TS Congress

September 21 & 22, 2023
4pm CEST / 7am PST / 10am EST

The Type of Conference Developers Need

№1 MOST ADOPTED TECHNOLOGY
Based on year-on-year growth

20+ INDUSTRY EXPERTS
Sharing know-hows

5K+ WEB DEVS
Gathering in the cloud

Stream page

Reserve a spot

01.05:05.35

Practice TypeScript Techniques Building React Server Components App

Maurice de Beijer
@mauricedb

- Maurice de Beijer
- The Problem Solver
- Microsoft MVP
- Freelance developer/instructor
- Currently at https://someday.com/
- Twitter: @mauricedb
- Web: http://www.TheProblemSolver.nl
- E-mail: maurice.de.beijer@gmail.com

Topics

- **Compiling** the code to catch type errors early
- Using the ***satisfies*** operator
- Use **type mappings** to optimize your code
- **Custom type mappings** are really powerful
  - Can help detect all sorts of errors quickly
  - Can make types more readable in IntelliSense
- Use **Template Literal Types** to manipulate types
- Use **Opaque Types** to help with type safety
- Make TypeScript even stricter and **catch more potential errors**
- Prevent objects from being **mutated by accident**
- **Improve performance** when importing large JSON files

Type it out by hand?

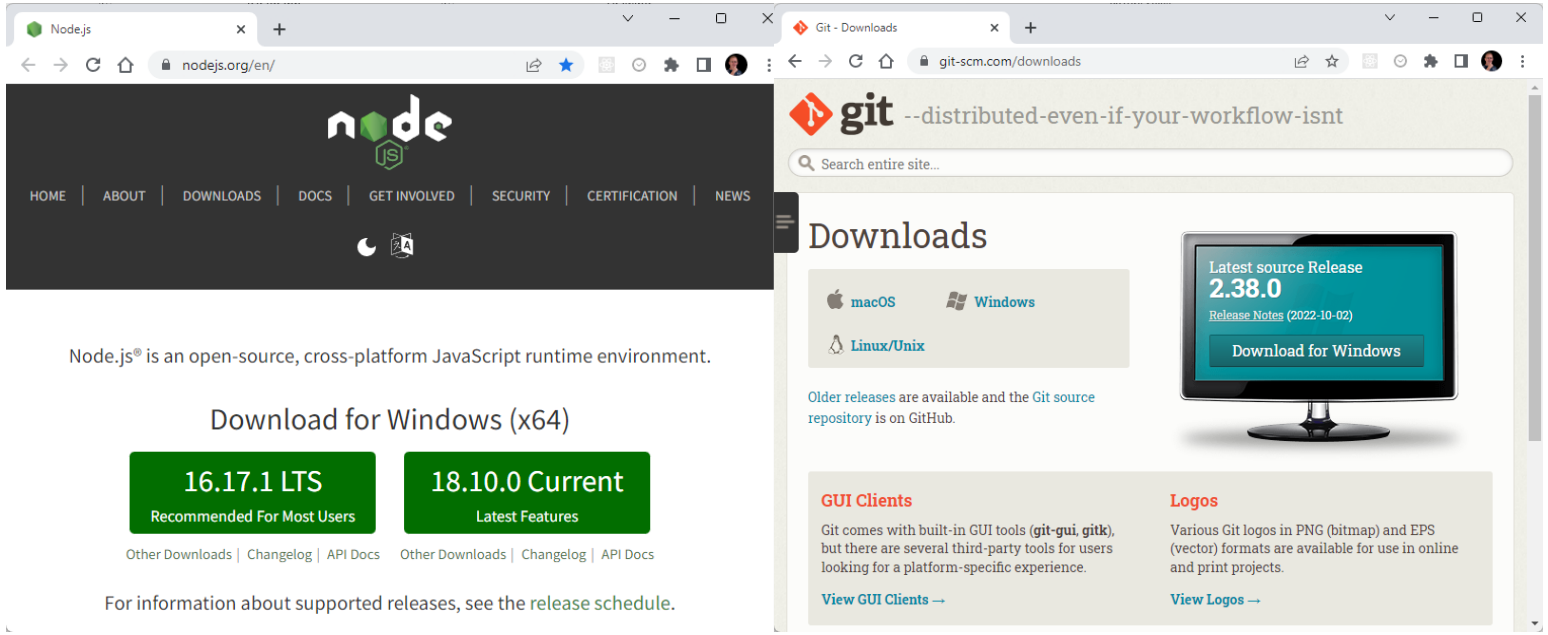
"Typing it drills it into your brain much better than simply copying and pasting it. You're forming new neuron pathways. Those pathways are going to help you in the future. Help them out now!"

Prerequisites

Install Node & NPM

Install the GitHub repository

Install Node.js & NPM



Following Along

- Repo: https://github.com/mauricedb/ts-conf-2023-ws
- Slides: https://bit.ly/ts-conf-2023-ws

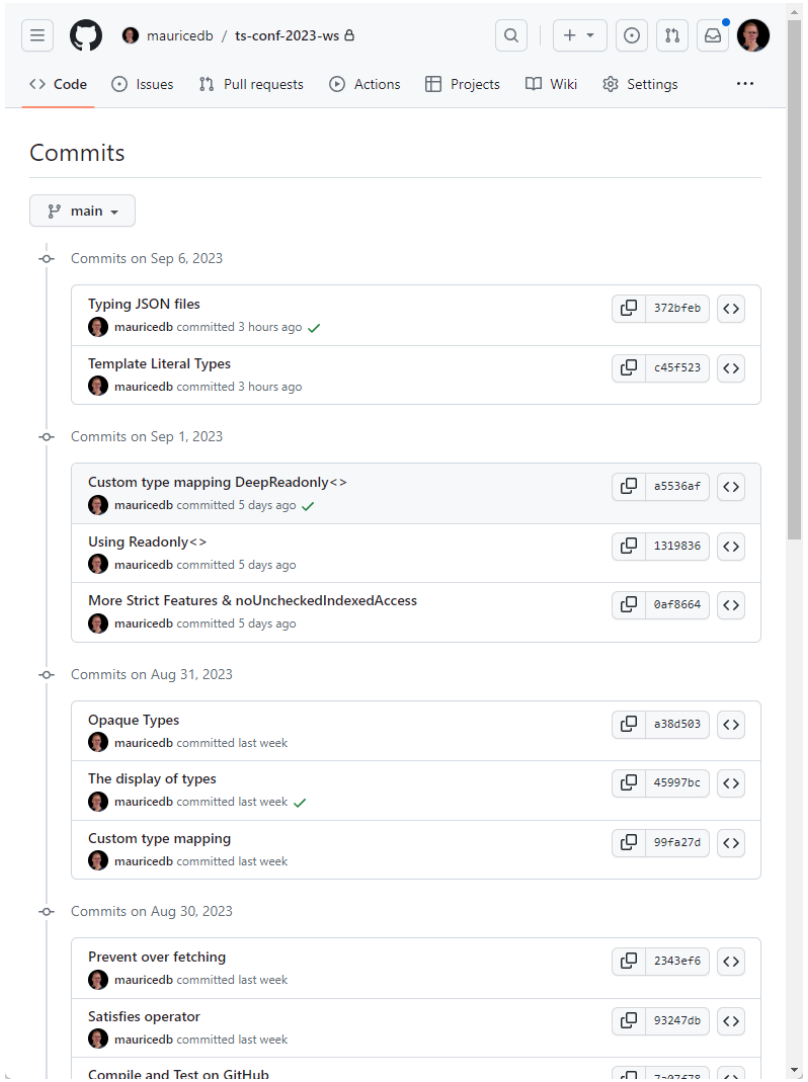
Create a new Next.js app

```
PS C:\Repos> npx create-next-app@latest
Need to install the following packages:
  create-next-app@13.4.19
Ok to proceed? (y)
√ What is your project named? ... ts-conf-2023-ws
√ Would you like to use TypeScript? ... No / Yes
√ Would you like to use ESLint? ... No / Yes
√ Would you like to use Tailwind CSS? ... No / Yes
√ Would you like to use `src/` directory? ... No / Yes
√ Would you like to use App Router? (recommended) ... No / Yes
√ Would you like to customize the default import alias? ... No / Yes
Creating a new Next.js app in C:\Repos\ts-conf-2023-ws.
```

