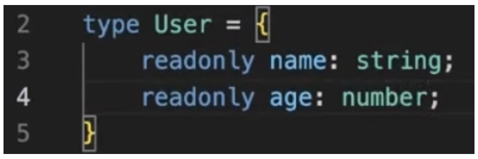
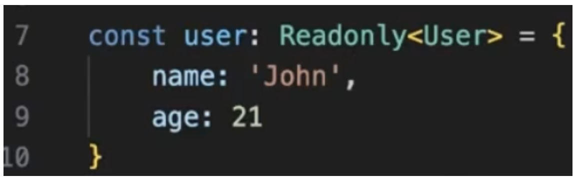
14.3 - Typescript 3

TODO:

1. 7 API calls

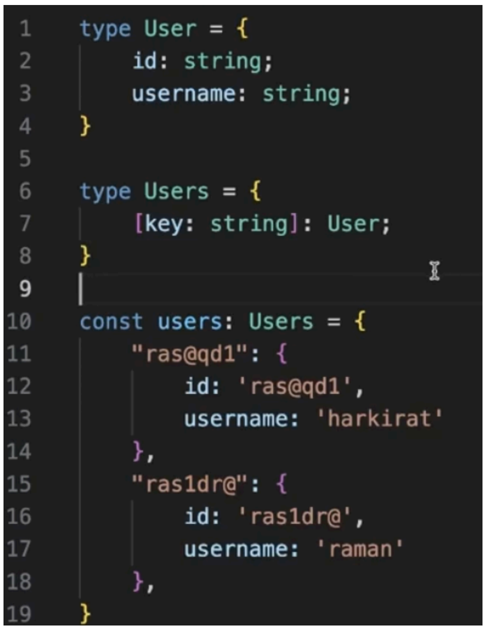
- To run your file inside the dist folder from your root folder, you don't need to cd into the dist folder everytime, you can also write: node dist/index.js
- **API: Pick**
    - Allows you to create a new type based on an existing type with selected properties.
    - Such as User has name, age, address. Use Pick to create UserProfile with name and age only.
        - **const UserProfile = Pick<User, 'name' | 'age'>;**
            - Now, UserProfile is a new type. Similar to how selectors work in recoil.
- General rule of thumb: A function should not have more than 5 arguments.
- **API: Partial**
    - The ability to make an argument optional with a question mark (but with an API).
    - type User = {name: string, age: number, ssn?: number}
        - Where ssn is optional
    - To make every field optional: type userOptional = Partial<User>
        - userOptional is: {name?: string, age?: number, ssn?: number}
    - Fun fact: Making a field optional means - ssn: number | undefined
        - The undefined is what makes it optional
- **API: ReadOnly**
    - The problem is, even if you have an object which is const, you can still update its individual properties and even with an array, you can change the elements inside of the array despite it being const.
    - **Note**: You cannot reassign const variables to a new object or a new array, but if they're an object or an array, you can change their internal values or properties. Meaning you cannot change the reference of the variable (it needs to point to the same object or array) but its internal values can be changed as that does not affect the reference.
    - Thus, to make it absolutely constant and not allow any change whatsoever (even to internal properties or individual positions), use ReadOnly API.
    - 

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

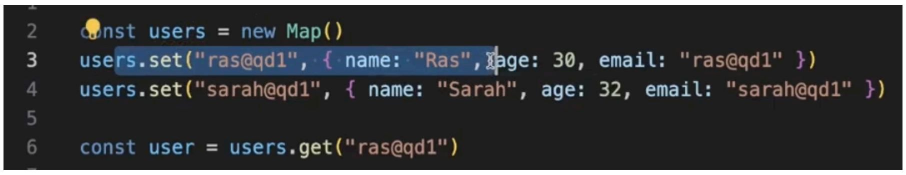
        - Append *readonly* to the variable or the entire object/type.
    - 

```
const user: Readonly<User> = {
    name: 'John',
    age: 21
}
```

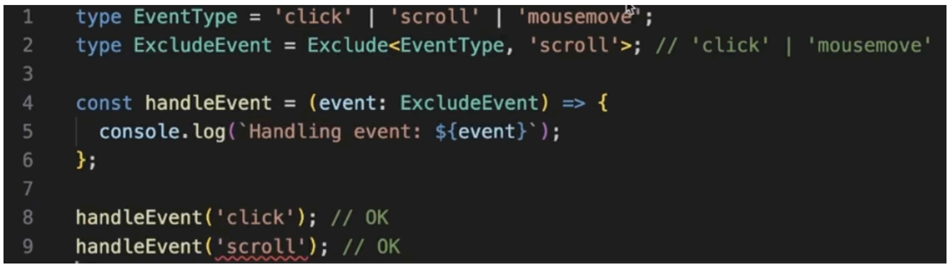
- Native objects type (key-value pairs type) in typescript:

- 

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

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

const users: Users = {
    "ras@qd1": {
        id: 'ras@qd1',
        username: 'harkirat'
    },
    "ras1dr@": {
        id: 'ras1dr@',
        username: 'raman'
    },
}
```

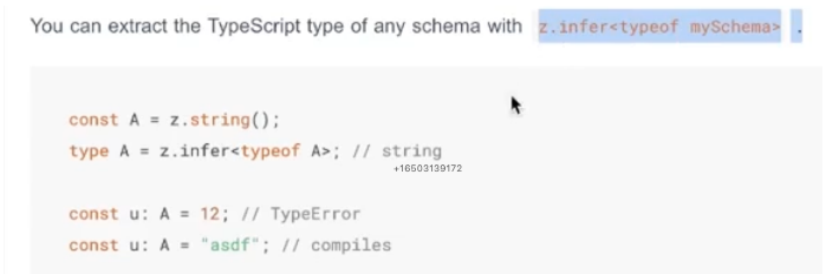
    - Key is a keyword covered in square brackets [] with its type.
- The above syntax is extremely ugly, which is why Records were introduced.
- **API: Records**
    - To make the key-value pair type more readable, records are always used over the native ugly syntax.
    - type UserObjectType = Record<string, number>
        - This is an object where the key is a string and the value is a number.
- **API: Map**
    - To create key-value pairs (aka objects) more easily (actual initialising objects, getting, deleting etc).
    - 

```
const users = new Map()
users.set("ras@qd1", { name: "Ras", age: 30, email: "ras@qd1" })
users.set("sarah@qd1", { name: "Sarah", age: 32, email: "sarah@qd1" })

const user = users.get("ras@qd1")
```

    - You can also specify the type when initialising the map: *const users = new Map<string, {name: string, age: number, email: string}>();*
    - You will want to use maps over records as they allow stricter type checking and data manipulation.
- **API: Exclude**
    - 

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

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

handleEvent('click'); // OK
handleEvent('scroll'); // OK
```

    - To exclude a certain type.
- **API: Zod Type Inference - Most Usable**
    - Basically, infer the type from a zod schema so that you don't have to repeat the type when getting the user data from req.body (which you need to be of the same type as the zod schema).

- 

  You can extract the TypeScript type of any schema with `z.infer<typeof mySchema>` .

  ```
  const A = z.string();
  type A = z.infer<typeof A>; // string

  const u: A = 12; // TypeError
  const u: A = "asdf"; // compiles
  ```

- Zod z gives you a generic function (<>) .infer<typeof A> where you can get the type of a schema A (this schema is pretty simply but works the same for an object).
- **Note**: This is a lot more useful in the frontend where the user is actually inputting the value and you need the strict type checking there, as the backend is assumed to receive the correct input.