Actividad 6:Extender Express.js con TypeScript

En esta actividad, vas a extender un servidor Express.js utilizando TypeScript para añadir anotaciones de tipo y crear una nueva ruta utilizando tipos personalizados. Esta es una oportunidad para aplicar los conocimientos adquiridos en TypeScript y mejorar la robustez y legibilidad del código de un servidor Node.js.

Objetivo:

- Reescribir el servidor Express.js que creaste previamente en actividades anteriores. Añadirás anotaciones de tipo a las rutas existentes y crearás una nueva ruta para mostrar detalles meteorológicos, utilizando tipos personalizados e interfaces.

Parte 1: Configuración del proyecto con TypeScript

1. Instalación de TypeScript y definiciones de tipos de Express:

- Abre una terminal en el directorio de tu proyecto Express.js.
- Ejecuta los siguientes comandos para instalar TypeScript y las definiciones de tipo para Express.js:

  npm install --save-dev typescript @types/express

2. Archivo package.json:

- Verifica que tu archivo package.json ahora incluya TypeScript y las definiciones de tipo de Express.js en las devDependencies:

```
{
  "name": "sample-express",
  "version": "1.0.0",
  "description": "sample express server",
  "license": "ISC",
  "type": "module",
  "dependencies": {
    "express": "^4.18.2",
    "node-fetch": "^3.2.6"
  },
```


```
    "devDependencies": {
      "@types/express": "^4.17.15",
      "typescript": "^4.9.4"
    }
  }
```

3. Archivo tsconfig.json:

- Crea un archivo tsconfig.json en el directorio raíz de tu proyecto y agrega la siguiente configuración para que TypeScript transpile tu código correctamente:

```
{
    "compilerOptions": {
    "esModuleInterop": true,
     "module": "es6",
    "moduleResolution": "node",
    "target": "es6",
    "noImplicitAny": true
    }
}
```

Parte 2: Creación de tipos personalizados

1. Definir tipos personalizados:

- Crea un archivo llamado custom.d.ts en el directorio raíz de tu proyecto. Añade los siguientes tipos personalizados e interfaces:

```
type responseItemType = {
    id: string;
    name: string;
};

type WeatherDetailType = {
```


```
        zipcode: string;
        weather: string;
        temp?: number;
    };

    interface WeatherQueryInterface {
    zipcode: string;
}
```

Estos tipos se utilizarán en la nueva ruta que agregarás en la siguiente parte.

Parte 3: Reescribir las rutas en TypeScript

1. Actualizar el archivo routes.ts:

- Cambia el nombre de tu archivo routes.js a routes.ts y ajusta el código para añadir anotaciones de tipo:

```
import fetch from "node-fetch";
const routeHello = (): string => "Hello World!";

const routeAPINames = async (): Promise<string> => {
        const url = "https://www.usemodernfullstack.dev/api/v1/users";
        let data: responseItemType[];
            try {
                const response = await fetch(url);
                data = (await response.json()) as responseItemType[];
        } catch (err) {
            return "Error";
    }
    const names = data
        .map((item) => id: ${item.id}, name: ${item.name})
```


```
        .join("<br>");
        return names;
    };

    const routeWeather = (query: WeatherQueryInterface): WeatherDetailType =>
        queryWeatherData(query);

    const queryWeatherData = (query: WeatherQueryInterface): WeatherDetailType => {
    return {
        zipcode: query.zipcode,
        weather: "sunny",
        temp: 35
        };
    };
    export { routeHello, routeAPINames, routeWeather };
```

2. Añadir anotaciones de tipo en index.ts:

- Cambia el nombre de index.js a index.ts y ajusta el código para incluir las anotaciones de tipo y la nueva ruta meteorológica:

```
    import { routeHello, routeAPINames, routeWeather } from "./routes";
    import express, { Request, Response } from "express";

    const server = express();
    const port = 3000;

    server.get("/hello", function (_req: Request, res: Response): void {
    const response = routeHello();
    res.send(response);
});
```


```
server.get("/api/names", async function (_req: Request, res: Response): Promise<void> {
  try {
    const response = await routeAPINames();
    res.send(response);
  } catch (err) {
    console.log(err);
  }
});

server.get("/api/weather/:zipcode", function (req: Request, res: Response): void {
  const response = routeWeather({ zipcode: req.params.zipcode });
  res.send(response);
});

server.listen(port, function (): void {
  console.log("Listening on " + port);
});
```

Parte 4: Transpilar y ejecutar el servidor

1. Transpilar el código TypeScript a JavaScript:

- Ejecuta el siguiente comando en la terminal para transpilar los archivos TypeScript:

  npx tsc

2. Iniciar el servidor:

- Después de la transpilación, inicia el servidor con el siguiente comando:

  node index.js

3. Probar el servidor:

- Abre tu navegador y navega a http://localhost:3000/api/weather/12345. Deberías ver una respuesta con los detalles meteorológicos.

Ejercicios adicionales:

1. Extender el servidor:

- Añade una nueva ruta /api/cities que devuelva un arreglo de objetos con detalles de ciudades (nombre y población). Define tipos personalizados para los datos que se devolverán.

Tip: Puedes crear un tipo CityType que contenga name y population.

2. Manejo de errores con tipos personalizados:

- Modifica la función routeAPINames para devolver un mensaje de error detallado cuando falle la petición. Crea un tipo personalizado ErrorResponse que tenga una propiedad message.

3. Añadir validación de datos:

- Añade validación de datos en la ruta /api/weather/:zipcode para asegurarte de que el código postal proporcionado tiene 5 dígitos. Si no es válido, devuelve un error con un tipo personalizado ValidationError.

4. Ampliar los tipos de respuesta:

- Añade un tipo personalizado para la respuesta que incluya una propiedad status y el resultado real de la ruta. Deberías devolver algo como { status: "success", data: ... }.

5. Uso de enum en lugar de tipos literales:

- Crea una nueva ruta /api/orders que devuelva detalles de pedidos. Define un tipo personalizado OrderType que incluya un estado del pedido como 'pending' | 'shipped' | 'delivered'. Luego, refactoriza el código para usar un **enum** en lugar de tipos literales para manejar los diferentes estados del pedido.

  Tip: Un enum puede definirse así:

```
enum OrderStatus {
        Pending = "pending",
        Shipped = "shipped",
        Delivered = "delivered"
        }
```

6. Refactorización con tipos genéricos:

- Refactoriza la función routeWeather para que acepte diferentes tipos de datos de entrada. Usa **tipos genéricos** para que la función pueda recibir cualquier tipo de parámetro y

procesar distintos formatos de datos. Luego, agrega otra función que use el mismo tipo genérico para procesar otra clase de datos.

```
function processData<T>(data: T): T {
        return data;
}
```