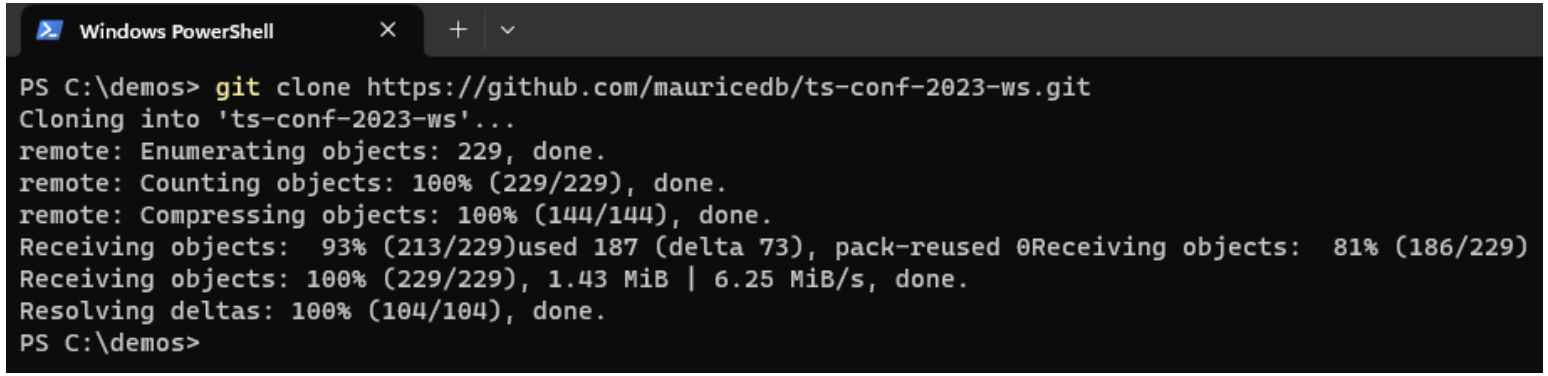
Adding Shadcn components

```
PS C:\Repos\ts-conf-2023-ws> npx shadcn-ui@latest init
√ Would you like to use TypeScript (recommended)? ... no / yes
√ Which style would you like to use? » Default
√ Which color would you like to use as base color? » Zinc
√ Where is your global CSS file? ... src\app\globals.css
√ Would you like to use CSS variables for colors? ... no / yes
√ Where is your tailwind.config.js located? ... tailwind.config.ts
√ Configure the import alias for components: ... @/components
√ Configure the import alias for utils: ... @/lib/utils
√ Are you using React Server Components? ... no / yes
√ Write configuration to components.json. Proceed? ... yes

√ Writing components.json...
√ Initializing project...
√ Installing dependencies...

Success! Project initialization completed.
```

