

# Best practices & advanced TypeScript tips for React developers

Maurice de Beijer  
@mauricedb



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



# Topics

- Writing **React components** using TypeScript
  - Mutually exclusive component props
  - Generic React component prop types
  - Deriving React component prop types
  - Inferring TypeScript types
- TypeScript **stricter** features
  - Beyond strict using noUncheckedIndexedAccess
- **Validating data at the boundary** using Zod
  - Inferring TypeScript types from Zod schemas
- **Type mapping**
  - With Omit<>, Pick<> and Readonly<>
  - Custom type mapping
- **Runtime and compile type safety**
  - Type predicate and assertion functions
  - Exhaustiveness checking

Type it out  
by hand?

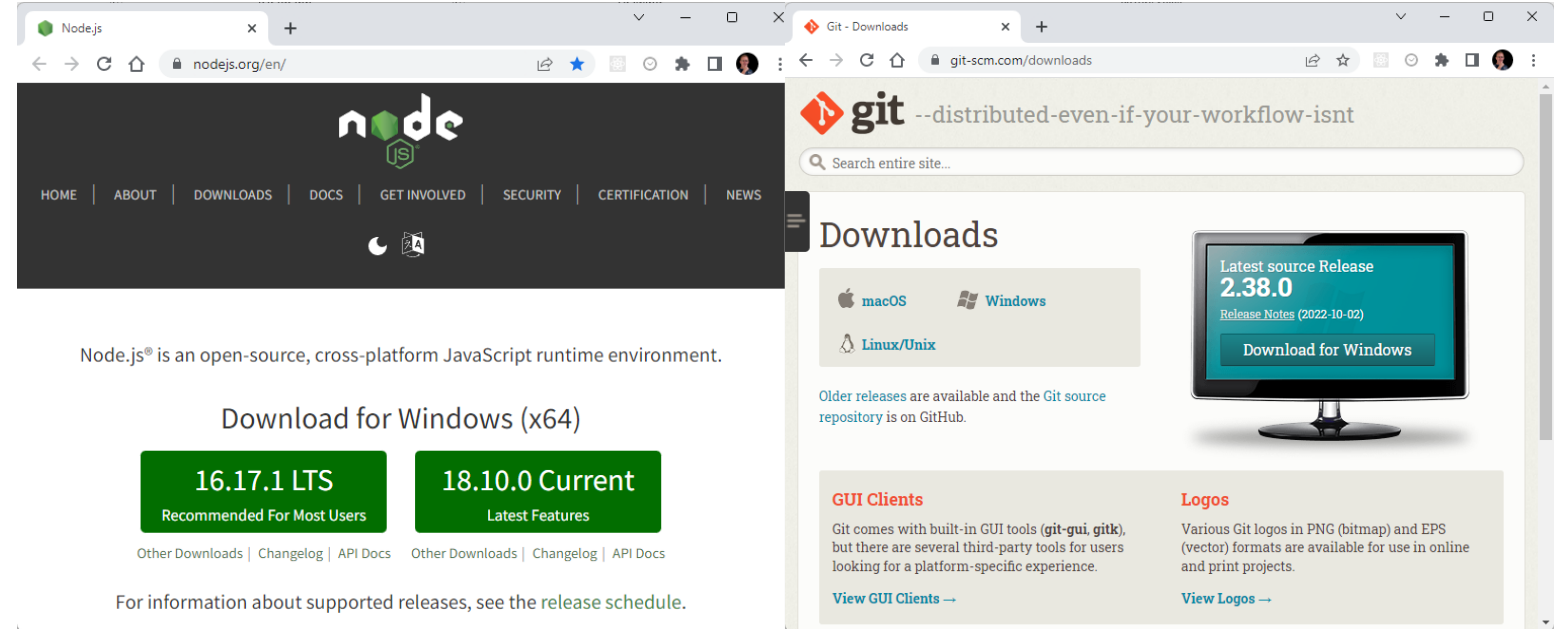
*"Typing it drills it into your brain much better than simply copying and pasting it. You're forming new neuron pathways. Those pathways are going to help you in the future. Help them out now!"*

