c Typing the express app

Now that we have a basic understanding of how TypeScript works and how to create small projects with it, it's time to start creating something actually useful. We are now going to create a new project with a bit more realistic use cases in mind.

One major change from the previous part is that *we're not going to use ts-node anymore*. It is a handy tool and helps you get started, but in the long run it is advicable to use the official TypeScript compiler that comes with the *typescript* npm-package. The official compiler generates and packages JavaScript files from the .ts files so that the built *production version* won't contain any TypeScript code anymore. This is the exact outcome we are aiming for, since TypeScript itself is not executable by browsers or Node.

Setting up the project

We will create a project for Ilari, who loves flying small planes but has a difficult time managing his flight history. He is a coder himself, so he doesn't necessarily need a user interface, but he'd like to use a software with HTTP-requests and retain the possibility of later adding a web-based user interface to the application.

Let's start by creating our first real project: `Ilari's flight diaries`. As usual run *npm init* and install the *typescript* package as a dev dependency.

```
npm install typescript --save-dev
```

TypeScript's Native Compiler (*tsc*) can help us to initialize our project, generating our *tsconfig.json* file. First, we need to add the *tsc* command to the list of executable scripts in *package.json* (unless you have installed `typescript` globally). Even if you installed TypeScript globally, you should always add it as a dev-dependency to your project.

The npm script for running *tsc* is set as follows:

```
{
  // ..
  "scripts": {
    "tsc": "tsc"
  },
  // ..
}
```

The bare *tsc* command is often added to the scripts so that other scripts can use it, hence don't be surprised to find it set up within the project like this.

We can now initialise our tsconfig.json settings by running:

```
npm run tsc -- --init
```

Note the extra -- before the actual argument! Arguments before -- are interpreted as being for the *npm* command, while the ones after that are meant for the command that is run through the script (i.e. *tsc* in this case).

The *tsconfig.json* file we just created contains a lengthy list of every configuration available to us. However, most of them are commented out. Studying this file can help you finding some configuration options you might need. It is also completely okay to keep the commented lines, in case you might need them someday.

At the moment, we want the following to be active:

```
{
  "compilerOptions": {
    "target": "ES6",
    "outDir": "./build/",
    "module": "commonjs",
    "strict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "esModuleInterop": true
  }
}
```

Let's go through each configuration:

The *target* configuration tells the compiler which `ECMAScript` version to use when generating JavaScript. ES6 is supported by most browsers, so it is a good and safe option.

outDir tells where the compiled code should be placed.

module tells the compiler that we want to use `CommonJS` modules in the compiled code. This means we can use the old *require* syntax instead of the *import* one, which is not supported in older versions of `Node`, such as the version 10.

strict is actually a shorthand for multiple separate options: *noImplicitAny, noImplicitThis, alwaysStrict, strictBindCallApply, strictNullChecks, strictFunctionTypes and strictPropertyInitialization*. They guide our coding style to use the TypeScript features more strictly. For us perhaps the most important is the already familiar noImplicitAny. It prevents implicitly setting type *any*, which can for example happen if you don't type the parameters of a function. Details about the rest of the configurations can be found in the tsconfig documentation. Using *strict* is suggested by the official documentation.

noUnusedLocals prevents having unused local variables, and *noUnusedParameters* throws an error if a function has unused parameters.

noFallthroughCasesInSwitch ensures that, in a `switch case`, each case ends either with a *return* or a *break* statement.

esModuleInterop allows interoperability between CommonJS and ES Modules; see more in the documentation.

Now that we have set our configuration, we can continue by installing *express* and, of course, also *@types/express*. Also, since this is a real project, which is intended to be grown over time, we will use eslint from the very beginning:

```
npm install express
npm install --save-dev eslint @types/express @typescript-eslint/eslint-plugin @typescript-eslint/parse
```

Now our *package.json* should look like this:

```
{
  "name": "flight_diary",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "tsc": "tsc"
  },
  "author": "Jane Doe",
  "license": "ISC",
  "devDependencies": {
    "@types/express": "^4.17.11",
    "@typescript-eslint/eslint-plugin": "^4.16.1",
    "@typescript-eslint/parser": "^4.16.1",
    "eslint": "^7.21.0",
    "typescript": "^4.2.2"
  },
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

We also create an *.eslintrc* file with the following content:

```
{
  "extends": [
    "eslint:recommended",
    "plugin:@typescript-eslint/recommended",
    "plugin:@typescript-eslint/recommended-requiring-type-checking"
  ],
  "plugins": ["@typescript-eslint"],
  "env": {
    "browser": true,
    "es6": true,
    "node": true
  },
  "rules": {
    "@typescript-eslint/semi": ["error"],
    "@typescript-eslint/explicit-function-return-type": "off",
    "@typescript-eslint/explicit-module-boundary-types": "off",
    "@typescript-eslint/restrict-template-expressions": "off",
    "@typescript-eslint/restrict-plus-operands": "off",
    "@typescript-eslint/no-unsafe-member-access": "off",
    "@typescript-eslint/no-unused-vars": [
      "error",
      { "argsIgnorePattern": "^_" }
    ],
    "no-case-declarations": "off"
  },
  "parser": "@typescript-eslint/parser",
  "parserOptions": {
    "project": "./tsconfig.json"
  }
}
```

Now we just need to set up our development environment, and we are ready to start writing some serious code. There are many different options for this. One option could be to use the familiar *nodemon* with ts-node. However, as we saw earlier, ts-node-dev does the exact same thing, so we will use that instead. So, let's install *ts-node-dev*

```
npm install --save-dev ts-node-dev
```

We finally define a few more npm script, and voilà, we are ready to begin:

```
{
  // ...
  "scripts": {
    "tsc": "tsc",
    "dev": "ts-node-dev index.ts",
    "lint": "eslint --ext .ts ."
  },
  // ...
}
```

As you can notice, there is a lot of stuff to go through before beginning actual coding. When you are working with a real project, careful preparations support your development process. Take the needed time to create a good setup for yourself and your team, so that everything runs smoothly in the long run.

Let there be code

Now we can finally start coding! As always, we start by creating a ping-endpoint, just to make sure everything is working.

The contents of the *index.ts* file:

```
import express from 'express';
const app = express();
app.use(express.json());

const PORT = 3000;

app.get('/ping', (_req, res) => {
  console.log('someone pinged here');
  res.send('pong');
});

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

Now if we run the app with *npm run dev* we can verify that a request to http://localhost:3000/ping gives a response *pong*, so our configuration is set!

When starting the app with *npm run dev*, it runs in development mode. The development mode is not suitable at all when we later operate the app in production.

Let's try to create a *production build* by running the TypeScript compiler. Since we have defined the *outdir* in our tsconfig.json, there's really nothing else to do but run the script *npm run tsc*.

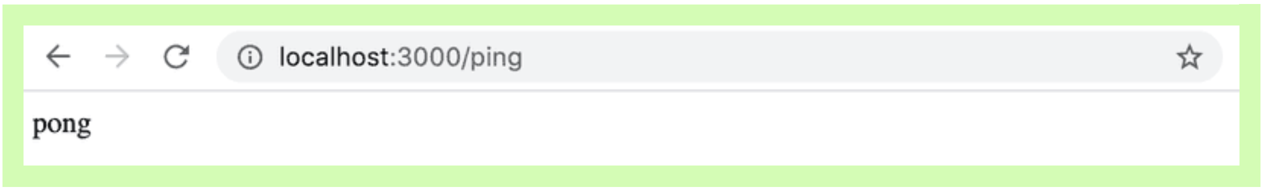
Just like magic a native runnable JavaScript production build of the express backend is created into the directory *build*.

Currently, if we run eslint it will also interpret the files in the *build* directory. We don't want that, since the code there is compiler generated. We can prevent this by creating a *.eslintignore* file which lists the content we want eslint to ignore, just like we do with git and *gitignore*.

Let's add an npm script for running the application in production mode:

```
{
  // ...
  "scripts": {
    "tsc": "tsc",
    "dev": "ts-node-dev index.ts",
    "lint": "eslint --ext .ts .",
    "start": "node build/index.js"
  },
  // ...
}
```

When we run app with *npm start* we can verify that also the production build works



pong

Now we have a minimal working pipeline for developing our project. With the help of our compiler and eslint it also ensures that a good code quality is maintained. With this base we can actually start creating an app that we could later on deploy into a production environment.

Exercises 9.8.-9.9.

Before you start the exercises

For this set of exercises you will be developing a backend for an existing project called Patientor, which is a simple medical record application for doctors who handle diagnoses and basic health information of their patients.

The frontend has already been built by outsider experts and your task is to create a backend to support the existing code.