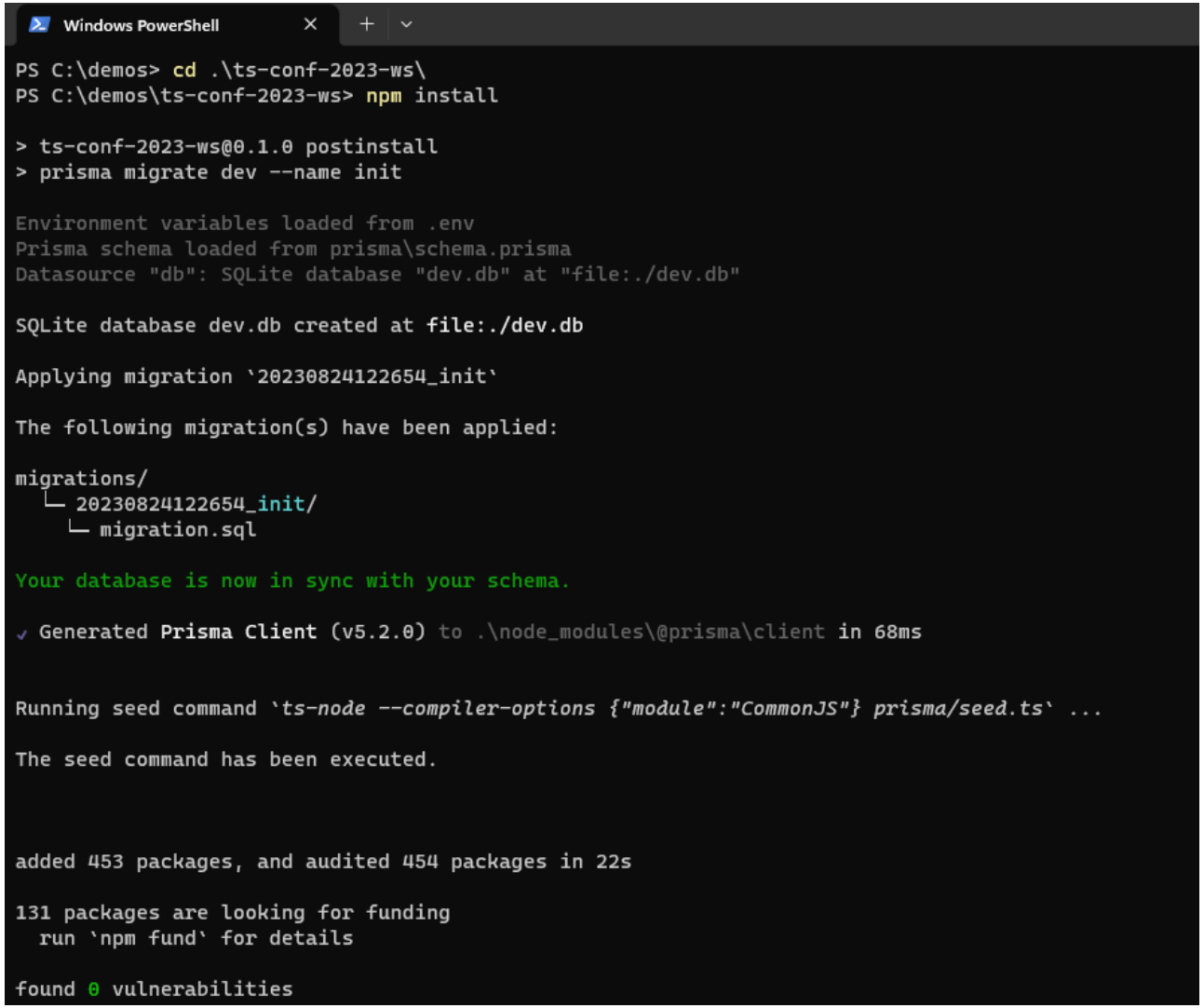
The changes



Clone the GitHub Repository



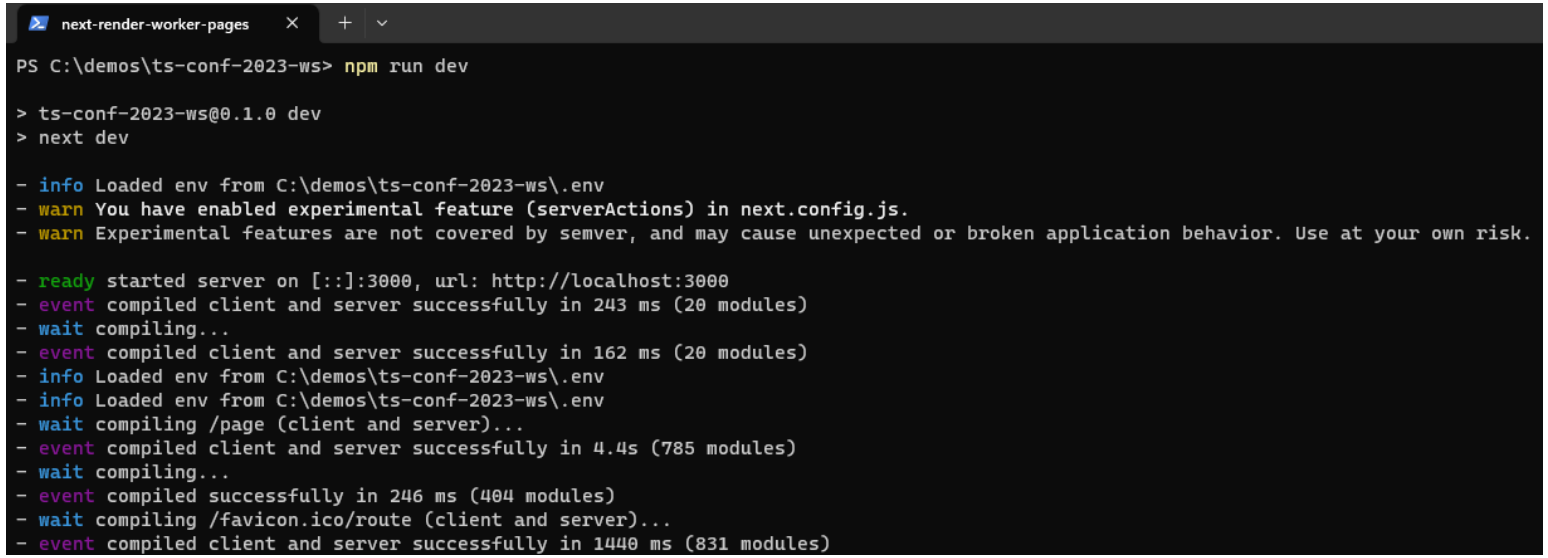
Install NPM Packages



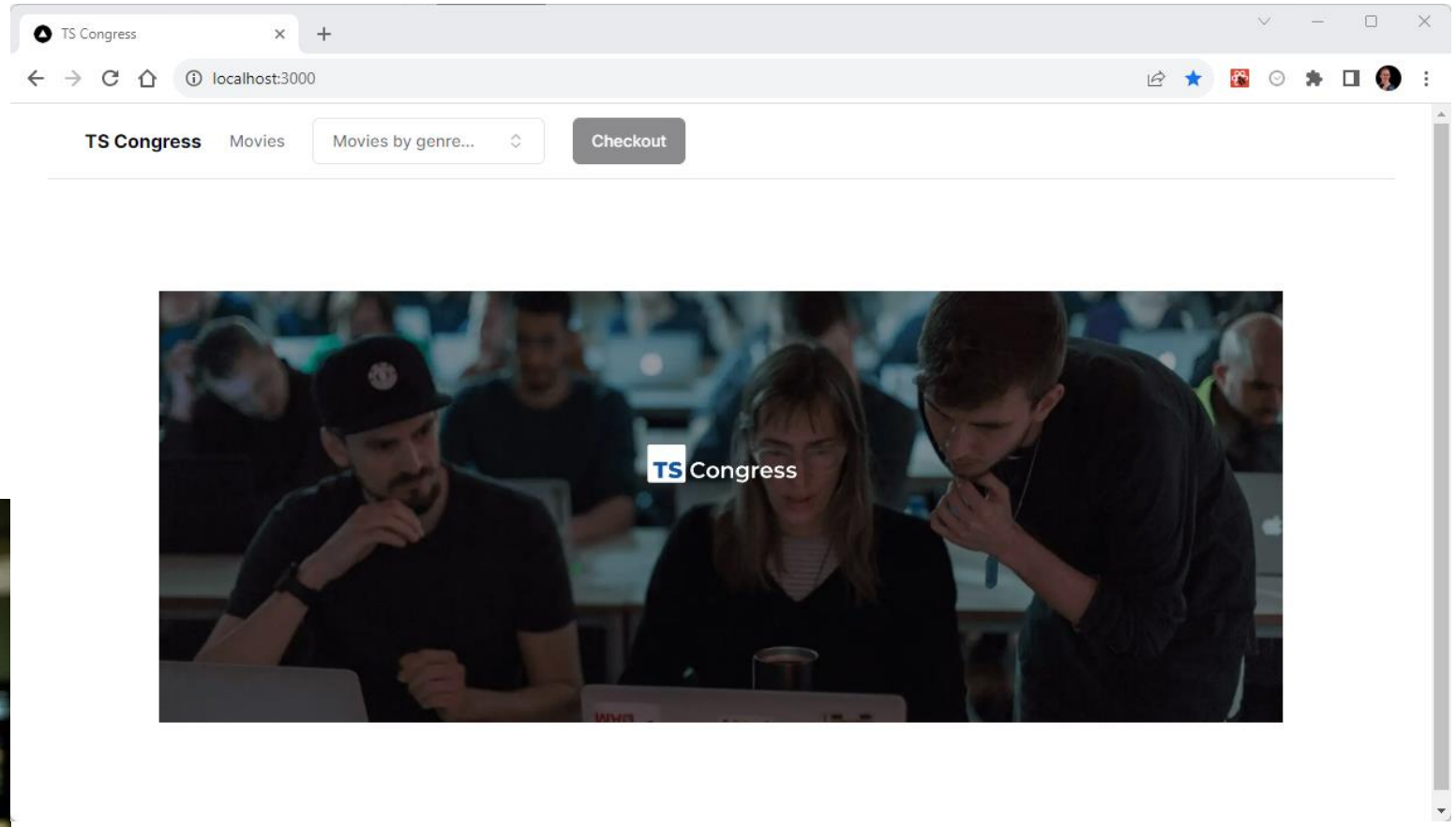
Start branch

- Start with the **00-start** branch
  - `git checkout --track origin/00-start`

Start the application



The application

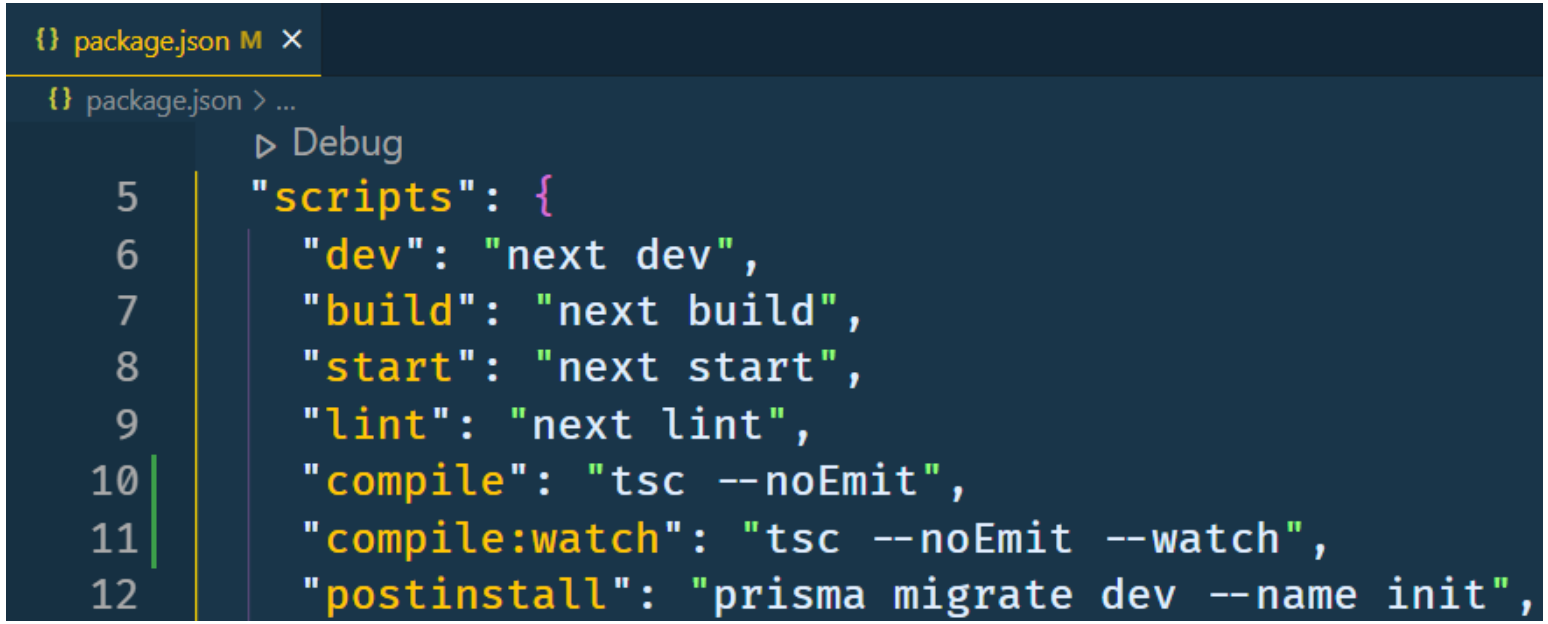


Compiling the code

Compiling the code

- Quite often **TypeScript code is not type checked** during development
  - Create React App use Babel (JavaScript)
  - Vite uses ESBuild (Go)
  - Next.js uses SWC (Rust)
- These **remove TypeScript annotations** and treat the code as modern JavaScript
  - This means that TypeScript type **errors can go unnoticed** ☹
- Run the **TypeScript compiler to check the code**

package.json



npm run compile

```
TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE  AZURE  GITLENS  COMMENTS

[10:47:59] Starting compilation in watch mode...

src/components/movie-card.tsx:50:61 - error TS2345: Argument of type 'Pick<{ id: number; title: string; overview: string; releas
e_date: string; backdrop_path: string; poster_path: string; popularity: number; vote_average: number; vote_count: number; }, "ti
tle" | "id" | "overview" | "backdrop_path" | "vote_average">' is not assignable to parameter of type '{ id: number; title: strin
g; overview: string; release_date: string; backdrop_path: string; poster_path: string; popularity: number; vote_average: number;
 vote_count: number; }'.
  Type 'Pick<{ id: number; title: string; overview: string; release_date: string; backdrop_path: string; poster_path: string; po
pularity: number; vote_average: number; vote_count: number; }, "title" | "id" | "overview" | "backdrop_path" | "vote_average">'
is missing the following properties from type '{ id: number; title: string; overview: string; release_date: string; backdrop_pat
h: string; poster_path: string; popularity: number; vote_average: number; vote_count: number; }': release_date, poster_path, pop
ularity, vote_count

50          <Button variant="secondary" onClick={() => addMovie(movie)}>
                                                       ~~~~~

[10:48:01] Found 1 error. Watching for file changes.
```

shopping-cart.tsx

npm run compile


```
TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE  AZURE  GITLENS  COMMENTS

[10:52:09] Starting compilation in watch mode...

[10:52:11] Found 0 errors. Watching for file changes.
```

