Chinook Frontend

HowTos / Tutorials

Version wip, 2025-03-30

Table of Contents

Simple CRUD Employees

We will walk you through creating simple crud functionality using the employees table. We assume that the database is running already (`docker-compose up -d`).

Creating the table model

Whenever you create new tables in the database, they must be made known to Kysely, so it can provide type-safe queries. Because the `employees` table is present already and in the `tables.ts` model, you can skip this step.

You generate the model using

```
yarn kysely:generate
```

This will create a file `tables.ts` in the `kysely-codegen` directory. We do this, so you can review the generated code and make changes if needed. Then move / copy / merge it to the final destination.

```
mv kysely-codegen/tables.ts app/shared/infrastructure/db/model/kysely
```

Creating derived types

The derived tables are normally not directly used. You create derived types. We do it in the `app/shared/infrastructure/db/model/kysely/entities.ts` file.

```
import { Employee as EmployeeTable } from './tables';

export type EmployeeEntity = Selectable<EmployeeTable>;
export type NewEmployeeEntity = Insertable<EmployeeTable>;
export type UpdatableEmployeeEntity = Updateable<EmployeeTable>;
```

The `Selectable`, `Insertable` and `Updateable` types are used to create the correct types for the different operations.

Create a new `/employees` list page

This requires the definition of a new route. Routes are defined in `app/routes.tsx`.

```
layout("routes/layouts/page-with-header.tsx", [
    route("/employees", "routes/employees/layout.tsx", [
            index("routes/employees/index.tsx"),
        ]),
])
```

This creates a nested layout `layout.tsx` and in that an index route and a route for creating new employees.

Create the layout page

A layout page is used to create a common layout for the different pages. In this case it should simple display a heading "Employees" and provide a page to render nested routes.

app/routes/employees/layout.tsx

```
import {Outlet} from 'react-router';

export default function EmployeesLayout() {
    return (
        <>
            <h1>Employees</h1>
            <Outlet />
        </>
    )
}
```

Create the index page

The index page is the default page for the `/employees` route.

A route module consists of functions for data loading and manipulation and a default export for the component.

The data loading function is called `loader` and is used to load data for the page. And we will use kysely to load all customers.

So lets have a look to the loader first. .app/routes/employees/index.tsx (loader)

```
import type {Route} from "./+types/index";
import {kyselyBuilder} from '~/shared/infrastructure/db/db.server';

export async function loader({request}: Route.LoaderArgs) {
    const db = kyselyBuilder()
    const employees = await db.selectFrom('employee').selectAll().execute();
    return {employees};
}
```

We first get a kysely instance and then select the employees from the database. As you will notice your IDE / Editor should provide you with type-safe access to the database. If you specify, e.g. selectFrom('employees') it will provide you with a hint. Code Completion will normally propose table and column names that are possilbe at the different places. The only thing, you will notice is, that selectAll or select('firstName') comes last, to be type-safe. Otherwise it is very similar to SQL.

And last but not least, we return an object containing one property "employees" containing the list of

employees.

What you will also notice is that we import the Route type from the './+types/index'. That is something that is generated by the vite react-router plugin. It create the necessary type information.

It can be manually called using

```
yarn react-router typegen
```

If you are looking for the generated file, it is located at `frontend/.react-router/types/app/routes/employees/+types/index.ts`

Create the component

The default export will then use the

```
export default function EmployeeList({loaderData}: Route.ComponentProps) {
    const {employees} = loaderData;
    return (
        <div>
            <ul>
                {employees.map((employee) => (
                    <li key={employee.employeeId}>
                        {employee.firstName} {employee.lastName}
                    </li>
                ))}
            </ul>
        </div>
    );
}
```

Also important is to add authentication / authorization to the route. Nested routes are loaded in parallel. Authentication from a parent route does not apply to child routes automatically.

However, this is quite easy:

```
import {authenticate} from '~/shared/services/auth.server';

export async function loader({request}: Route.LoaderArgs) {
    await authenticate(request, "/employees");
 ...
}
```

If you navigate to `http://localhost:5173/employees` you should see a list of employees.

We will create a possibility to create new employees.

Add feature "Create Employee"

Add a new route

We update the block in `app/routes.tsx` to add a new route for creating employees.

```
 route("/employees", "routes/employees/layout.tsx", [
            index("routes/employees/index.tsx"),
            route("/employees/new", "routes/employees/new.tsx"),
        ]),
```

Add a new route module

And add a new file `app/routes/employees/new.tsx` with the following content:

```
import {type Employee, employee} from "~/modules/employees/employee.model";
import type {Route} from "./+types/new";
import {zodResolver} from '@hookform/resolvers/zod';
import {useRemixForm} from 'remix-hook-form';
import {Form} from "react-router";
import {Button, Stack, TextInput} from "@mantine/core";

const resolver = zodResolver(employee);

export default function EmployeeNew({actionData}: Route.ComponentProps) {
    const {register, handleSubmit, formState: {errors}} = useRemixForm<Employee>
({resolver})
    return (
        <>
            <Form onSubmit={handleSubmit} method="post">
                <Stack gap={"md"}>
                    <TextInput label={"First Name"} {...register("firstName")} error=
{errors.firstName?.message}/>
                    <TextInput label={"Last Name"} {...register("lastName")} error=
{errors.lastName?.message}/>
                    <Button type="submit">Create</Button>
                </Stack>
            </Form>
        </>
    )
}
```

You see, we need to create a schema and a type "employee / Employee" for the employee.

We do this in `app/modules/employees/employee.model.ts`:

```
import { z } from "zod";
```

```
export const employee = z.object({
    firstName: z.string().min(2).max(50),
    lastName: z.string().min(2).max(50),
})

export type Employee = z.infer<typeof employee>;
```

The zod library is used to create a schema for the employee. This schema is use for validation. As we will see, on the client and on the server.

Now you can navigate to http://localhost:5173/employees/new. But creation does not work.

You get the error:

```
Error
Method Not Allowed
```

This is because, we have not defined an action for the route. The action is used to handle the non-GET requests.

Create the action

```
import {getValidatedFormData, useRemixForm} from 'remix-hook-form';
import {data, Form} from "react-router";
import {kyselyBuilder} from '~/shared/infrastructure/db/db.server';

export async function action({request}: Route.ActionArgs) {
    const {data: formData, errors, receivedValues: defaultValues} = await
getValidatedFormData(request, resolver);
    if (errors) {
        return data({errors, defaultValues}, {status: 400});
    }
    const db = kyselyBuilder()
    await db.insertInto('employee').values(formData).execute();
    return redirect('/employees');
}
```

Notice: We are reusing the same resolver as in the component. The validation has only been implemented once. We do a simple form submit and no explicit REST call from the client.

Now you can navigate to http://localhost:5173/employees/new and create a new employee. You should see the new employee in the list of employees.

Conclusion

You have seen, how to access the database using kysely and validate data using zod on the client and

on the server. In real world project the routes should only be thin delegates to further services and components. You should minimize logic and UI in the routes and push downwards to specialized modules. This keeps the application more maintainable and leads to a separation of concerns and less duplicated code. We will see how to do it in another example.