AdvanceTypescript API

(https://projects.100xdevs.com/tracks/ts-hard/ts-hard-1)

Pre-requisites

Before you go through this module, make sure you've gone through basic typescript classes.
You understand interfaces , types and how typescript is used in a simple Node.js application

If you understand the following code, you're good to go!

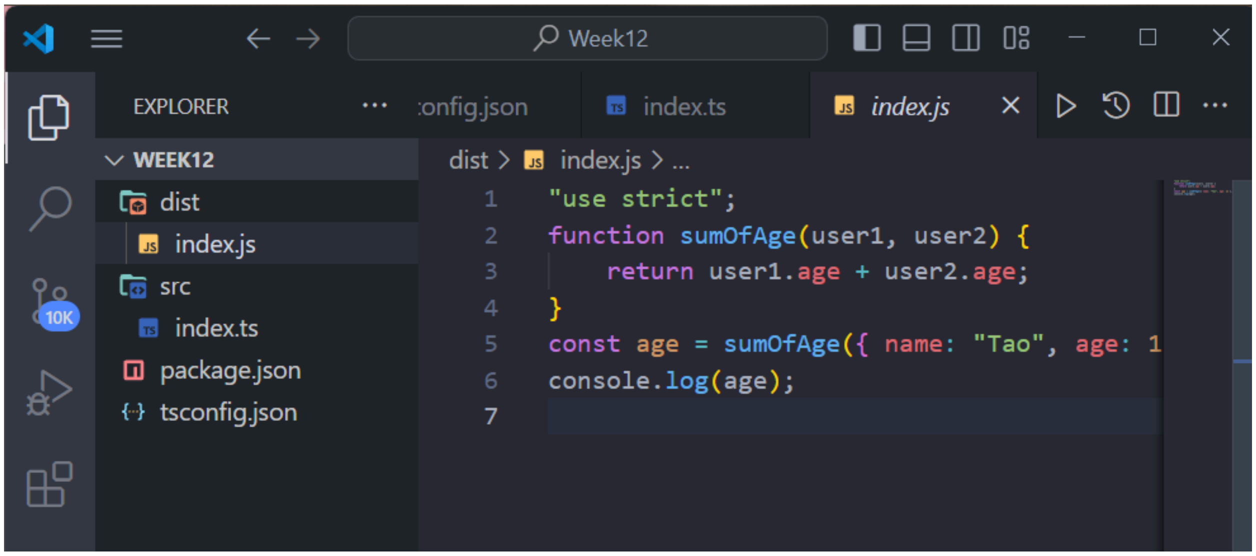
```
interface User {
    name: string;
    age: number;
}

function sumOfAge(user1: User, user2: User) {
  return a.age + b.age;
};

// Example usage
const result = sumOfAge({
    name: "harkirat",
    age: 20
}, {
    name: "raman",
    age: 21
});
console.log(result); // Output: 9
```

Setup procedure

- npx tsc –init
- {

  “rootDir”:”./src”,

```
        "outDir":"./dist"
    }
```



Week12

EXPLORER

WEEK12
- dist
  - index.js
- src
  - index.ts
- package.json
- tsconfig.json

dist > index.js > ...

```js
"use strict";
function sumOfAge(user1, user2) {
    return user1.age + user2.age;
}
const age = sumOfAge({ name: "Tao", age: 1
console.log(age);
```

Pick

Pick allows you to create a new type by selecting a set of properties (Keys) from an existing type (Type).