Compile and Test on GitHub

Compile and Test on GitHub

- Make sure to **check your code with each pull request** on GitHub
  - Include ESLint and Next build
- **Generate types** from other sources if appropriate
  - Prisma, GraphQL, OpenAPI etc.
  - When included in GitHub they can be out of date ☹

compile-and-test.yml

compile-and-test.yml U

.github > workflows > compile-and-test.yml

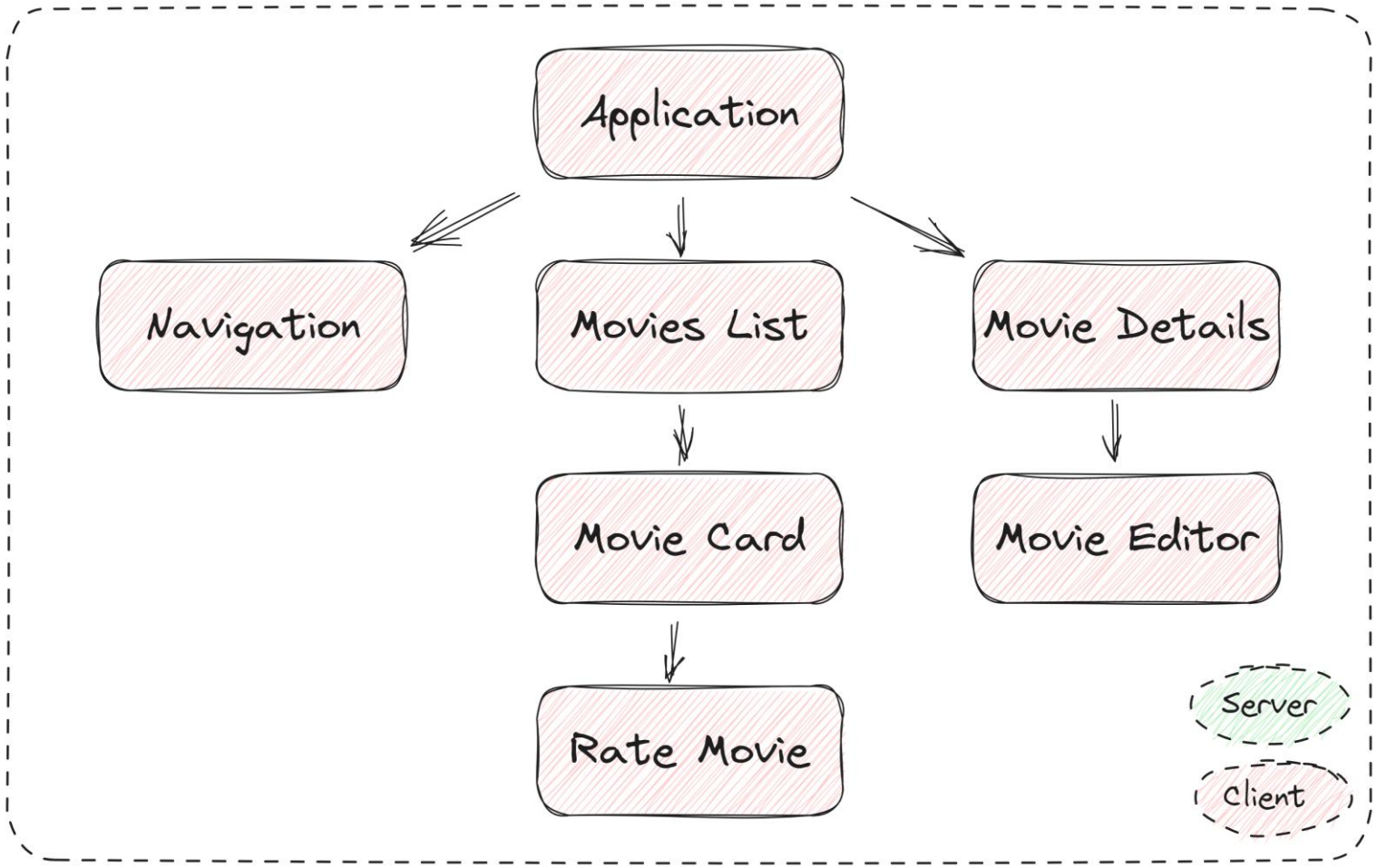
```yaml
name: Compile and Test
on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  test:
    timeout-minutes: 15
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: 18
          cache: 'npm'
      - name: Install dependencies
        run: npm ci --no-audit --prefer-offline
      - name: Type check
        run: npm run compile
      - name: Lint
        run: npm run lint
      # - name: Test
      #   run: npm run test
      - name: Build
        run: npm run build
```

What are React Server Components?

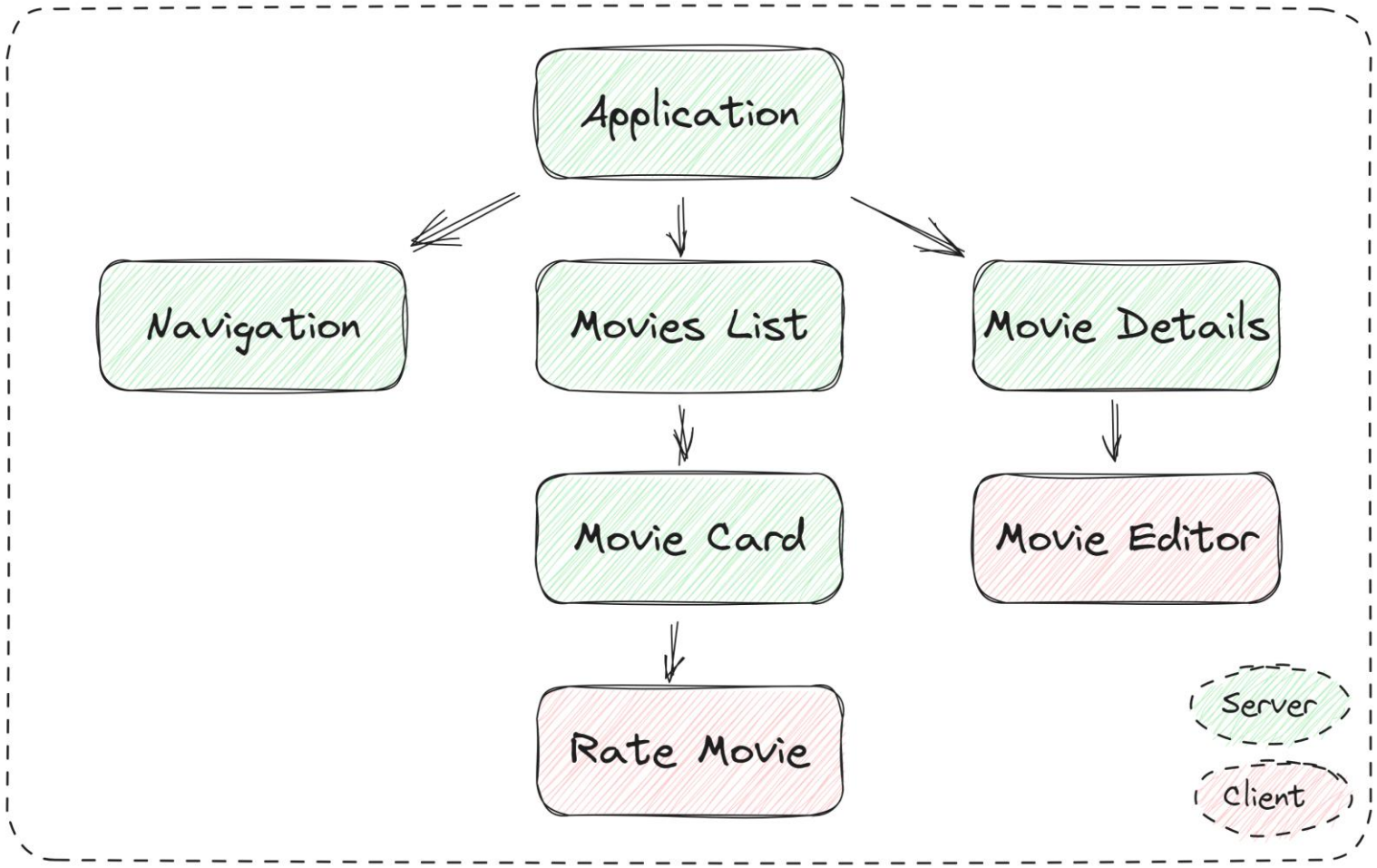
React Server Components

- React Server Components **only execute on the server**
  - Traditionally React components always execute in the browser
- This is **not the same as Server Side Rendering**
  - With SSR components are executed both on the client and server
- Applications are a **combination of server and client components**
- The result is that the back and front-end **code are more integrated**
  - Leading to **better type safety** ☺

Before RSC


Application
Navigation
Movies List
Movie Details
Movie Card
Movie Editor
Rate Movie
Server
Client

With RSC


Application
Navigation
Movies List
Movie Details
Movie Card
Movie Editor
Rate Movie
Server
Client

React Server Components

- Server components can be **asynchronous**
  - Great to load data from some API
- Server components **render just once**
  - No re-rendering with state changes or event handling
- The server component **code is not send to the browser**
  - Can safely use secure API key's etc.
  - Smaller bundle sizes
- ☞ React Server Components require TypeScript 5.1 ☜

Client Component

- **Server components can render server and client components**
  - Client components can only render client components
- Adding **'use client'** to the top of a component makes it a client component

Server Component

genre-loader.tsx

src > components > genre-loader.tsx > ...