9.8: Patientor backend, step1

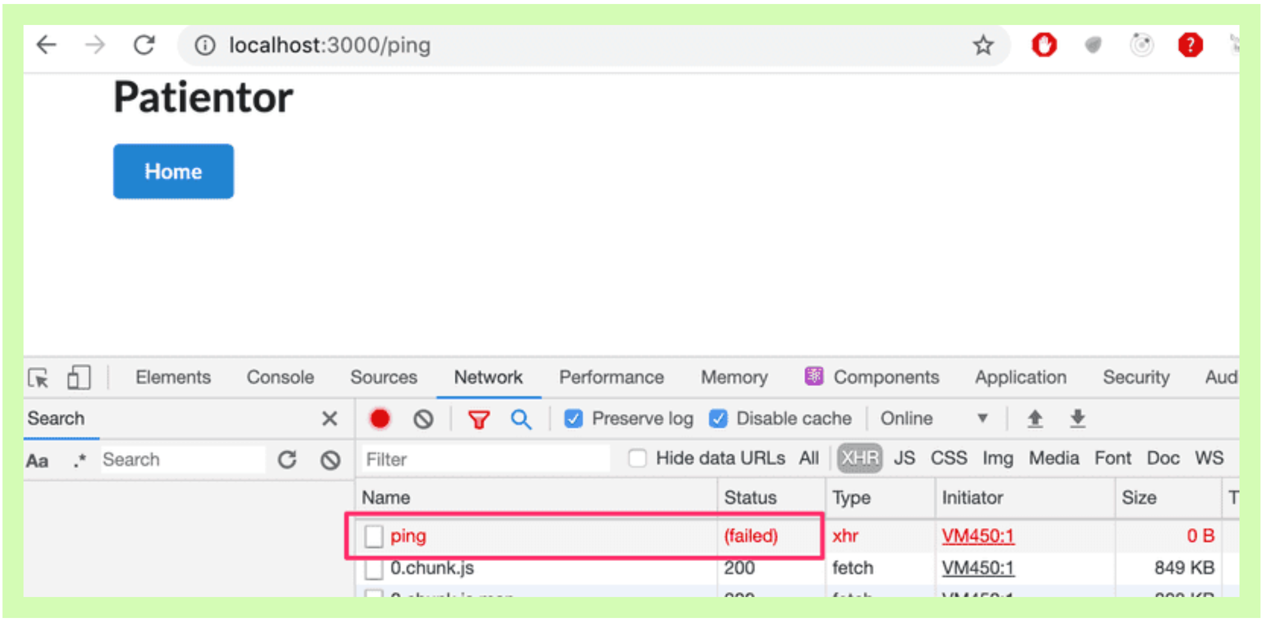
Initialise a new backend project that will work with the frontend. Configure eslint and tsconfig with the same configurations as proposed in the material. Define an endpoint that answers to HTTP GET requests to route */ping*.

The project should be runnable with npm scripts, both in development mode and, as compiled code, in production mode.

9.9: Patientor backend, step2

Fork and clone the project patientor. Start the project with the help of the README file. You should be able to use the frontend without a functioning backend.

Ensure that backend answers to the ping request that *frontend* has made on startup. Check developer tool to make sure it really works:



Name	Status	Type	Initiator	Size
ping	(failed)	xhr	VM450:1	0 B
0.chunk.js	200	fetch	VM450:1	849 KB

You might also want to have a look at the tab *console*. If something fails part 3 of the course shows how the problem can be solved.

Implementing the functionality

Finally we are ready to start writing some code.

Let's start from the basics. Ilari wants to be able to keep track of his experiences on his flight journeys.

He wants to be able to save diary entries which contain:

- The date of the entry
- Weather conditions (good, windy, rainy or stormy)
- Visibility (good, ok or poor)
- Free text detailing the experience

We have obtained some sample data, which we will use as a base to build on. The data is saved in json format, and can be found from here.

The data looks like the following

```
[
  {
    "id": 1,
    "date": "2017-01-01",
    "weather": "rainy",
    "visibility": "poor",
    "comment": "Pretty scary flight, I'm glad I'm alive"
  },
  {
    "id": 2,
    "date": "2017-04-01",
    "weather": "sunny",
    "visibility": "good",
    "comment": "Everything went better than expected, I'm learning much"
  },
  // ...
]
```

Let's start by creating an endpoint which returns all flight diary entries.

First we need to make some decisions on how to structure our source code. It is better to place all source code under *src* directory, so source code is not mixed with configuration files. We will move *index.ts* there and make the necessary changes to the npm scripts.

We will place all routers, modules which are responsible for handling a set of specific resources such as *diaries*, under the directory *src/routes*. This is a bit different than what we did in part 4, where we used directory *src/controllers*.

The router taking care of all diary endpoints is in *src/routes/diaries.ts* and looks like this:

```
import express from 'express';

const router = express.Router();

router.get('/', (_req, res) => {
  res.send('Fetching all diaries!');
});

router.post('/', (_req, res) => {
  res.send('Saving a diary!');
});

export default router;
```

We'll route all requests to prefix */api/diaries* to that specific router in `index.ts`

```
import express from 'express';
import diaryRouter from './routes/diaries';
const app = express();
app.use(express.json());

const PORT = 3000;

app.get('/ping', (_req, res) => {
  console.log('someone pinged here');
  res.send('pong');
});

app.use('/api/diaries', diaryRouter);


app.listen(PORT, () => {
    console.log(`Server running on port ${PORT}`);
});
```

And now if we make a HTTP GET request to http://localhost:3000/api/diaries we should see the message *Fetching all diaries!*.

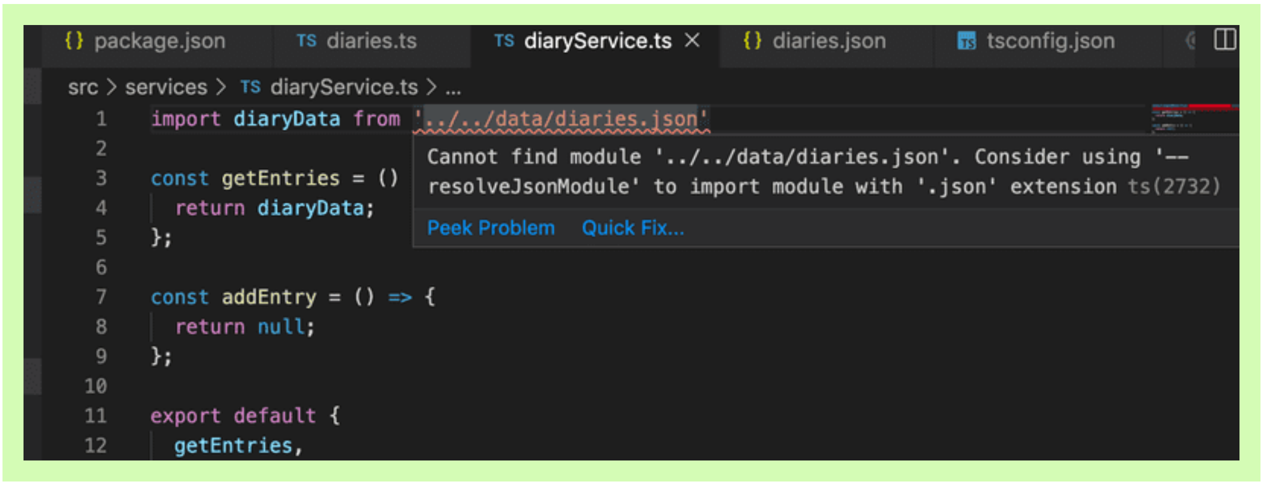
Next we need to start serving the seed data (found here) from the app. We will fetch the data and save it to *data/diaries.json*.

We won't be writing the code for the actual data manipulations on the router. We will create a *service* which takes care of the data manipulation instead. It is quite common practice to separate the "business logic" from the router code into its own modules, which are quite often called *services*. The name service originates from Domain driven design and was made popular by the Spring framework.

Let's create a *src/services* directory and place the *diaryService.ts* file in it. The file contains two functions for fetching and saving diary entries:

```
import diaryData from '../../data/diaries.json';

const getEntries = () => {
  return diaryData;
};

const addEntry = () => {
  return null;
};

export default {
  getEntries,
  addEntry
};
```

But something is not right



```
import diaryData from '../../data/diaries.json'

const getEntries = ()
  return diaryData;
};

const addEntry = () => {
  return null;
};

export default {
  getEntries,
```

```
Cannot find module '../../data/diaries.json'. Consider using '--resolveJsonModule' to import module with '.json' extension ts(2732)
```

The hint says we might want to use *resolveJsonModule*. Let's add it to our tsconfig:

```
{
  "compilerOptions": {
    "target": "ES6",
    "outDir": "./build/",
    "module": "commonjs",
    "strict": true,
    "noUnusedLocals": true,
    "noUnusedParameters": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "esModuleInterop": true,
    "resolveJsonModule": true
  }
}
```

And our problem is solved.

> NB: For some reason VSCode tends to complain it cannot find the file *../../data/diaries.json* from the service despite the file existing. That is a bug in the editor, and goes away when the editor is restarted.

Earlier we saw how the compiler can decide the type of a variable by the value it is assigned. Similarly the compiler can interpret large data sets consisting of objects and arrays. Due to this the compiler can actually warn us if we try to do something suspicious with the json data we are handling. For example if we are handling an array containing objects of specific type, and we try to add an object which does not have all the fields the other objects have, or has type conflicts (for example a number where there should be a string), the compiler can give us a warning.

Even though the compiler is pretty good at making sure we don't do anything unwanted, it is safer to define the types for the data ourselves.

Currently we have a basic working TypeScript express app, but there are barely any actual *typings* in the code. Since we know what type of data should be accepted for the weather and visibility fields, there is no reason for us not to include their types to the code.