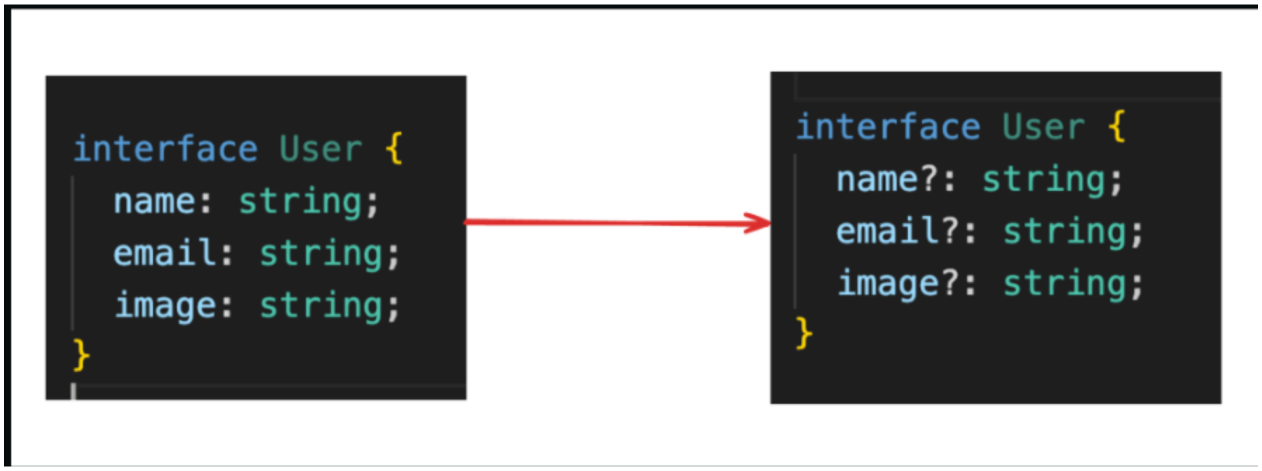
Imagine you have a User model with several properties, but for a user profile display, you only need a subset of these properties.

```ts
interface User {
    id: number;
    name: string;
    email: string;
    createdAt: Date;
  }

  // For a profile display, only pick `name` and `email`
  type UserProfile = Pick<User, 'name' | 'email'>;

  const displayUserProfile = (user: UserProfile) => {
    console.log(`Name: ${user.name}, Email: ${user.email}`);
  };
```

```
interface User {
    id: number;
    name: string;
    age:number;
    email: string;
    password:string;
    createdAt: Date;
  }
//   const user: User = user.findOne({where:{email:"thapa@gassami.com"}})
//database call

  type UpdatedProps = Pick<User,'name'|'age'|'email'>
//   we are using generics
  function updateUser(name:string,age:number,password:string){
    // hit the database and update the user
    // told that we can only change name,age and password
    // argumnet accepted are according to it
    // only when three argument are there its fine but as the argument
grows function start looking ugly
    // one way could be like use updated Props
}

// interface UpdatedProps{
//     name: string,
//     age:number,
//     password:string
// }
// function updateUser(updatedProps:UpdatedProps){

// }
// problem is that suppose we have changed the type of age suppose from
number to string then we have to change it in two places interface User
and interface UpdatedProps, makes it possible that we can miss it.
// We want updatedProps to be subset of User we will do it by Pick API
```

Partial

Partial makes all properties of a type optional, creating a type with the same properties, but each marked as optional.



```
interface User {
  name: string;
  email: string;
  image: string;
}
```

```
interface User {
  name?: string;
  email?: string;
  image?: string;
}
```

Specifically useful when you want to do updates

```
interface User {
    id: string;
    name: string;
    age: string;
    email: string;
    password: string;
};

type UpdateProps = Pick<User, 'name' | 'age' | 'email'>

type UpdatePropsOptional = Partial<UpdateProps>

function updateUser(updatedProps: UpdatePropsOptional) {
    // hit the database tp update the user
}
updateUser({})
```

```
interface User {
  id: string;
  name: string;
  age: string;
  email: string;
  password: string;
}

type UpdateProps = Pick<User, "name" | "age" | "email">;
```