You, 2 weeks ago | 1 author (You)

```tsx
import { prisma } from '@/lib/db'
import { GenreSelector } from '@/components/genre-selector'

export async function GenreLoader() {
  const genres = await prisma.genre.findMany({
    orderBy: { name: 'asc' },
  })

  return <GenreSelector genres={genres} />
}
```

Satisfies operator

Satisfies operator

- The satisfies operator lets us **validate that the type of an expression matches some type**, without changing the resulting type of that expression.
- 💡 The goal is to **report type error where they are caused** 💡

movie-card.tsx

TS movie-card.tsx M ✕ TS page.tsx M

src > components > TS movie-card.tsx > MovieCard

```tsx
33    return (
34      <Card className="flex flex-col">
35        <CardHeader>
36          <CardTitle>{movie.title}</CardTitle>
37          <CardDescription>
38            Vote average: {movie.vote_average} (
39            {movie.vote_count.toLocaleString()} votes)
40          </CardDescription>
41        </CardHeader>
```

movies/page.tsx

TS movie-card.tsx M TS page.tsx M

src > app > movies > TS page.tsx > getMovies > genre

```tsx
async function getMovies(
  page: number,
  genreId?: string,
): Promise<MovieRequiredForCard[]> {
  const select = {
    id: true,
    title: true,
    overview: true,
    backdrop_path: true,
    vote_average: true,
    vote_count: true,
  } satisfies Prisma.MovieSelect
```

Prevent Overfetching

Prevent Overfetching

- Using **satisfies Prisma.MovieSelect** is not optimal
- **Allows Overfetching** of movie data
  - Making the application slightly slower than required
- This **can be prevented** with a type mapping
  - Create a **MovieSelect** like object with just the required keys

movies/page.tsx

TS page.tsx M ×

src > app > movies > TS page.tsx > getMovies > genre > include

```tsx
type ExactMovieSelect = Required<
  Pick<Prisma.MovieSelect, keyof MovieRequiredForCard>
>

async function getMovies(
  page: number,
  genreId?: string,
): Promise<MovieRequiredForCard[]> {
  const select = {
    id: true,
    title: true,
    overview: true,
    backdrop_path: true,
    vote_average: true,
    vote_count: true,
  } satisfies ExactMovieSelect
```

movies/page.tsx

page.tsx M

src > app > movies > page.tsx > getMovies > genre > include

```tsx
type PrismaSelect<TRow> = {
  [Prop in keyof TRow]: true
}

type ExactMovieSelect = PrismaSelect<MovieRequiredForCard>

async function getMovies(
  page: number,
  genreId?: string,
): Promise<MovieRequiredForCard[]> {
  const select = {
    id: true,
    title: true,
    overview: true,
    backdrop_path: true,
    vote_average: true,
    vote_count: true,
  } satisfies ExactMovieSelect
```

Break time

Photo by Mindspace Studio on Unsplash

Custom type mapping

Custom type mapping

- The TypeScript type system itself is a **programming language**
  - With variables, loops, conditional logic etc.
- Example: turn Zod schema into it's TypeScript type

type-mappng.ts M

src > lib > type-mappng.ts > ...

```ts
import { User } from 'lucide-react'
import { z } from 'zod'

const userSchema = z.object({
  firstName: z.string(),
  lastName: z.string(),
  age: z.number().optional(),
})

// extract the inferred type
type User = z.infer<typeof userSchema>

const user: User = {
  firstName: 'Maurice',
  lastName: 'de Beijer',
}
```

Type mapping Boolean logic

type-mappng.ts 1, M

src > lib > type-mappng.ts > copyOfMovie

```ts
import { Movie } from '@prisma/client'

type AllProps<TObject> = {
  [Prop in keyof TObject]: TObject[Prop]
}

type OnlyPropsOfTypeString<TObject> = {
  [Prop in keyof TObject as TObject[Prop] extends string
    ? Prop
    : never]: TObject[Prop]
}

type AllPropsOfAnObject<TObject extends Record<keyof any, unknown>> = {
  [Prop in keyof TObject]: TObject[Prop]
}

type CopyOfMovie = AllProps<Movie>
type StringPartsOfMovie = OnlyPropsOfTypeString<Movie>
type AllPropsOfMovie = AllPropsOfAnObject<Movie>
type AllPropsOfString = AllPropsOfAnObject<string>
```

type-mapping.ts

type-mappng.ts M

src > lib > type-mappng.ts > ...

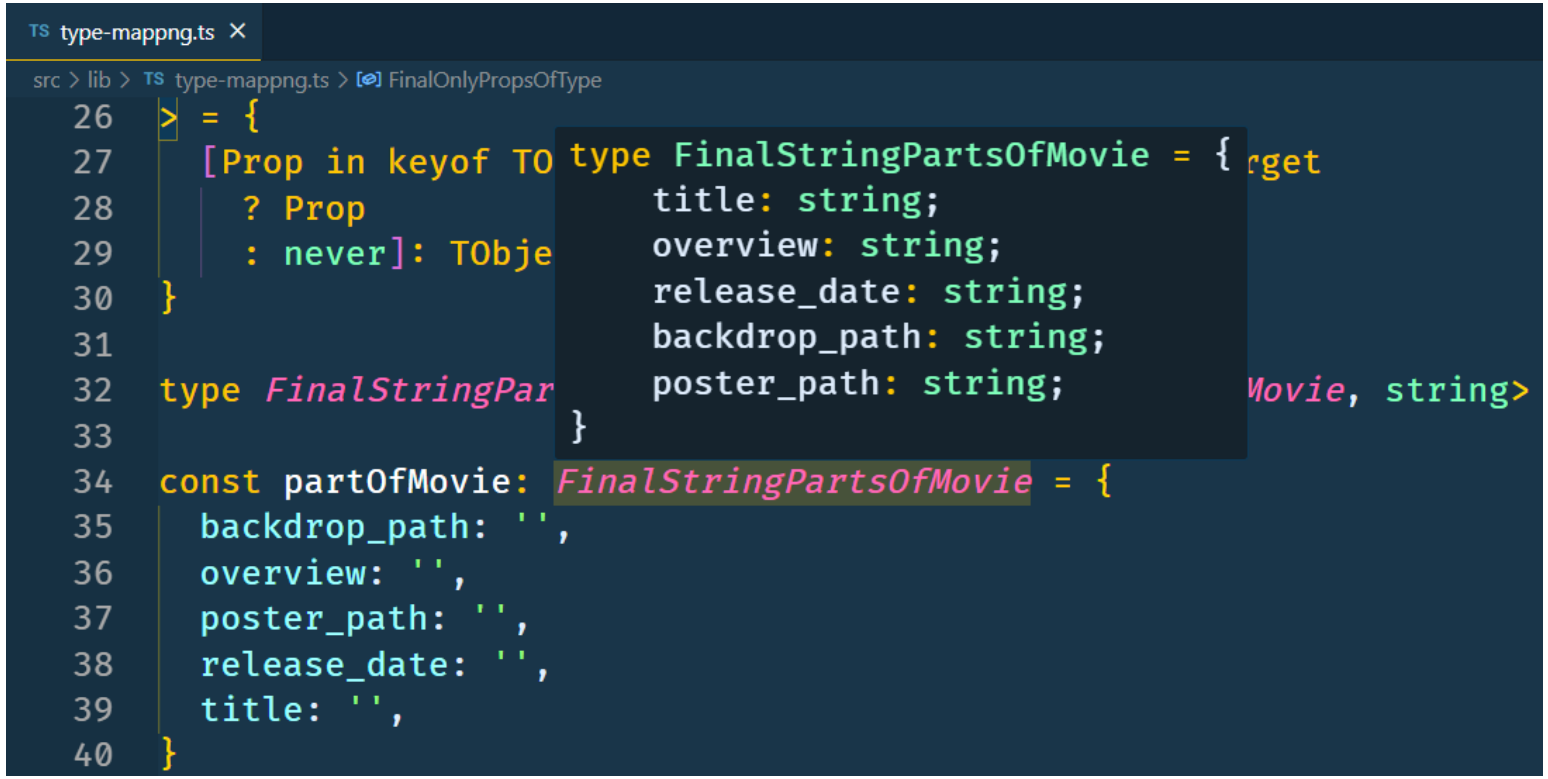
```ts
type FinalOnlyPropsOfType<
  TObject extends Record<keyof any, unknown>,
  Target,
> = {
  [Prop in keyof TObject as TObject[Prop] extends Target
    ? Prop
    : never]: TObject[Prop]
}

type FinalStringPartsOfMovie = FinalOnlyPropsOfType<Movie, string>

const partOfMovie: FinalStringPartsOfMovie = {
  backdrop_path: '',
  overview: '',
  poster_path: '',
  release_date: '',
  title: '',
}
```