Let's create a file for our types, *types.ts*, where we'll define all our types for this project.

First, let's type the *Weather* and *Visibility* values using a union type of the allowed strings:

```
export type Weather = 'sunny' | 'rainy' | 'cloudy' | 'windy' | 'stormy';

export type Visibility = 'great' | 'good' | 'ok' | 'poor';
```

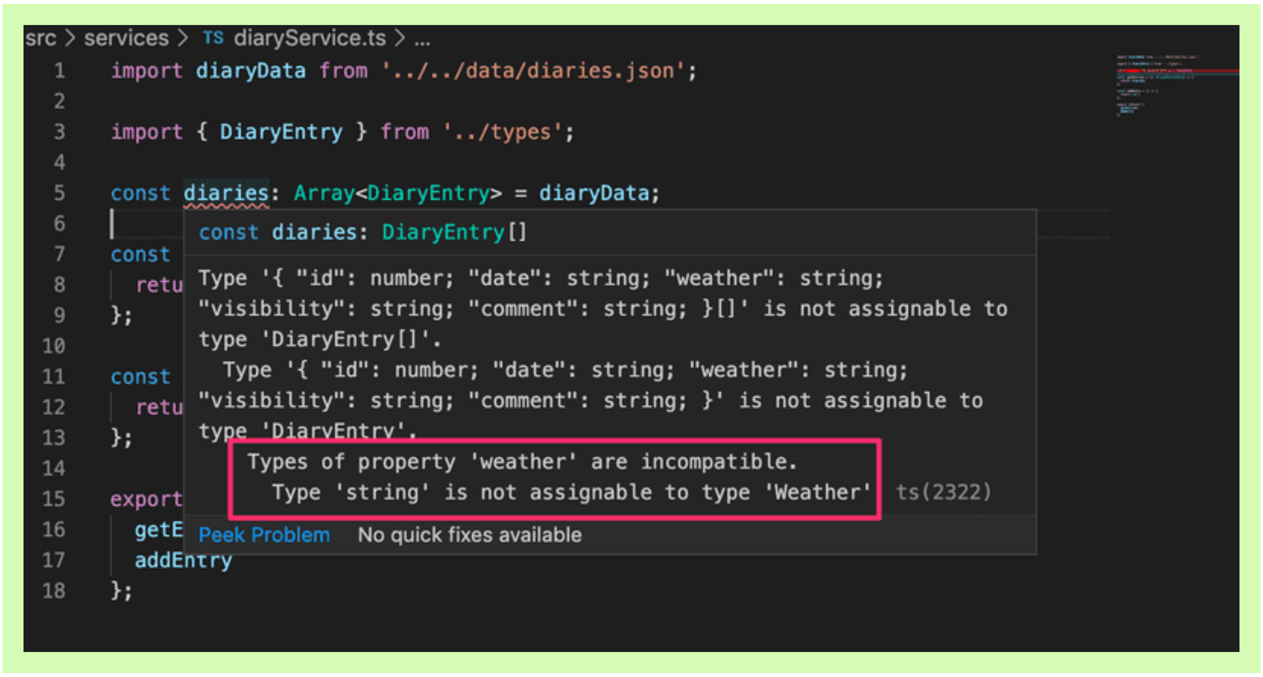
And from there we can continue by creating a DiaryEntry type, which will be an interface:

```
export interface DiaryEntry {
  id: number;
  date: string;
  weather: Weather;
  visibility: Visibility;
  comment: string;
}
```

We can now try to type our imported json:

```
import diaryData from '../../data/diaries.json';

import { DiaryEntry } from '../types';

const diaries: Array<DiaryEntry> = diaryData;

const getEntries = (): Array<DiaryEntry> => {
  return diaries;
};

const addEntry = () => {
  return null;
};

export default {
  getEntries,
  addEntry
};
```

But since the json already has its values declared, assigning a type for the data set results in an error:



```
src > services > TS diaryService.ts > ...
import diaryData from '../../data/diaries.json';

import { DiaryEntry } from '../types';

const diaries: Array<DiaryEntry> = diaryData;

const
  retu
};

const
  retu
};

export
  getE
  addEntry
};
```

```
const diaries: DiaryEntry[]
Type '{ "id": number; "date": string; "weather": string;
"visibility": string; "comment": string; }[]' is not assignable to
type 'DiaryEntry[]'.
  Type '{ "id": number; "date": string; "weather": string;
"visibility": string; "comment": string; }' is not assignable to
type 'DiaryEntry'.
    Types of property 'weather' are incompatible.
      Type 'string' is not assignable to type 'Weather'  ts(2322)
Peek Problem   No quick fixes available
```

The end of the error message reveals the problem: the *weather* fields are incompatible. In *DiaryEntry* we specified that its type is *Weather*, but TypeScript compiler had inferred its type to be *string*.

We can fix the problem by doing type assertion. This should be done only if we are certain we know what we are doing. If we assert the type of the variable *diaryData* to be *DiaryEntry* with the keyword *as*, everything should work:

```
import diaryData from '../../data/entries.json'

import { Weather, Visibility, DiaryEntry } from '../types'

const diaries: Array<DiaryEntry> = diaryData as Array<DiaryEntry>;

const getEntries = (): Array<DiaryEntry> => {
  return diaries;
}

const addEntry = () => {
  return null
}

export default {
  getEntries,
  addEntry
};
```

We should never use type assertion unless there is no other way to proceed, as there is always the danger we assert an unfit type to an object and cause a nasty runtime error. While the compiler trusts you to know what you are doing when using *as*, doing this we are not using the full power of TypeScript but relying on the coder to secure the code.

In our case we could change how we export our data so we can type it within the data file. Since we cannot use typings in a JSON-file, we should convert the json-file to a ts-file which exports the typed data like so:

```
import { DiaryEntry } from "../src/types";

const diaryEntries: Array<DiaryEntry> = [
  {
      "id": 1,
      "date": "2017-01-01",
      "weather": "rainy",
      "visibility": "poor",
      "comment": "Pretty scary flight, I'm glad I'm alive"
  },
  // ...
];

export default diaryEntries;
```

Now when we import the array, the compiler interprets it correctly and the *weather* and *visibility* fields are understood right:

```
import diaries from '../../data/diaries';

import { DiaryEntry } from '../types';

const getEntries = (): Array<DiaryEntry> => {
  return diaries;
}

const addEntry = () => {
  return null;
}

export default {
  getEntries,
  addEntry
};
```

Note, that if we want to be able to save entries without a certain field, e.g. *comment*, we could set the type of the field as optional by adding *?* to the type declaration:

```
export interface DiaryEntry {
  id: number;
  date: string;
  weather: Weather;
  visibility: Visibility;
  comment?: string;
}
```

Node and JSON modules

It is important to take note of a problem that may arise when using the tsconfig resolveJsonModule option:

```
{
  "compilerOptions": {
    // ...
    "resolveJsonModule": true
  }
}
```

According to the node documentation for file modules, node will try to resolve modules in order of extensions:

```
 ["js", "json", "node"]
```

In addition to that, by default, *ts-node* and *ts-node-dev* extend the list of possible node module extensions to:

```
 ["js", "json", "node", "ts", "tsx"]
```

> NB: The validity of *.js*, *.json* and *.node* files as modules in TypeScript depend on environment configuration, including *tsconfig* options such as *allowJs* and *resolveJsonModule*.

Consider a flat folder structure containing files:

```
├── myModule.json
└── myModule.ts
```

In TypeScript, with the *resolveJsonModule* option set to true, the file *myModule.json* becomes a valid node module. Now, imagine a scenario where we wish to take the file *myModule.ts* into use:

```
import myModule from "./myModule";
```

Looking closely at the order of node module extensions:

```
 ["js", "json", "node", "ts", "tsx"]
```

We notice that the *.json* file extension takes precedence over *.ts* and so *myModule.json* will be imported and not *myModule.ts*.

In order to avoid time eating bugs, it is recommended that within a flat directory, each file with a valid node module extension has a unique filename.

Utility Types

Sometimes we might want to use a specific modification of a type. For example, consider a page for listing some data, some of which is sensitive and some of which is non-sensitive. We might want to be sure that no sensitive data is used or displayed. We could *pick* the fields of a type we allow to be used to enforce this. We can do that by using the utility type Pick.

In our project we should consider that Ilari might want to create a listing of all his diary entries *excluding* the comment field, since during a very scary flight he might end up writing something he wouldn't necessarily want to show anyone else.

The Pick utility type allows us to choose which fields of an existing type we want to use. Pick can be used to either construct a completely new type, or to inform a function what it should return on runtime. Utility types are a special kinds of type tools, but they can be used just like regular types.

In our case, in order to create a "censored" version of the *DiaryEntry* for public displays, we can use Pick in the function declaration:

```
const getNonSensitiveEntries =
  (): Array<Pick<DiaryEntry, 'id' | 'date' | 'weather' | 'visibility'>> => {
    // ...
  }
```

and the compiler would expect the function to return an array of values of the modified DiaryEntry type, which include only the four selected fields.