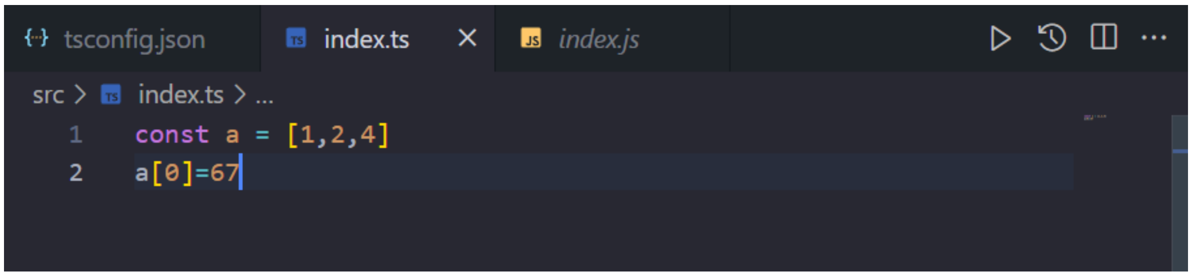
```
//We have to give all three else we will see an error

//user in real world might not be updating all name,age and email together
in one go , they may be updating name in one case , suppose a user want to
update name ,age and email all
type UpdatePropsOptional = Partial<UpdateProps>;
// type UpdatePropsOptional1 {
//     name?:string,
//     age?:string,
//     email?:string
// }

function updateUser(updatedProps: UpdatePropsOptional) {
  // hit the database tp update the user
//   optionally allow all three

}
updateUser({});
```

Readonly

When you have a configuration object that should not be altered after initialization, making it **Readonly** ensures its properties cannot be changed

```
interface Config {
    readonly endpoint: string;
    readonly apiKey: string;
  }

  const config: Readonly<Config> = {
    endpoint: 'https://api.example.com',
    apiKey: 'abcdef123456',
  };

  // config.apiKey = 'newkey'; // Error: Cannot assign to 'apiKey' because
it is a read-only property.We don't want developer to mistakenly update
the apiKey, project itself wont compile since it is Readonly
```

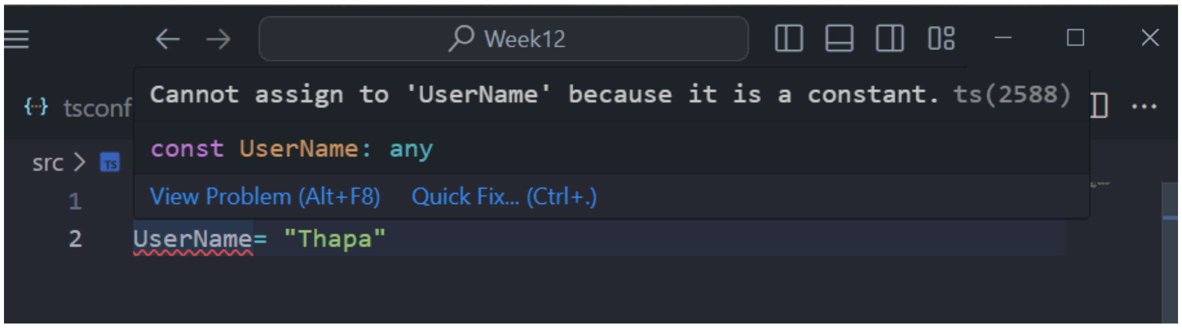
This is compile time checking , not runtime (unlike const)



```
const a = [1,2,4]
a[0]=67
```

We can see that Typescript hasn't raised any error as we are modifying a const value. Basic reason is that we are not really changing a , means reference of array a ,we are only changing the internal value of array a , but our intention behind using const are not fulfilled