The display of types

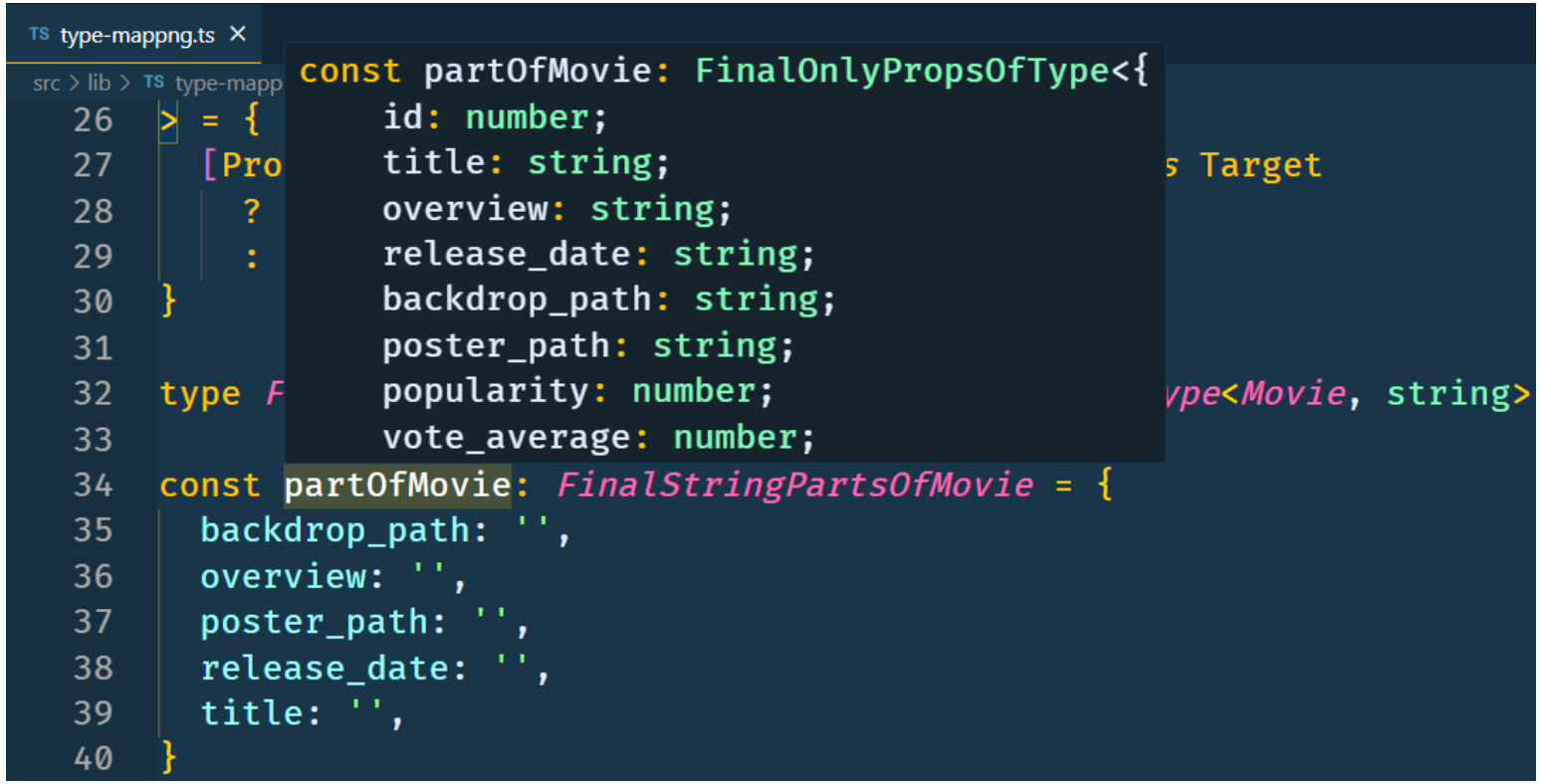
The display of types

- Type mappings sometimes lead to **hard to read types**
- **Easy to fix** with another type mapping

type-mapping.ts



type-mapping.ts



type-mapping.ts

type-mappng.ts M

src > lib > type-mappng.ts > ...

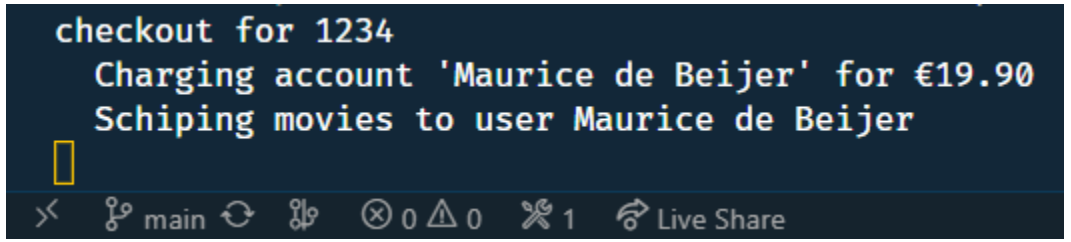
```ts
// Taken from https://effectivetypescript.com/2022/02/25/gentips-4-display/
type Resolve<T> = T extends Function ? T : { [K in keyof T]: T[K] }

type FinalStringPartsOfMovie = Resolve<FinalOnlyPropsOfType<Movie, string>>


        const partOfMovie: {
            title: string;
            overview: string;
            release_date: string;
            backdrop_path: string;
            poster_path: string;
        }
const partOfMovie: FinalStringPartsOfMovie = {
  backdrop_path: '',
  overview: '',
  poster_path: '',
  release_date: '',
  title: '',
}
```

Opaque Types

Opaque Types

- A lot of business data ultimately end up as a **primitive data type**
  - They are all modeled as string, number etc.
- The **compiler doesn't know the difference** between them
  - A PO box number and invoice total amount are both type "number"
    - The same for the compiler
    - Very different for the business case
- **Opaque types can make it easier to reason about code**
  - By providing distinct types and a clear separation

Console output



checkout-shopping-cart.ts

TS checkout-shopping-cart.ts M × TS checkout-dialog.tsx M

src > server > TS checkout-shopping-cart.ts > ...

```ts
declare const _type: unique symbol

type Opaque<A, B> = A & {
  readonly [_type]: B
}

type Account = Opaque<string, 'Account'>

function isAccount(value: string): value is Account {
  return typeof value === 'string' && value.length === 4
}

function assertAccount(value: string): asserts value is Account {
  if (!isAccount(value)) {
    throw new Error(`Expected account with 4 digits, got ${value}`)
  }
}

type Amount = Opaque<number, 'Amount'>
```

checkout-shopping-cart.ts

TS checkout-shopping-cart.ts M × TS checkout-dialog.tsx M

src > server > TS checkout-shopping-cart.ts > ...

```ts
export async function checkoutShoppingCart(
  account: string,
  name: string,
  amount: number,
) {
  assertAccount(account)
  assertClientName(name)
  assertAmount(amount)

  console.group(`checkout for ${name}`)
  chargeAccount(account, amount)
  schipMovies(name)
  console.groupEnd()
}

function chargeAccount(account: Account, amount: Amount) {
  console.log(
    `Charging account '${account}' for ${amount.toLocaleString('en', {
      style: 'currency',
      currency: 'EUR',
    })}`,
  )
}
```

checkout-dialog.tsx

TS checkout-shopping-cart.ts M | TS checkout-dialog.tsx M

src > components > TS checkout-dialog.tsx > CheckoutDialog

```tsx
const onSubmit = async (data: CheckoutForm) => {
  try {
    await checkoutShoppingCart(data.account, data.name, totalAmount)
    toast({
      title: 'Success',
      description: 'Checkout completed',
    })
    setCheckoutOpen(false)
    clearCart()
  } catch (error) {
    const description =
      error instanceof Error ? error.message : 'Something went wrong'
    toast({
      title: 'Oops',
      description,
      variant: 'destructive',
    })
  }
}
```