Since Pick requires the type it modifies to be given as a type variable, just like Array does, we now have two nested type variables and the syntax is starting to look a bit odd. We can improve the code's readability by using the alternative array syntax:

```
const getNonSensitiveEntries =
  (): Pick<DiaryEntry, 'id' | 'date' | 'weather' | 'visibility'>[] => {
    // ...
  }
```

In this case we want to exclude only one field, so even better would be to use the Omit utility type, which we can use to declare which fields to exclude:

```
const getNonSensitiveEntries = (): Omit<DiaryEntry, 'comment'>[] => {
  // ...
}
```

Another way would be to declare a completely new type for the *NonSensitiveDiaryEntry*:

```
export type NonSensitiveDiaryEntry = Omit<DiaryEntry, 'comment'>;
```

The code now becomes

```
import diaries from '../../data/diaries';
import { NonSensitiveDiaryEntry, DiaryEntry } from '../types';

const getEntries = (): DiaryEntry[] => {
  return diaries;
};

const getNonSensitiveEntries = (): NonSensitiveDiaryEntry[] => {
  return diaries;
};

const addEntry = () => {
  return null;
};

export default {
  getEntries,
  addEntry,
  getNonSensitiveEntries
};
```

One thing in our application is a cause for concern. In the *getNonSensitiveEntries* we are returning the complete diary entries, and *no error is given* despite typing!

This happens because TypeScript only checks whether we have all of the required fields or not, but excess fields are not prohibited. In our case this means that it is *not prohibited* to return an object of type *DiaryEntry[]*, but if we were to try to access the *comment* field, it would not be possible because we would be accessing a field that TypeScript is unaware of even though it exists.

Unfortunately this can lead to unwanted behaviour if you are not aware of what you are doing; the situation is valid as long as TypeScript is concerned, but you are most likely allowing use that is not wanted. If we were now to return all of the diaryEntries from the *getNonSensitiveEntries* function to the *frontend*, we would actually be leaking the unwanted fields to the requesting browser even thought our types seem to imply otherwise!

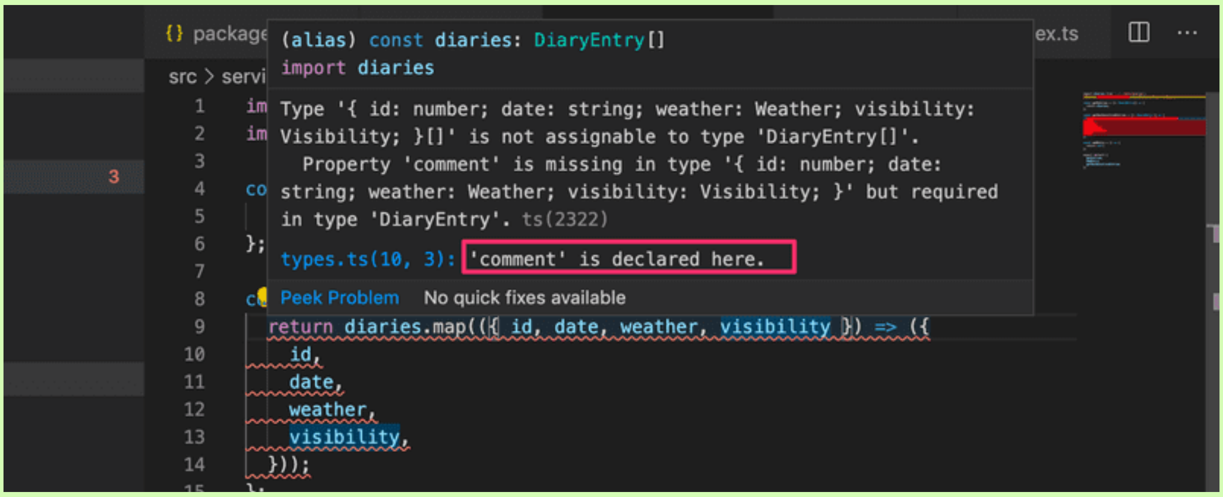
Because TypeScript doesn't modify the actual data but only its type, we need to exclude the fields ourselves:

```
import diaries from '../../data/entries.js'

import { NonSensitiveDiaryEntry, DiaryEntry } from '../types'

const getEntries = () : DiaryEntry[] => {
  return diaries
}

const getNonSensitiveEntries = (): NonSensitiveDiaryEntry [] => {
  return diaries.map(({ id, date, weather, visibility }) => ({
    id,
    date,
    weather,
    visibility,
  }));
};

const addDiary = () => {
  return []
}

export default {
  getEntries,
  getNonSensitiveEntries,
  addDiary
}
```

If we now would try to return this data with the basic *DiaryEntry* type, i.e. if we would type the function as follows

```
const getNonSensitiveEntries = () : DiaryEntry[] => {
```

we would get the following error:



```
(alias) const diaries: DiaryEntry[]
import diaries
Type '{ id: number; date: string; weather: Weather; visibility:
Visibility; }[]' is not assignable to type 'DiaryEntry[]'.
  Property 'comment' is missing in type '{ id: number; date:
string; weather: Weather; visibility: Visibility; }' but required
in type 'DiaryEntry'. ts(2322)
types.ts(10, 3): 'comment' is declared here.
Peek Problem   No quick fixes available
```

```
  return diaries.map(({ id, date, weather, visibility }) => ({
    id,
    date,
    weather,
    visibility,
  }));
```

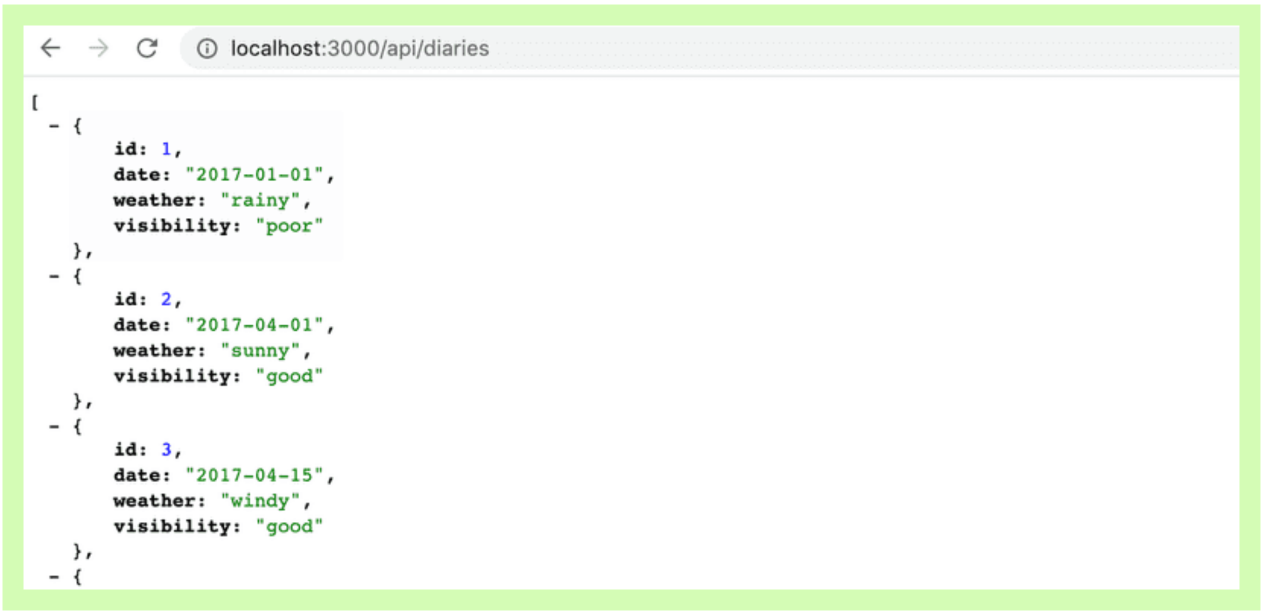
Again the last line of the error message is the most helpful one. Let's undo this undesired modification.

Utility types include many handy tools, and it is definitely worth it to take some time to study the documentation.

Finally, we can complete the route which returns all diary entries:

```
import express from 'express';
import diaryService from '../services/diaryService';

const router = express.Router();

router.get('/', (_req, res) => {
  res.send(diaryService.getNonSensitiveEntries());
});

router.post('/', (_req, res) => {
    res.send('Saving a diary!');
});

export default router;
```

The response is what we expect it to be



```
[
  - {
        id: 1,
        date: "2017-01-01",
        weather: "rainy",
        visibility: "poor"
    },
  - {
        id: 2,
        date: "2017-04-01",
        weather: "sunny",
        visibility: "good"
    },
  - {
        id: 3,
        date: "2017-04-15",
        weather: "windy",
        visibility: "good"
    },
  - {
```

Exercises 9.10.-9.11.

Similarly to Ilari's flight service, we do not use a real database in our app but instead use hardcoded data, that is in the files diagnoses.json and patients.json. Get the files and store those into a directory called *data* under your project. All data modification can be done in runtime memory, so during this part it is *not necessary to write to a file*.

9.10: Patientor backend, step3

Create a type *Diagnose* and use it to create endpoint */api/diagnoses* for fetching all diagnoses with HTTP GET.

Structure your code properly by using meaningfully named directories and files.

