The project

The code is stored in: https://github.com/alberto-rizzo-dev/employees-schedules.git

The project is about a site that manage of employees's work shifts.
The site include 3 main pages:

- Main page: it contains the shifts's table
- A page for adding new employees
- A page for adding new shifts

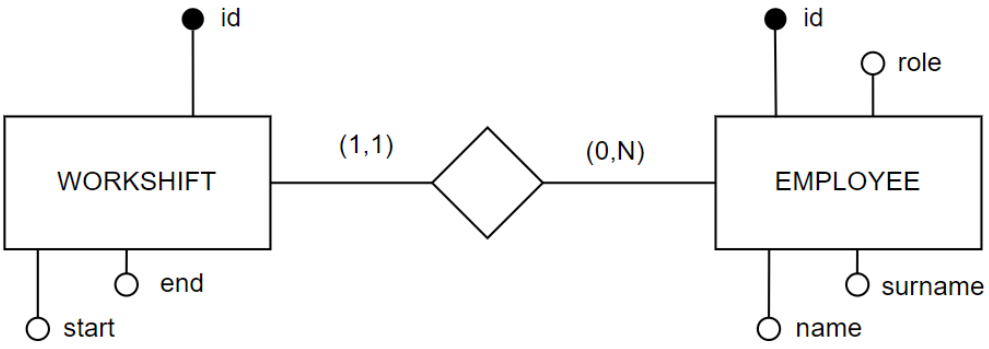
The project was done with next js, using typescript and tailwind css for the styling.
The data is stored in a postgres database.

The code inside the app folder is divided in that structure:
- [id] / edit : contain the code for the page for editing shifts
- add-employee/ : contain the code for the page for adding new employees
- insert-workshifts/ : contain the code for the page for adding new shifts
- ui/ : contains react components used for the project such as tables, forms and buttons
- lib/ : contains utilities files such as connection to the database or types definitions

(in the main folder there is a .env file. It includes the keys for connecting to my sql database)

Database

The database is hosted using vercel and made in postgress.
The database schema is shown below.



Employee:
- id: serial PRIMARY KEY
- name: varchar(20) NOT NULL
- surname: varchar(20) NOT NULL
- role: varchar(20) NOT NULL

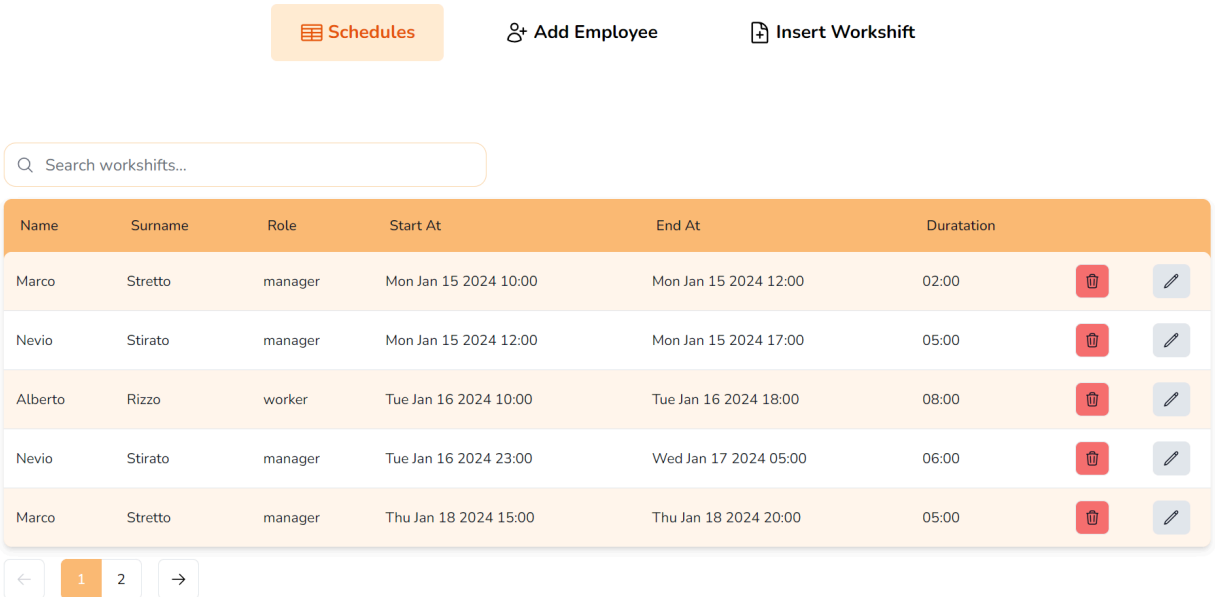
Workshift:

- id: serial PRIMARY KEY
- start: timestamp NULL
- end: timestamp NOT NULL
- employee: int REFERENCES employee(id)

Maybe it would have been better if the role attribute had been a table.
As this is a simple project i preferred manage the list of roles by the front end application instead of doing that in the database.

```
role: 'worker' |'manager' | 'office worker' | 'manager';
```

The main page



In the main page you can:

- Search shifts shown in the table
- Delete or update shifts

Search:
Url search params are used to look for shifts. This params is set by the search component, using ‘useSearchParams’ hook and, then used to update the query that the table uses to fetch the work shifts.