More strict features

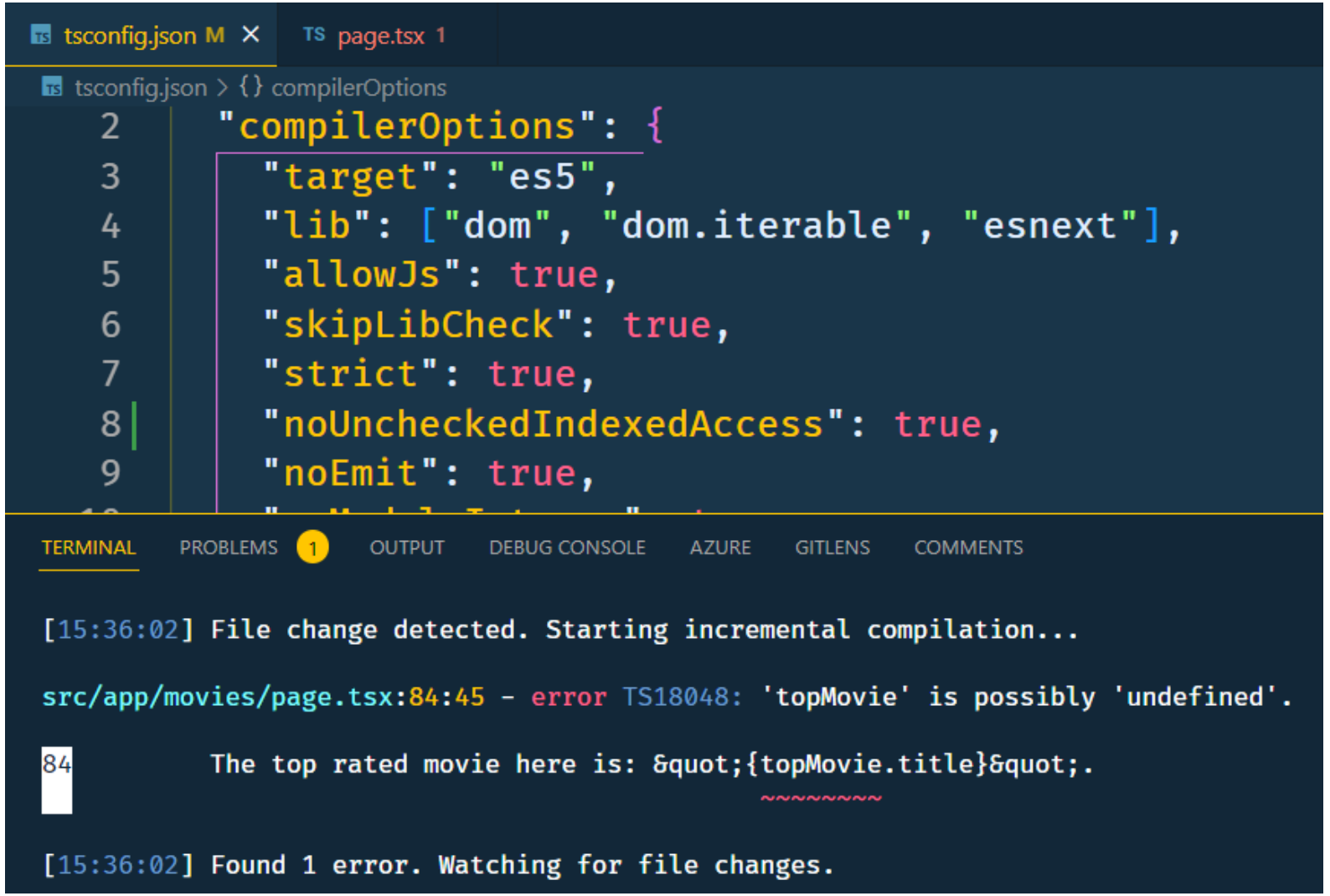
More Strict Features

- There are **many more strict settings** not enabled by "strict"
  - allowUnreachableCode
  - allowUnusedLabels
  - exactOptionalPropertyTypes
  - noFallthroughCasesInSwitch
  - noImplicitOverride
  - noImplicitReturns
  - noPropertyAccessFromIndexSignature
  - noUncheckedIndexedAccess
  - noUnusedLocals
  - noUnusedParameters

noUnchecked IndexedAccess

- By default **every** index from an array is seen as the **array element type**
  - Even if it exceeds the items available and will result in *undefined*
- Enabling "***noUncheckedIndexedAccess"*** requires you to check the element before using
  - Whether the element is the *array element type* or *undefined*
- Try showing the **Horror** movies and observe a runtime error ☹

tsconfig.json



movies/page.tsx

tsconfig.json M page.tsx M

src > app > movies > page.tsx > MoviesPage

```tsx
return (
  <main className="flex-1 space-y-4 p-8 pt-6">
    <h2 className="text-3xl font-bold tracking-tight">Movies</h2>
    {topMovie ? (
      <p className="p-6 text-center font-bold">
        The top rated movie here is: &amp;quot;{topMovie.title}&amp;quot;.
      </p>
    ) : (
      <p className="p-6 text-center font-bold">
        There are no movies in this genre.
      </p>
    )}
```

Using Readonly<>

Readonly<T>

- The *Readonly<T>* mapped type **creates a read-only mapped type**
  - Can't change properties anymore
  - Or use "array.push()" etc.
- ⚠ *Readonly<T>* is **not recursive** ⚠
  - Only the first level of properties becomes read-only
- There is also a *ReadonlyArray<T>* mapped type
- 💡 Recommended for **function arguments** to show intent 💡
  - And AJAX responses etc.

movie-card.tsx

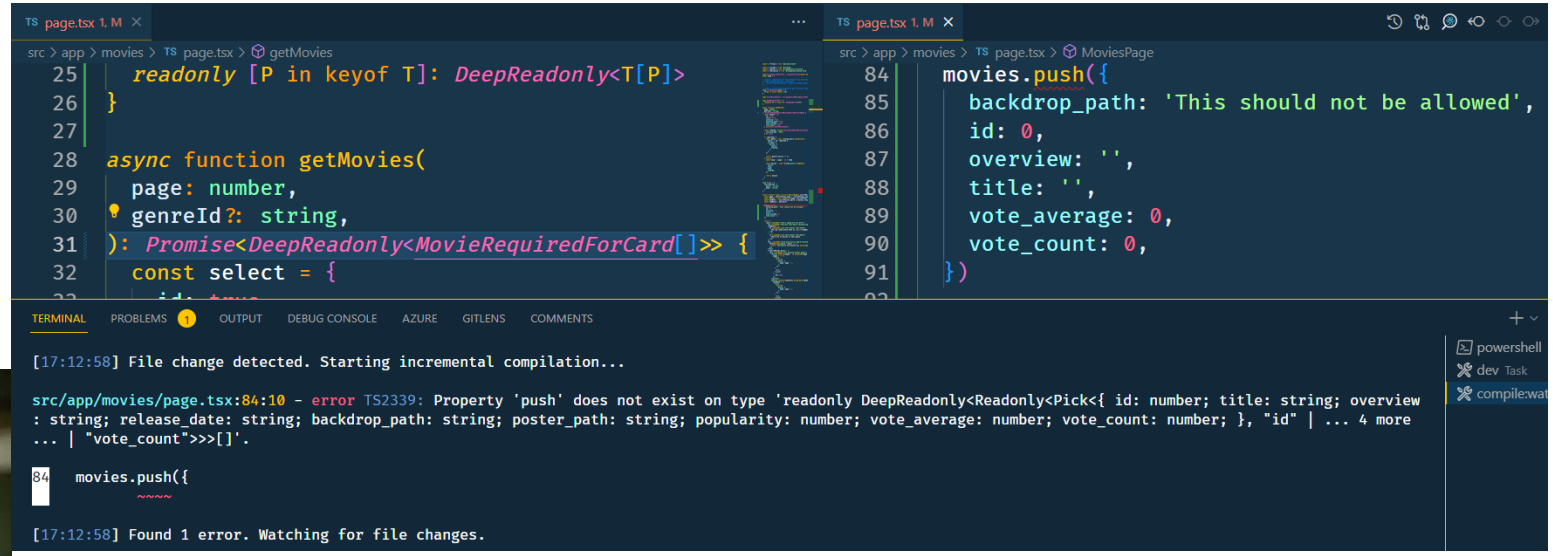

```
TS movie-card.tsx 1, M
src > components > TS movie-card.tsx > MovieCard
18  type Props = {
19    movie: Readonly<
20      Pick<
21        Movie,
22        | 'id'
23        | 'title'
24        | 'overview'
25        | 'backdrop_path'
26        | 'vote_average'
27        | 'vote_count'
28      >
29    >
30  }
31
32  export const MovieCard = ({ movie }: Props) => {
33    const { addMovie } = useShoppingCart()
34
35    movie.title = 'This should not be allowed'
36

TERMINAL  PROBLEMS 1  OUTPUT  DEBUG CONSOLE  AZURE  GITLENS  COMMENTS

[16:58:53] File change detected. Starting incremental compilation...

src/components/movie-card.tsx:35:9 - error TS2540: Cannot assign to 'title' beca

35    movie.title = 'This should not be allowed'
          ~~~~~

[16:58:53] Found 1 error. Watching for file changes.
```

Custom type mapping DeepReadonly<>

DeepReadonly<T>

- Make a whole **nested object structure read-only**
  - Recursive mapped types are very powerful
  - An improvement over the default "*Readonly<T>*"
- Source: https://gist.github.com/basarat/1c2923f91643a16a90de638e76bce0ab

movies/page.tsx



Template Literal Types

Template Literal Types

- Builds on string literal types but **manipulates type definitions**
  - For example change a types snake case to camel case key names