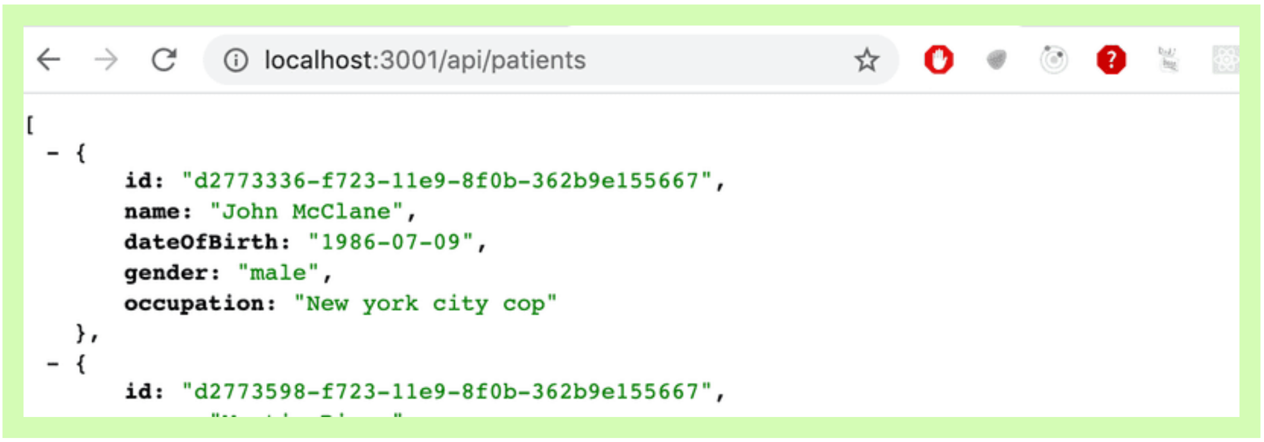
Note that *diagnoses* may or may not contain the field *latin*. You might want to use optional properties in the type definition.

9.11: Patientor backend, step4

Create data type *Patient* and set up a GET-endpoint */api/patients* that returns all patients to the frontend excluding field *ssn*. Use a utility type to make sure you are selecting and returning only the wanted fields.

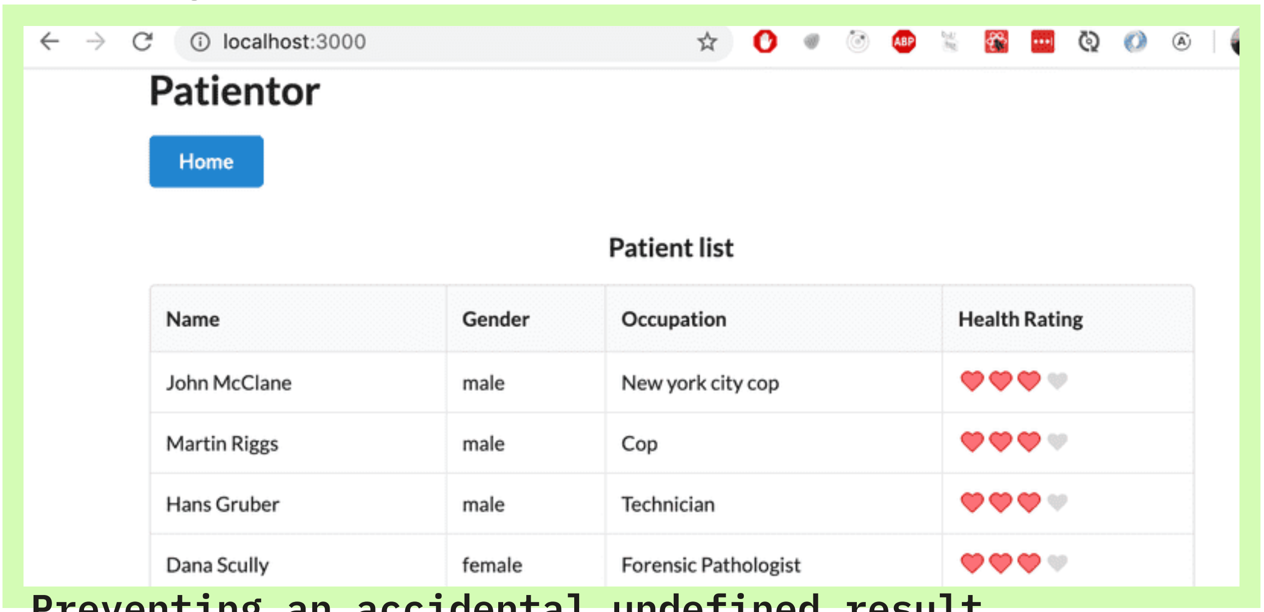
In this exercise you may assume that field *gender* has type *string*.

Try the endpoint with browser and ensure that *ssn* is not included in the response:



```
[
  - {
        id: "d2773336-f723-11e9-8f0b-362b9e155667",
        name: "John McClane",
        dateOfBirth: "1986-07-09",
        gender: "male",
        occupation: "New york city cop"
    },
  - {
        id: "d2773598-f723-11e9-8f0b-362b9e155667",
```

After creating the endpoint, ensure that the *frontend* shows the list of patients:



Name	Gender	Occupation	Health Rating
John McClane	male	New york city cop	❤❤❤♡
Martin Riggs	male	Cop	❤❤❤♡
Hans Gruber	male	Technician	❤❤❤♡
Dana Scully	female	Forensic Pathologist	❤❤❤♡

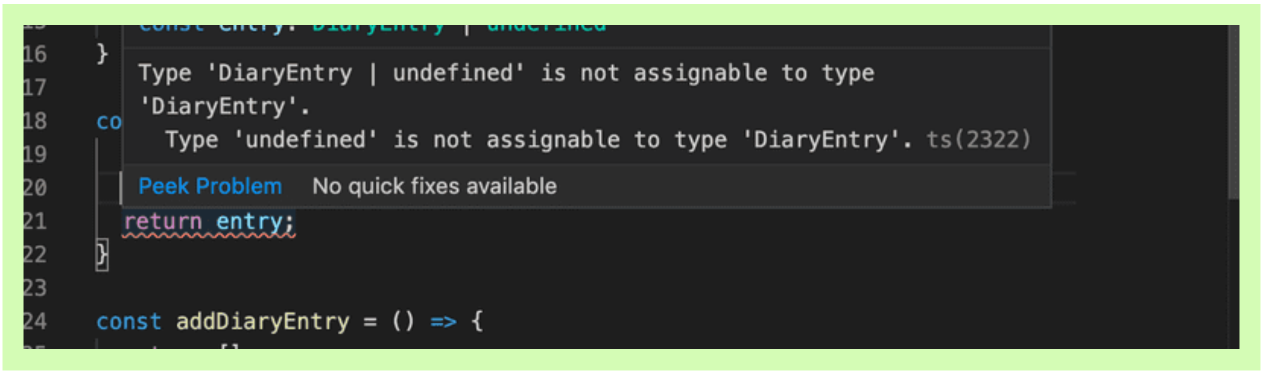
Preventing an accidental undefined result

Let's extend the backend to support fetching one specific entry with a HTTP GET request to route *api/diaries/:id*.

The DiaryService needs to be extended with *findById*-function:

```
// ...

const findById = (id: number): DiaryEntry => {
  const entry = diaries.find(d => d.id === id);
  return entry;
};

export default {
  getEntries,
  getNonSensitiveEntries,
  addDiary,
  findById
}
```

But once again, a new problem emerges:



```
Type 'DiaryEntry | undefined' is not assignable to type
'DiaryEntry'.
  Type 'undefined' is not assignable to type 'DiaryEntry'. ts(2322)
Peek Problem    No quick fixes available
```

```
  }

  co
  
    return entry;
}

const addDiaryEntry = () => {
```

The issue is, that there is no guarantee that an entry with the specified id can be found. It is good that we are made aware of this potential problem already at compile phase. Without TypeScript we would not be warned about this problem, and in the worst case scenario we could have ended up returning an *undefined* object instead of informing the user about the specified entry not being found.

First of all in cases like this we need to decide what the *return value* should be if an object is not found, and how the case should be handled. The *find* method of an array returns *undefined* if the object is not found, and this is actually fine with us. We can solve our problem by typing the return value as follows

```
const findById = (id: number): DiaryEntry | undefined => {
  const entry = diaries.find(d => d.id === id);
  return entry;
}
```

The route handler is the following

```
import express from 'express';
import diaryService from '../services/diaryService'

router.get('/:id', (req, res) => {
  const diary = diaryService.findById(Number(req.params.id));

  if (diary) {
    res.send(diary);
  } else {
    res.sendStatus(404);
  }
});

// ...

export default router;
```

Adding a new diary

Let's start building the HTTP POST endpoint for adding new flight diary entries. The new entries should have the same type as the existing data.