But if we do for string typescript is able to raise error,



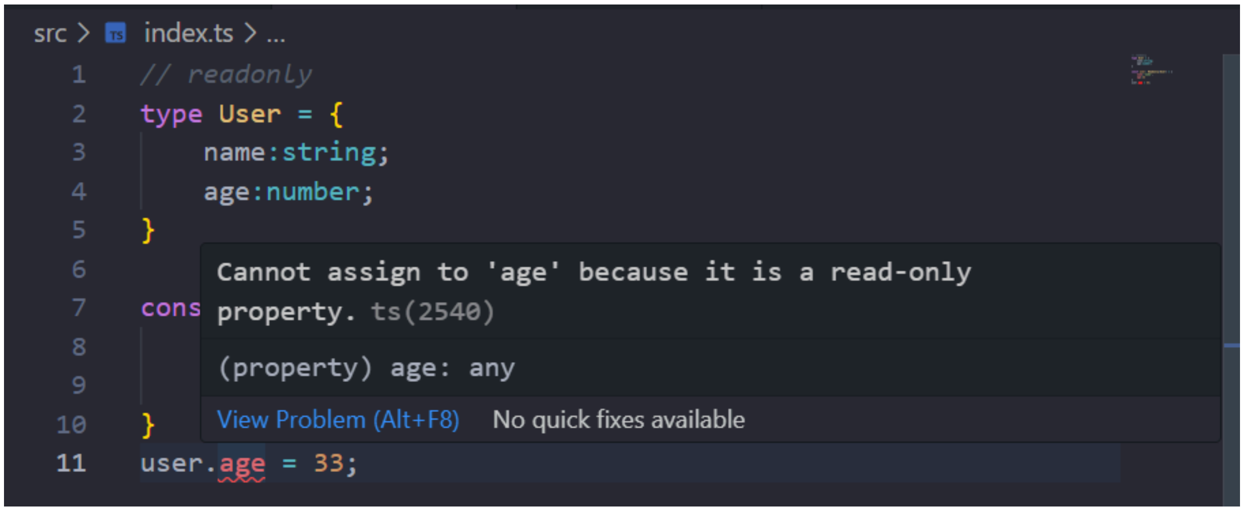
```
Cannot assign to 'UserName' because it is a constant. ts(2588)
const UserName: any
```

```
UserName= "Thapa"
```

Example:

```
// readonly
type User = {
    name:string;
    age:number;
}

const user: Readonly<User> = {
    name:'John',
    age:22
}
user.age = 33;
```

Raising an error



```
src > index.ts > ...
// readonly
type User = {
    name:string;
    age:number;
}

cons
    
    
}
user.age = 33;
```

Cannot assign to 'age' because it is a read-only property. ts(2540)

(property) age: any

View Problem (Alt+F8)   No quick fixes available

Record and Map

Record

Record let's you give a cleaner type to objects

You can type objects like follows:-

```
interface User {
    id: string;
    name: string;
  }

  type Users = { [key: string]: User };

  const users: Users = {
    'abc123': { id: 'abc123', name: 'John Doe' },
    'xyz789': { id: 'xyz789', name: 'Jane Doe' },
  };
```

```
type User = {
  id:string;
  username:string;
}

// objects in typescript
type Users = {
  [key:string]:User;
}

const users = {
  "asda1":{
    id:"asda1",
    username:"thapa"
  },
  "abcd1":{
    id:"abcd1",
    username:"thapuu"
  }
}
type UsersAge = {
  [key: string]: number;
}

const users1:UsersAge = {
    "sadasd":1321,
    "adaad":12
}
```

Or use Record

```
interface User {
  id: string;
  name: string;
}

type Users = Record<string, User>;

const users: Users = {
  'abc123': { id: 'abc123', name: 'John Doe' },
  'xyz789': { id: 'xyz789', name: 'Jane Doe' },
};

console.log(users['abc123']); // Output: { id: 'abc123', name: 'John Doe'
}
```

Record give cleaner types to object

```
type Users = Record<string, number>;
// Users is Record key is a string and value is a number

const users1:Users = {
    "sadasd":1321,
    "adaad":12
}
```

```
type Users = Record<string,{age:number; name:string} >;
// Record is something only typescript compiler understand

const users1:Users = {
    "sadasd":{age:21, name:"Thapiui"},
    "adaad":{age:22, name:"ConfuThapu"}
}
```