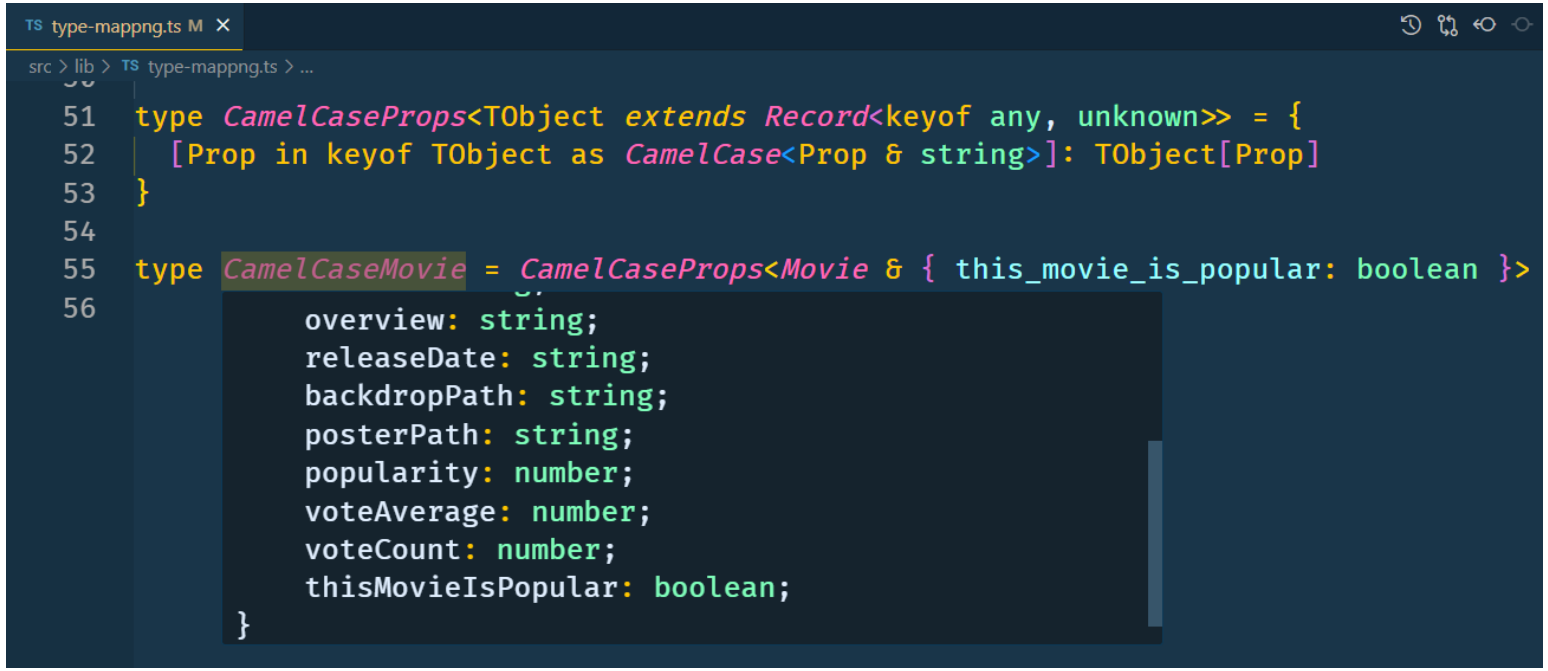
type-mapping.ts

TS type-mappng.ts M

src > lib > TS type-mappng.ts > ...

```ts
type CamelCase<S extends string> = S extends `${infer First}_${infer Rest}`
  ? `${Lowercase<First>}${Capitalize<CamelCase<Rest>>}`
  : Lowercase<S>

type CamelCaseProps<TObject extends Record<keyof any, unknown>> = {
  [Prop in keyof TObject as CamelCase<Prop & string>]: TObject[Prop]
}

type CamelCaseMovie = CamelCaseProps<Movie & { this_movie_is_popular: boolean }>
```

type-mapping.ts



type-mapping.ts

TS type-mappng.ts M

src > lib > TS type-mappng.ts > ...

```
57  type Names = 'foo' | 'bar'
58  type Suffixes = 'Key' | 'Value'
59
60  type Pairs = `${Names}${Suffixes}`
61
```

'Pairs' is declared but never used. ts(6196)

⚠ Error(TS6196)

`Pairs` is declared but never used.

```
type Pairs = "fooKey" | "fooValue" | "barKey" | "barValue"
```

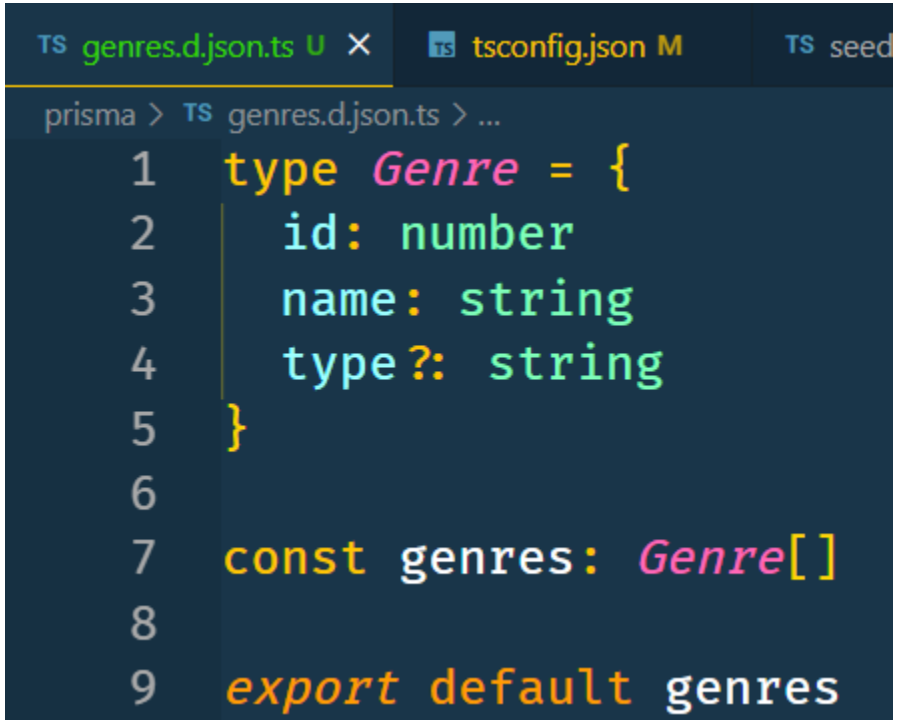
Quick Fix... (Ctrl+.)

Typing JSON files

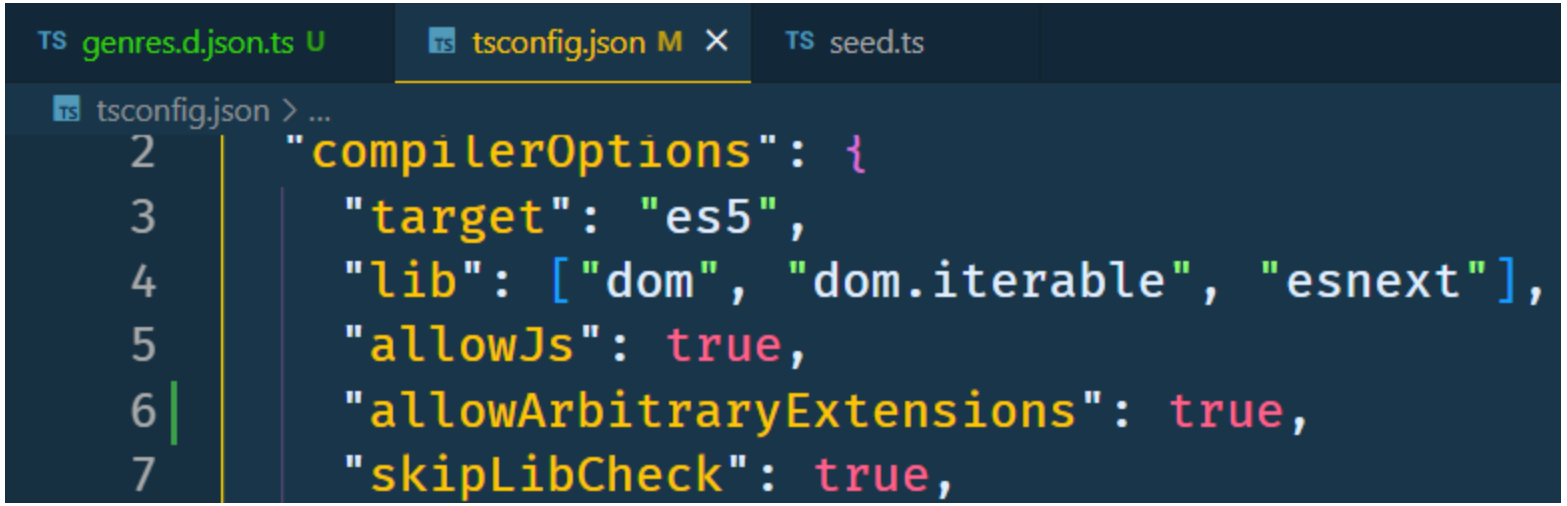
Typing JSON files

- **IntelliSense will parse an imported JSON file** to determine the types
  - This can be very slow with large files
- Add a ***<filename>.d.json.ts*** to explicitly type the imported data
  - ⚠ Will not validate that the JSON actually has the correct shape ⚠

genres.d.json.ts



tsconfig.json



seed.ts


```
TS genres.d.json.ts U   tsconfig.json M   TS seed.ts ×
prisma > TS seed.ts > main
1  import { PrismaClient } from '@prisma/client'
2
3  const p (alias) const genres: Genre[]
4  import  import genres
5  import genres from './genres.json'
```

Conclusion

- **Compiling** the code to catch type errors early
  - Compile and Test your code with each pull request
- Using the ***satisfies*** operator
- Use **type mappings** to optimize your code
  - Prevent over fetching of data with Prisma
- **Custom type mappings** are really powerful
  - Can help detect all sorts of errors quickly
- Make mapped types more **readable**
  - Using another type mapping ☺
- **Opaque Types** can help with type safety
  - Not all strings represent the same data but for TypeScript a string is a string
- Use **more strict features** beyond the basics
  - Using noUncheckedIndexedAccess can help catch potential runtime errors
- Using **Readonly<>** and **ReadonlyArray<>**
  - Prevent objects from being mutated by accident
  - Custom type mapping DeepReadonly<>
- Use **Template Literal Types** to manipulate types
- **Typing JSON files** can help performance when importing large JSON files

Maurice de Beijer

@mauricedb

maurice.de.beijer@gmail.com