The code handling of the response looks as follows

```
router.post('/', (req, res) => {
  const { date, weather, visibility, comment } = req.body;
  const newDiaryEntry = diaryService.addDiary(
    date,
    weather,
    visibility,
    comment,
  );
  res.json(newDiaryEntry);
});
```

corresponding method in *diaryService* looks like this

```
import {
  NonSensitiveDiaryEntry,
  DiaryEntry,
  Visibility,
  Weather
} from '../types';


const addDiary = (
    date: string, weather: Weather, visibility: Visibility, comment: string
  ): DiaryEntry => {

  const newDiaryEntry = {
    id: Math.max(...diaries.map(d => d.id)) + 1,
    date,
    weather,
    visibility,
    comment,
  }

  diaries.push(newDiaryEntry);
  return newDiaryEntry;
};
```

As you can see, the *addDiary* function is becoming quite hard to read now that we have all the fields as separate parameters. It might be better to just send the data as an object to the function:

```
router.post('/', (req, res) => {
  const { date, weather, visibility, comment } = req.body;
  const newDiaryEntry = diaryService.addDiary({
    date,
    weather,
    visibility,
    comment,
  });
  res.json(newDiaryEntry);
})
```

But wait, what is the type of this object? It is not exactly a *DiaryEntry*, since it is still missing the *id* field. It could be useful to create a new type, *NewDiaryEntry*, for a not yet saved entry. Let's create that in *types.ts* using the existing *DiaryEntry* type and the Omit utility type:

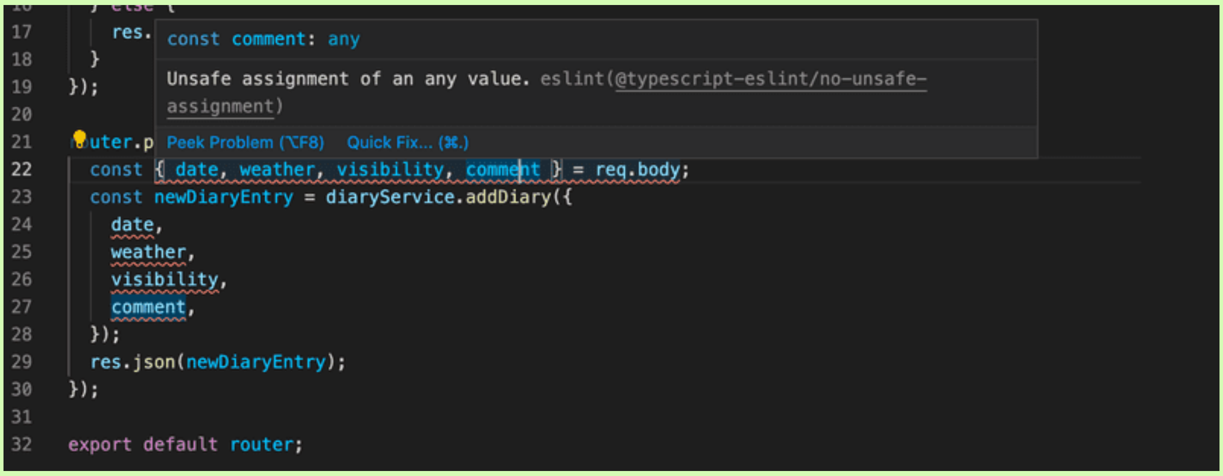
```
export type NewDiaryEntry = Omit<DiaryEntry, 'id'>;
```

Now we can use the new type in our DiaryService, and destructure the new entry object when creating an entry to be saved:

```
import { NewDiaryEntry, NonSensitiveDiaryEntry, DiaryEntry } from '../types';

// ...

const addDiary = ( entry: NewDiaryEntry ): DiaryEntry => {
  const newDiaryEntry = {
    id: Math.max(...diaries.map(d => d.id)) + 1,
    ...entry
  };

  diaries.push(newDiaryEntry);
  return newDiaryEntry;
};
```

Now the code looks much cleaner!

There is still a complain in our code:



```
      res.  const comment: any
    }       Unsafe assignment of an any value. eslint(@typescript-eslint/no-unsafe-
  });       assignment)

  router.p  Peek Problem (⌥F8)   Quick Fix... (⌘.)
    const { date, weather, visibility, comment } = req.body;
    const newDiaryEntry = diaryService.addDiary({
      date,
      weather,
      visibility,
      comment,
    });
    res.json(newDiaryEntry);
  });

  export default router;
```

The cause is eslint rule `@typescript-eslint/no-unsafe-assignment` that prevents us assigning the fields of request body to variables.

For the time being, let us just ignore the eslint-rule from the whole file by addin following as the first line of the file:

```
/* eslint-disable @typescript-eslint/no-unsafe-assignment */
```

In order to parse the incoming data we must have the *json* middleware configured:

```
import express from 'express';
import diaryRouter from './routes/diaries';
const app = express();
app.use(express.json());

const PORT = 3000;

app.use('/api/diaries', diaryRouter);

app.listen(PORT, () => {
  console.log(`Server running on port ${PORT}`);
});
```

Now the application is ready to receive HTTP POST requests for new diary entries of the correct type!

Proofing requests

There are plenty of things which can go wrong when we accept data from outside sources. Applications rarely work completely on their own, and we are forced to live with the fact that data from sources outside of our system cannot be fully trusted. When we receive data from an outside source, there is no way it can be already typed when we receive it. We need to make decisions on how to handle the uncertainty that comes with this.

The disabled eslint rule was actually giving us a hint the the following assignment is a risky one:

```
const diary = diaryService.findById(Number(req.params.id));
```

We certainly would like to have a certainity that the object in a post request is of a right type so let us define a function *toNewDiaryEntry* that receives the request body as a parameter and returns a properly typed *NewDiaryEntry* object. The function shall be defined in the file *utils.ts*.

The route definition uses the function as follows

```
import toNewDiaryEntry from '../utils';

// ...

router.post('/', (req, res) => {
  try {
    const newDiaryEntry = toNewDiaryEntry(req.body);

    const addedEntry = diaryService.addDiary(newDiaryEntry);
    res.json(addedEntry);
  } catch (e) {
    res.status(400).send(e.message);
  }
})
```

We can now also remove first line that ignores the eslint rule *no-unsafe-assignment*.

Since we are now making secure code and trying to ensure that we are getting exactly the data we want from the requests, we should get started with parsing and validating each field we are expecting to receive.

The skeleton of the function *toNewDiaryEntry* looks like the following:

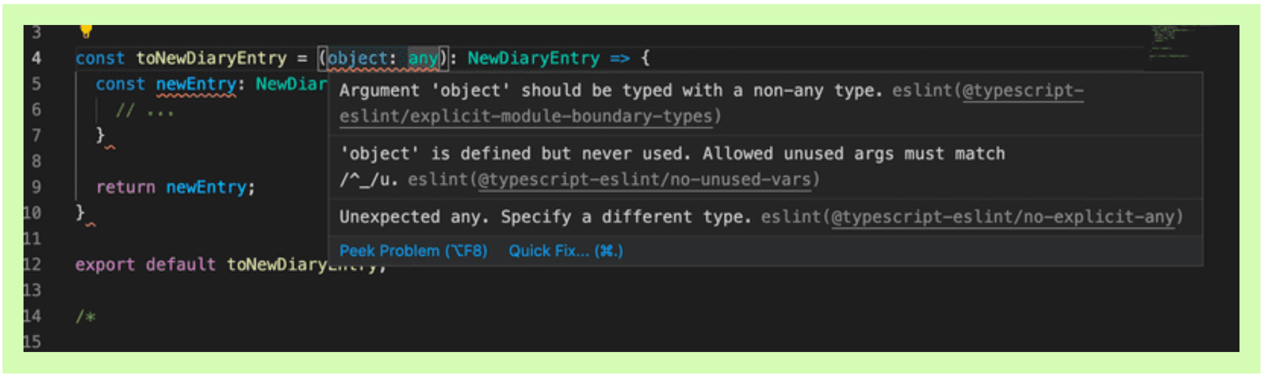
```
import { NewDiaryEntry } from './types';

const toNewDiaryEntry = (object): NewDiaryEntry => {
  const newEntry: NewDiaryEntry = {
    // ...
  }

  return newEntry;
}
```

```
export default toNewDiaryEntry;
```

The function should parse each field and make sure that the return value is exactly of type *NewDiaryEntry*. This means we should check each field separately.

Once again we have a type issue: what is the *object* type? Since the *object* is in fact the body of a request, Express has typed it as *any*. Since the idea of this function is to map fields of unknown type to fields of the correct type and check whether they are defined as expected, this might be the rare case where we actually *want to allow the any type*.

However if we type the object as *any*, eslint gives us two complaints:



```
const toNewDiaryEntry = (object: any): NewDiaryEntry => {
  const newEntry: NewDiar
    // ...
  }

  return newEntry;
}

export default toNewDiary

/*
```

```
Argument 'object' should be typed with a non-any type. eslint(@typescript-eslint/explicit-module-boundary-types)

'object' is defined but never used. Allowed unused args must match /^_/u. eslint(@typescript-eslint/no-unused-vars)

Unexpected any. Specify a different type. eslint(@typescript-eslint/no-explicit-any)
```

We could ignore these rules but a better idea is to follow the advice the editor gives when trying the *quick fix* and give the parameter type `unknown`:

```
import { NewDiaryEntry } from './types';

const toNewDiaryEntry = (object: unknown): NewDiaryEntry => {
  const newEntry: NewDiaryEntry = {
    // ...
  }

  return newEntry;
}

export default toNewDiaryEntry;
```

`unknown` is a new kind of top type that was introduced in TypeScript version 3 to be the type-safe counterpart of *any*. Anything is assignable to *unknown*, but *unknown* isn't assignable to anything but itself and *any* without a type assertion or a control flow based narrowing. Likewise, no operations are permitted on an *unknown* without first asserting or narrowing to a more specific type.

unknown is the ideal type for our kind of situation of input validation, since we don't yet need to define the type to match *any* type, but can first verify the type and then confirm the expected type. With the use of *unknown* we also don't need to worry about the *@typescript-eslint/no-explicit-any* eslint rule, since we are not using *any*. However, we might still need to use *any* in some cases where we are not yet sure about the type and need to access properties of an *any* object in order to validate or type check the property values themselves.