Down below is the code of Search component, the search function is called when the input textbox is updated (using debounce for not queerying the server each keyboard input).

```tsx
'use client';
import { MagnifyingGlassIcon } from '@heroicons/react/24/outline';
import { useSearchParams, usePathname, useRouter } from 'next/navigation';
import { useDebouncedCallback } from 'use-debounce'; //for not queerying server

export default function Search({ placeholder }: { placeholder: string }) {
  const searchParams = useSearchParams();
  const pathname = usePathname();
  const { replace } = useRouter();

  const search = useDebouncedCallback((parameter) => {
    const params = new URLSearchParams(searchParams);
    params.set('page', '1');
    if (parameter) {
        params.set('query', parameter);
      } else {
        params.delete('query');
      }
    replace(`${pathname}?${params.toString()}`);
  }, 300);
```

Then in the code of the page (page.tsx) the created query is passed to the Table component.

```tsx
export default async function Page({
    searchParams,
  }: {
    searchParams?: {
      query?: string;
      page?: string;
    };
  }) {
    const query = searchParams?.query || '';
    const page = Number(searchParams?.page) || 1;
    const totalPages = await fetchTablePages(query);

    return (
        <div className="md:p-10 p-2 flex-column justify-center items-center">
            <div className="pb-3"><Search placeholder="Search workshifts..." /><
            <Suspense key={query + page} fallback={<TableSkeleton/>}>
                <EmployeesTable query={query} currentPage={page}/>
            </Suspense>
```

Finally in the table component the query is used to fetch data from the server.

```tsx
const rows = await fetchTableData(query,currentPage);
```

(currentPage is used for the pagination of the table)

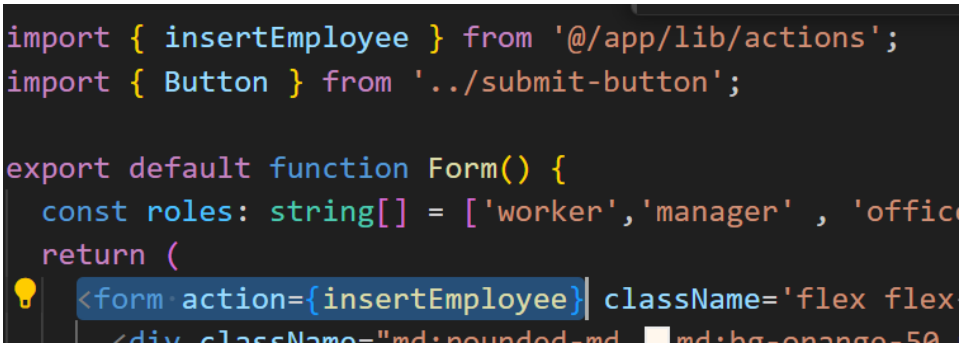
Fetching data

The code of the query that fetches data is in lib\db_connection.tsx, using vercel SDK.
Vercel SDK also provides protection against SQL injections.

For fetching and mutating data the project uses Server actions.
React Server Actions allow you to run asynchronous code directly on the server.
They eliminate the need to create API endpoints to mutate your data. Instead, you write asynchronous functions that execute on the server and can be invoked from your Client or Server Components.
Server Actions offer an effective security solution, protecting against different types of attacks, securing your data, and ensuring authorized access.

Mutating data

In the pages for inserting new employees or shifts forms are used to pass data to the actions file (lib\actions, the data is captured and validated using Zod).
Below there is an example of adding a new employee.



The actions file uses Server Actions and sends queries for muting data to the postgres database.

Here is an example.

```
export async function insertEmployee(formData: FormData) {
    const { name, surname, role } = FormSchemaEmployee.parse({
      name: formData.get('name'),
      surname: formData.get('surname'),
      role: formData.get('role'),
    });
    if(!(/^[A-Za-z\s]*$/.test(name)) || !(/^[A-Za-z\s]*$/.test(surname)))
        throw new Error('Name and surname must contain only letters.');
    try{
        await sql`
        INSERT INTO employee (name, surname, role)
        VALUES (${name},${surname},${role})
        `;

    }catch(err){
        throw new Error('Failed to insert employee.');
    }
    revalidatePath('/');
    redirect('/');
}
```

(revalidatePath is used for refreshing the table in the main page, redirect to go back to it)

Any error that occurs in the actions code is throwed and managed by the file error.tsx in the app folder (it returns a visual alert to the user with the message of the error).

Thanks for reading.