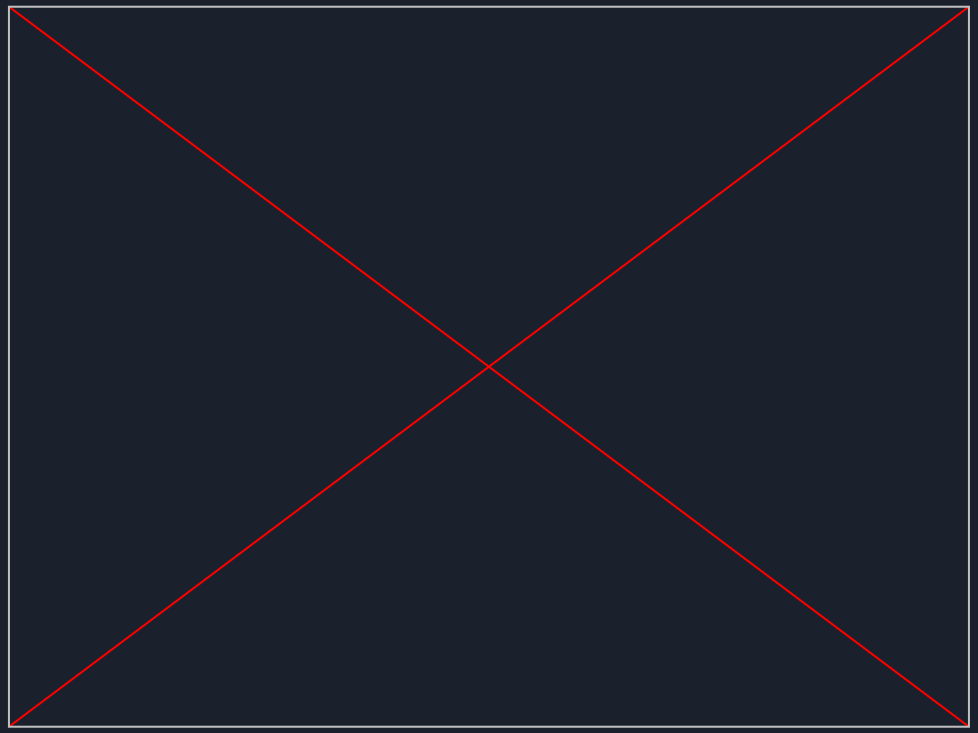
Y agregarle los siguientes headers a la request hecha por el cliente:

```
headers: {
    'Access-Control-Allow-Origin': '*',
},
```

DOM

Node modules

wat



Un video de DestroyAllSoftware sobre cosas "divertidas" de Javascript.
Saltear al minuto **1:25** para ver específicamente sobre este lenguaje.

Fin