Let us start creating the parsers for each of the fields of *object*.

To validate the *comment* field we need to check that it exists, and to ensure that it is of the type *string*.

The function should look something like this:

```
const parseComment = (comment: unknown): string => {
  if (!comment || !isString(comment)) {
    throw new Error('Incorrect or missing comment');
  }

  return comment;
}
```

The function gets a parameter of type *any* and returns it as type *string* if it exists and is of the right type.

The string validation function looks like this

```
const isString = (text: unknown): text is string => {
  return typeof text === 'string' || text instanceof String;
};
```

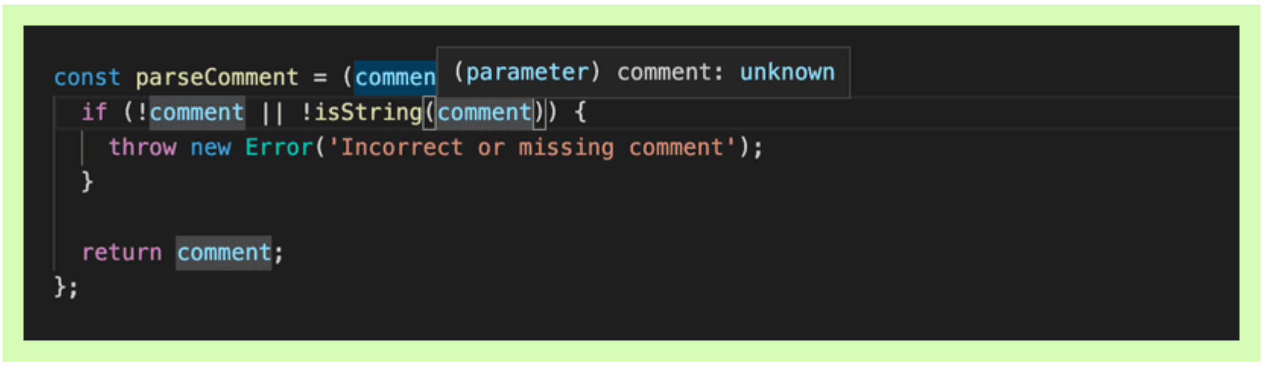
The function is a so called type guard. That means it is a function which returns a boolean *and* which has a *type predicate* as the return type. In our case the type predicate is

```
text is string
```

The general form of a type predicate is `parameterName is Type` where the `parameterName` is the name of the function parameter and `Type` is the targeted type.

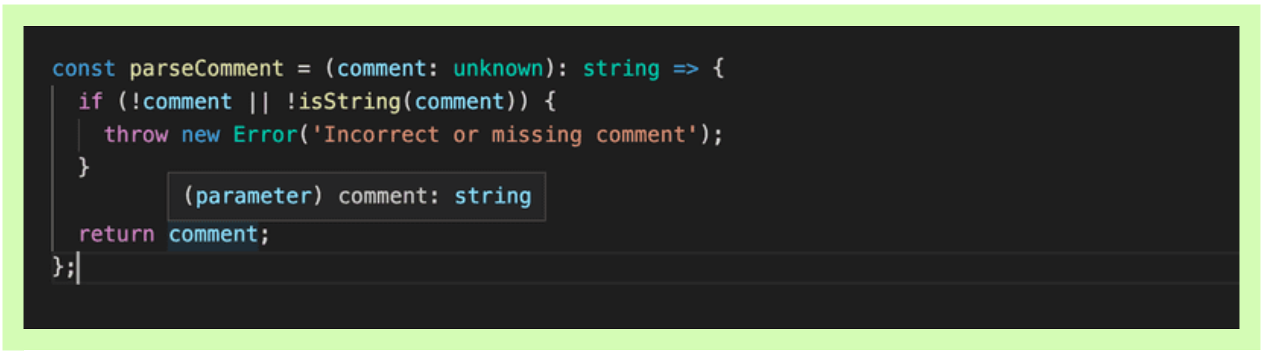
If the type guard function returns true, the TypeScript compiler knows that the tested variable has the type that was defined in the type predicate.

Before the type guard is called, the actual type of the variable *comment* is not known:



```
const parseComment = (commen (parameter) comment: unknown
  if (!comment || !isString(comment)) {
    throw new Error('Incorrect or missing comment');
  }

  return comment;
};
```

But after the call, if the code proceeds past the exception (that is the type guard returned true), compiler knows that *comment* is of the type *string*:



```
const parseComment = (comment: unknown): string => {
  if (!comment || !isString(comment)) {
    throw new Error('Incorrect or missing comment');
  }
           (parameter) comment: string
  return comment;
};
```

Why do we have two conditions in the string type guard?

```
const isString = (text: unknown): text is string => {
  return typeof text === 'string' || text instanceof String;
}
```

would it not be enough to write the guard like this

```
const isString = (text: unknown): text is string => {
  return typeof text === 'string';
}
```

Most likely the simpler form is good enough for all practical purposes. However, if we want to be absolutely sure, both conditions are needed. There are two different ways to create string objects in JavaScript which both work a bit differently with respect to the *typeof* and *instanceof* operators:

```
const a = "I'm a string primitive";
const b = new String("I'm a String Object");
typeof a; --> returns 'string'
typeof b; --> returns 'object'
a instanceof String; --> returns false
```

```
b instanceof String; --> returns true
```

However, it is unlikely that anyone would create a string with a constructor function. Most likely the simpler version of the type guard would be just fine.

Next, let's consider the *date* field. Parsing and validating the date object is pretty similar to what we did with comments. Since TypeScript doesn't really know a type for a date, we need to treat it as a *string*. We should however still use JavaScript level validation to check whether the date format is acceptable.

We will add the following functions

```
const isDate = (date: string): boolean => {
  return Boolean(Date.parse(date));
};

const parseDate = (date: unknown): string => {
  if (!date || !isString(date) || !isDate(date)) {
      throw new Error('Incorrect or missing date: ' + date);
  }
  return date;
};
```

The code is really nothing special. The only thing is, that we can't use a type guard here since a date in this case is only considered to be a *string*. Note, that even though the *parseDate* function accepts the *date* variable as unknown, after we check the type with *isString* its type is set as string, which is why we can give the variable to the *isDate* function requiring a string without any problems.

Finally we are ready to move on to the last two types, Weather and Visibility.

We would like the validation and parsing to work as follows:

```
const parseWeather = (weather: unknown): Weather => {
  if (!weather || !isString(weather) || !isWeather(weather)) {
      throw new Error('Incorrect or missing weather: ' + weather)
  }
  return weather;
};
```

The question is, how can we validate that the string is of a specific form? One possible way to write the type guard would be this:

```
const isWeather = (str: string): str is Weather => {
  return ['sunny', 'rainy', 'cloudy', 'stormy' ].includes(str);
};
```

This would work just fine, but the problem is that the list of possible weathers does not necessarily stay in sync with the type definitions if the type is altered. This is most certainly not good, since we would like to have just one source for all possible weather types.

In our case a better solution would be to improve the actual Weather type. Instead of a type alias we should use the TypeScript enum, which allows us to use the actual values in our code in runtime, not only in the compilation phase.

Let us redefine the type *Weather* as follows:

```
export enum Weather {
  Sunny = 'sunny',
  Rainy = 'rainy',
  Cloudy = 'cloudy',
  Stormy = 'stormy',
  Windy = 'windy',
}
```

Now we can check that a string is one of the accepted values, and the type guard can be written like this:

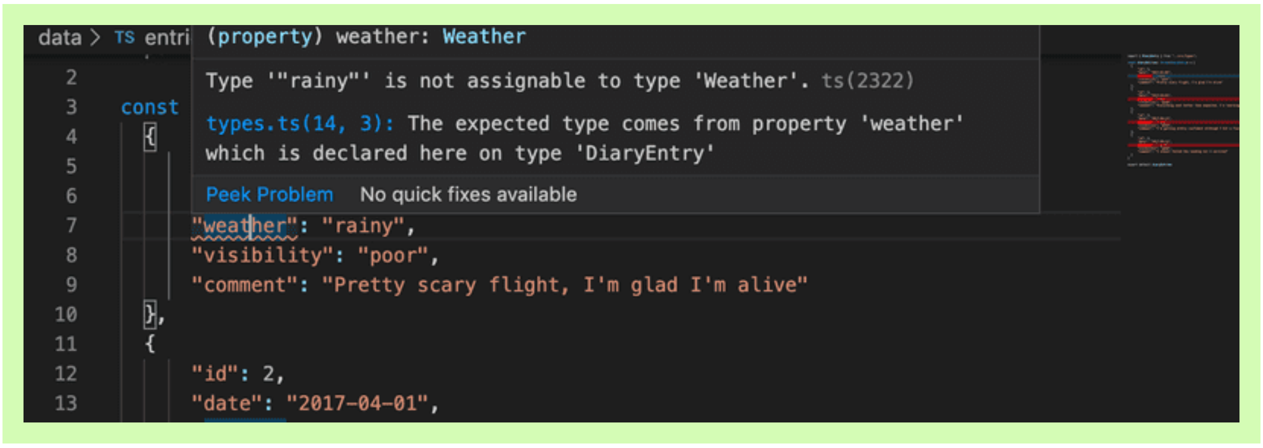
```
// eslint-disable-next-line @typescript-eslint/no-explicit-any
const isWeather = (param: any): param is Weather => {
  return Object.values(Weather).includes(param);
};
```

One thing to notice here is that we have changed the parameter type to *any*. If it would be string, the *includes* check would not compile. This makes sense also if you consider the reusability of the function. Allowing *any* as a parameter, the function can be used with confidence knowing that whatever we might feed to it, the function always tells us whether the variable is a valid weather or not.