Maps

```
interface User {
  id: string;
  name: string;
}

// Initialize an empty Map
const usersMap = new Map<string, User>();

// Add users to the map using .set
usersMap.set('abc123', { id: 'abc123', name: 'John Doe' });
usersMap.set('xyz789', { id: 'xyz789', name: 'Jane Doe' });

// Accessing a value using .get
console.log(usersMap.get('abc123')); // Output: { id: 'abc123', name:
'John Doe' }
```

It is a javascript concept

```
const users = new Map();
users.set("asdada", { name: "ssa", age: 23, email: "asfg@" });
users.set("asdada1", { name: "ssa1", age: 33,email:"asfg@1" });

// We used to do
// users["asdad1"]
const user = users.get("asdada");
console.log(user);
```

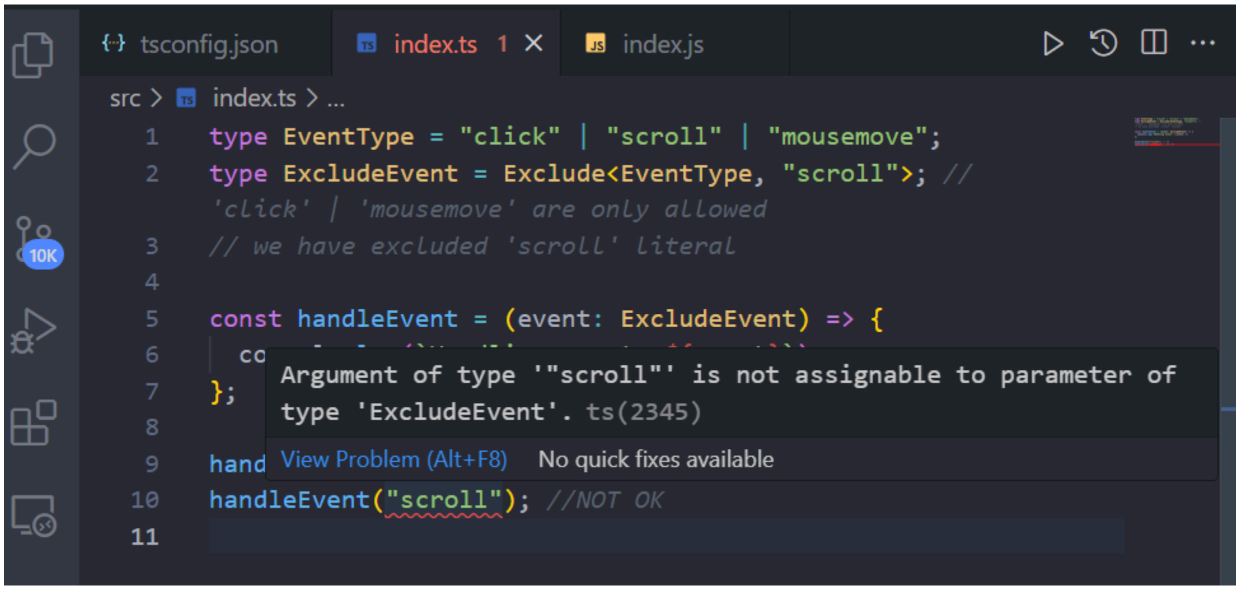
When we are generating Map we can specify type

```
type User ={
  name:string;
  age:number;
  email:string;
}

const users = new Map<string,User>();
users.set("asdada", { name: "ssa", age: 23, email: "asfg@" });
users.set("asdada1", { name: "ssa1", age: 33,email:"asfg@1" });

// We used to do
// users["asdad1"]
const user = users.get("asdada");
console.log(user);
```