The function *parseWeather* can be simplified a bit

```
const parseWeather = (weather: unknown): Weather => {
  if (!weather || !isWeather(weather)) {
      throw new Error('Incorrect or missing weather: ' + weather);
  }
  return weather;
};
```

One issue arises after these changes. Our data does not conform to our types anymore:



```
data > TS entri  (property) weather: Weather

                 Type '"rainy"' is not assignable to type 'Weather'. ts(2322)

                 types.ts(14, 3): The expected type comes from property 'weather'
                 which is declared here on type 'DiaryEntry'

                 Peek Problem   No quick fixes available
const
  {
      "weather": "rainy",
      "visibility": "poor",
      "comment": "Pretty scary flight, I'm glad I'm alive"
  },
  {
      "id": 2,
      "date": "2017-04-01",
```

This is because we cannot just assume a string is an enum.

We can fix this by mapping the initial data elements to *DiaryEntry* type with the *toNewDiaryEntry* function:

```
import { DiaryEntry } from "../src/types";
import toNewDiaryEntry from "../src/utils";

const data = [
  {
      "id": 1,
      "date": "2017-01-01",
      "weather": "rainy",
      "visibility": "poor",
      "comment": "Pretty scary flight, I'm glad I'm alive"
  },
  // ...
]

const diaryEntries: DiaryEntry [] = data.map(obj => {
  const object = toNewDiaryEntry(obj) as DiaryEntry;
  object.id = obj.id;
  return object;
});

export default diaryEntries;
```

Note that since *toNewDiaryEntry* returns an object of the type *NewDiaryEntry* we need to assert it to be *DiaryEntry* with the `as` operator.

Enums are usually used when there is a set of predetermined values which are not expected to change in the future. Usually enums are used for much tighter unchanging values (for example weekdays, months, directions) but since they offer us a great way to validate our incoming values we might as well use them in our case.

We still need to give the same treatment to *visibility*. The enum looks following

```
export enum Visibility {
  Great = 'great',
  Good = 'good',
  Ok = 'ok',
  Poor = 'poor',
}
```

The type guard and the parser are below

```
// eslint-disable-next-line @typescript-eslint/no-explicit-any
const isVisibility = (param: any): param is Visibility => {
  return Object.values(Visibility).includes(param);
};

const parseVisibility = (visibility: unknown): Visibility => {
  if (!visibility || !isVisibility(visibility)) {
      throw new Error('Incorrect or missing visibility: ' + visibility);
  }
  return visibility;
};
```

And finally we can finalize the *toNewDiaryEntry* function that takes care of validating and parsing the fields of the post data. There is however one more thing to take care of. If we try to access the fields of the parameter *object* as follows:

```
const toNewDiaryEntry = (object: unknown): NewDiaryEntry => {
  const newEntry: NewDiaryEntry = {
    comment: parseComment(object.comment),
    date: parseDate(object.date),
    weather: parseWeather(object.weather),
    visibility: parseVisibility(object.visibility)
  };

  return newEntry;
};
```

we notice that the code does not compile. Thie is due to the fact that the `unknown` type does not allow any operations, so also accessing the fields is not possible.

We can fix this by destructuring the fields to variables of the type unknown as follows:

```
type Fields = { comment : unknown, date: unknown, weather: unknown, visibility: unknown };
```

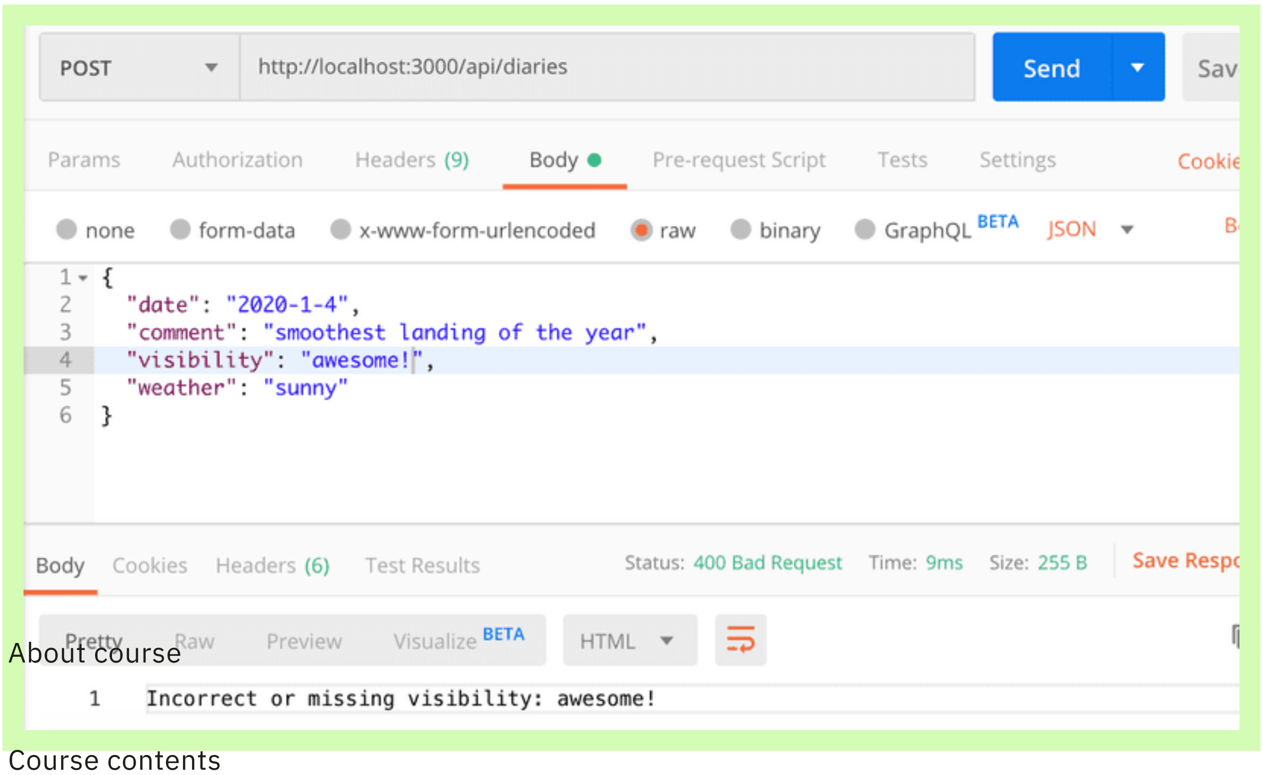
```
const toNewDiaryEntry = ({ comment, date, weather, visibility } : Fields): NewDiaryEntry => {
  const newEntry: NewDiaryEntry = {
    comment: parseComment(comment),
    date: parseDate(date),
    weather: parseWeather(weather),
    visibility: parseVisibility(visibility)
  };

  return newEntry;
};
```

The first version of our flight diary application is now completed!

The other option to bypass the problem would be to use the type *any* for the parameter and disable the lint rule for that line:

```
// eslint-disable-next-line @typescript-eslint/no-explicit-any
const toNewDiaryEntry = (object: any): NewDiaryEntry => {
  const newEntry: NewDiaryEntry = {
    comment: parseComment(object.comment),
    date: parseDate(object.date),
    weather: parseWeather(object.weather),
    visibility: parseVisibility(object.visibility)
  };

  return newEntry;
};
```

If we now try to create a new diary entry with invalid or missing fields we are getting an appropriate error message



http://localhost:3000/api/diaries

```
{
  "date": "2020-1-4",
  "comment": "smoothest landing of the year",
  "visibility": "awesome!",
  "weather": "sunny"
}
```

Status: 400 Bad Request Time: 9ms Size: 255 B

```
Incorrect or missing visibility: awesome!
```

About course
Course contents
FAQ
Partners
Challenge

The source code of the application can be found on GitHub.

Exercises 9.12.-9.13.

9.12: Patientor backend, step5

Create a POST-endpoint */api/patients* for adding patients. Ensure that you can add patients also from the frontend. You can create unique ids of type *string* using the uuid library:

```
import {v1 as uuid} from 'uuid'
const id = uuid()
```

9.13: Patientor backend, step6

Set-up safe parsing, [illegible] and type guards to the POST `/api/patients` request.

Refactor the [illegible] ld t[illegible] enum ty[illegible]

UNIVERSITY OF HELSINKI

Propose changes to material

Part 9b
Previous part

Part 9d
Next part

HOUSTON