Exclude

In a function that can accept several types of inputs but you want to exclude specific types from being passed to it.

```
type Event = 'click' | 'scroll' | 'mousemove';
type ExcludeEvent = Exclude<Event, 'scroll'>; // 'click' | 'mousemove'

const handleEvent = (event: ExcludeEvent) => {
  console.log(`Handling event: ${event}`);
};

handleEvent('click'); // OK
type EventType = "click" | "scroll" | "mousemove";
type ExcludeEvent = Exclude<EventType, "scroll">; // 'click' | 'mousemove' are only allowed
// we have excluded 'scroll' literal

const handleEvent = (event: ExcludeEvent) => {
  console.log(`Handling event: ${event}`);
};

handleEvent("click"); // OK
handleEvent("scroll"); //NOT OK
```



```
type EventType = "click" | "scroll" | "mousemove";
type ExcludeEvent = Exclude<EventType, "scroll">; // 'click' | 'mousemove' are only allowed
// we have excluded 'scroll' literal

const handleEvent = (event: ExcludeEvent) => {
  co
};

hand
handleEvent("scroll"); //NOT OK
```

```
Argument of type '"scroll"' is not assignable to parameter of type 'ExcludeEvent'. ts(2345)
```

Type inference in zod

(https://zod.dev/?id=type-inference **)**

When using zod, we're done runtime validation.

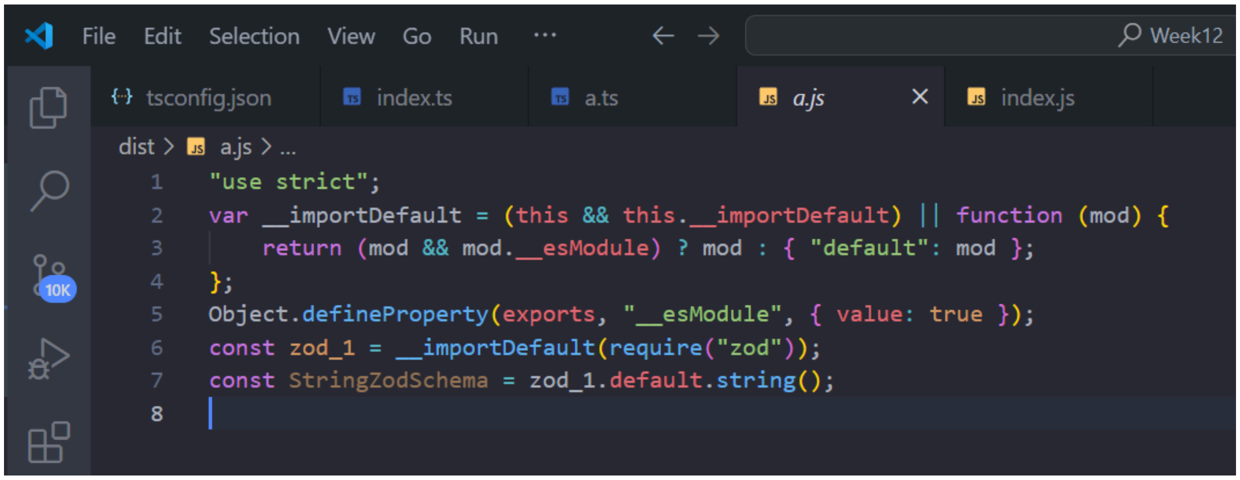
For example, the following code makes sure that the user is sending the right inputs to update their profile information

npm install express @types/express zod

Examples:

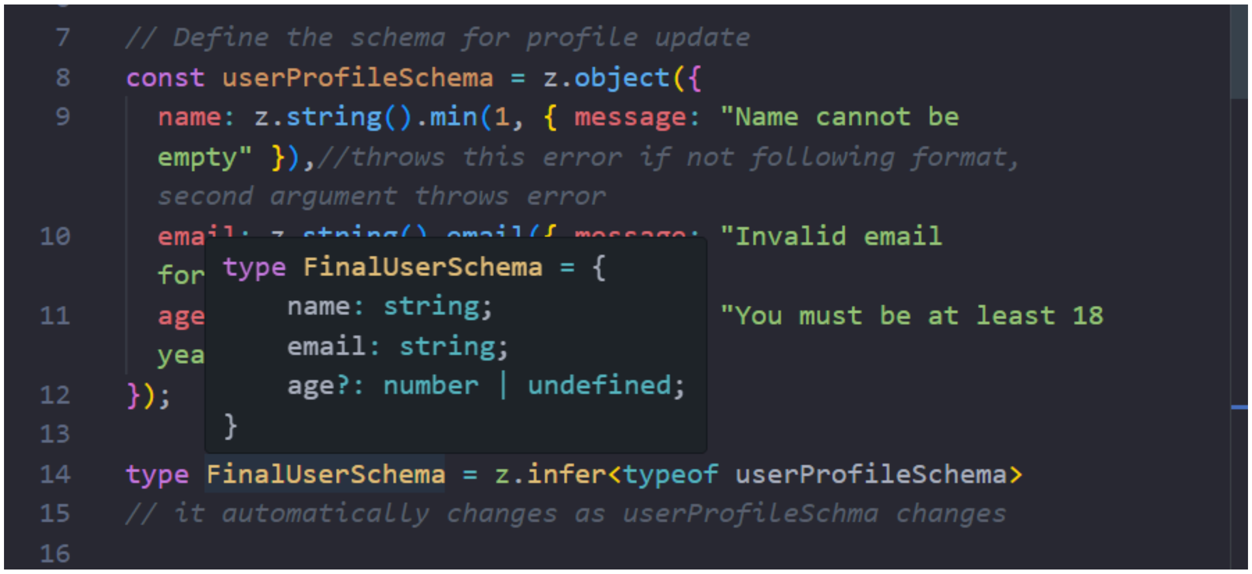
```
import z from 'zod';

const StringZodSchema = z.string();
type StringZodType = z.infer<typeof StringZodSchema>;
```

Let’s se its js file



```
"use strict";
var __importDefault = (this && this.__importDefault) || function (mod) {
    return (mod && mod.__esModule) ? mod : { "default": mod };
};
Object.defineProperty(exports, "__esModule", { value: true });
const zod_1 = __importDefault(require("zod"));
const StringZodSchema = zod_1.default.string();
```

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

// initialize an  empty express app
const app = express();

// Define the schema for profile update
const userProfileSchema = z.object({
  name: z.string().min(1, { message: "Name cannot be empty" }),//throws
this error if not following format, second argument throws error
  email: z.string().email({ message: "Invalid email format" }),
  age: z.number().min(18, { message: "You must be at least 18 years old"
}).optional(),
});

type FinalUserSchema = z.infer<typeof userProfileSchema>
// it automatically changes as userProfileSchma changes

app.put("/user", (req, res) => {
  // checking whether body follows the userProfileSchema or not
  const { success } = userProfileSchema.safeParse(req.body);
  // const updateBody = req.body; // how to assign a type to updateBody?,
problem is that updateBody type is any
  // something like this
  // const updateBody: {
  //   name:string;
  //   email:string;
  //   age?:number;
  // } = req.body
  // wouldn't it be better that we can infer this type using the
userProfileSchema or basically zod Schema, and we don't have to write it
twice
  const updateBody: FinalUserSchema = req.body;

  if (!success) {
    res.status(411).json({});
    return
  }
  // update database here
  res.json({
    message: "User updated"
  })
});

app.listen(3000);
```



```
// Define the schema for profile update
const userProfileSchema = z.object({
  name: z.string().min(1, { message: "Name cannot be
  empty" }),//throws this error if not following format,
  second argument throws error
  email: z.string().email({ message: "Invalid email
  for
  age                                "You must be at least 18
  yea
});
type FinalUserSchema = {
    name: string;
    email: string;
    age?: number | undefined;
}

type FinalUserSchema = z.infer<typeof userProfileSchema>
// it automatically changes as userProfileSchma changes
```