# Prerequisites

Install Node & NPM

Install the GitHub repository

# Install Node.js & NPM



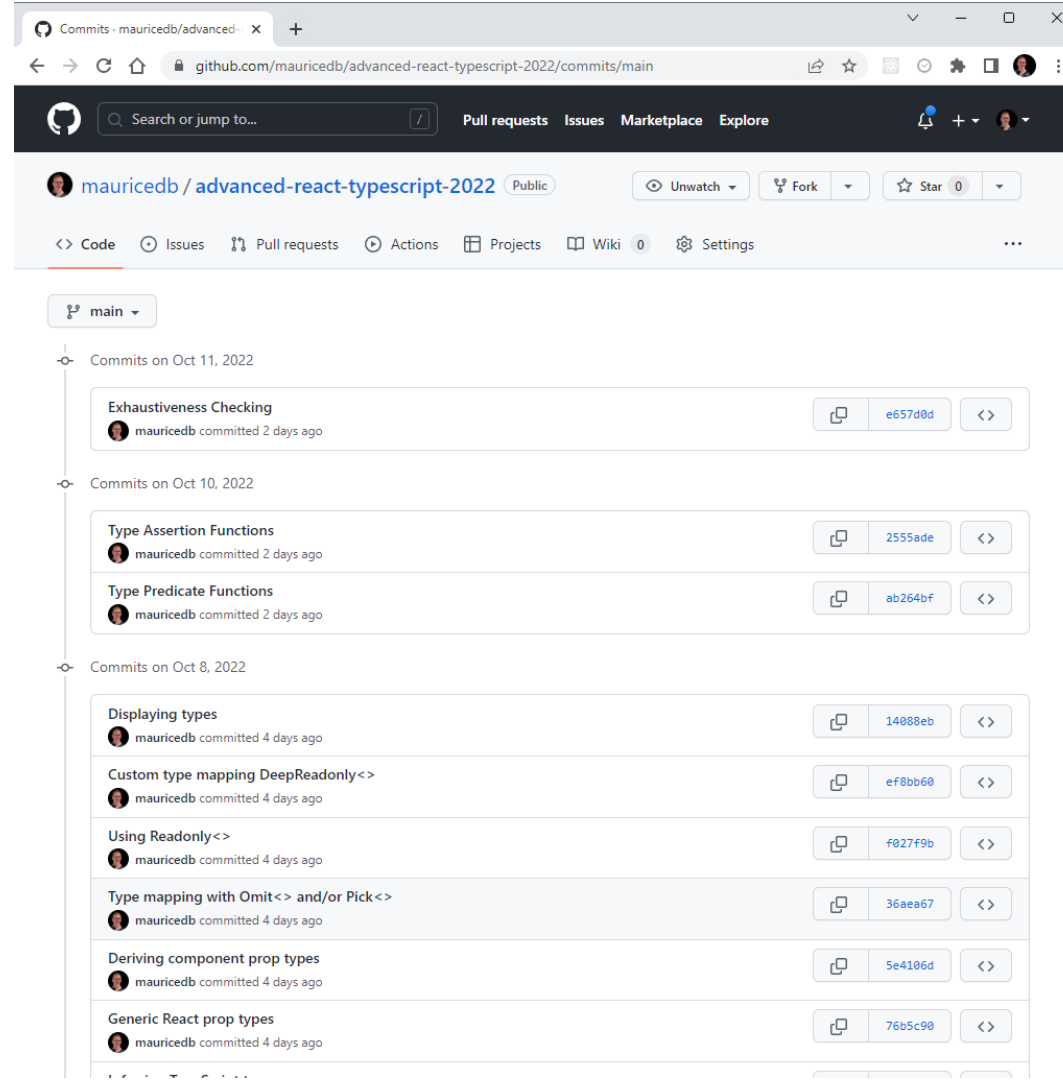
```
PS C:\Repos> node --version
v16.17.1
PS C:\Repos> git --version
git version 2.38.0.windows.1
PS C:\Repos> npm --version
8.19.2
```

# Following Along

```
TS alert.tsx x
src > 01-js-to-ts > TS alert.tsx >
You, 15 minutes ago | 1 author (You)
1 import { FC } from 'react';
2 import { useIntl } from 'react-intl';
3
4 type Props = {
5   messageId: string;
6   variant: 'primary' | 'secondary' | 'success' | 'danger';
7 };
8
9 export const Alert: FC<Props> = ({ messageId, variant }) => {
10   const { formatMessage } = useIntl();
11
12   if (!messageId) {
13     throw new Error('The messageId prop is required');
14   }
15
16   return (
17     <div className={`alert alert-${variant}`} role="alert">
18       {formatMessage({ id: messageId })}
19     </div>
20   );
21 };
```

- Repo: <https://github.com/mauricedb/advanced-react-typescript-2022>
- Slides: <https://bit.ly/react-ts-2022>

# The changes





# Clone the GitHub Repository

```
PS C:\Temp> git clone git@github.com:mauricedb/advanced-react-typescript-2022.git
Cloning into 'advanced-react-typescript-2022'...
remote: Enumerating objects: 145, done.
remote: Counting objects: 100% (145/145), done.
remote: Compressing objects: 100% (92/92), done.
Receiving objects: 76% (111/145), 9.28 MiB | 4.60 MiB/sremote: Total 145 (delta 53), reused 138 (delta 46), pack-reused 0
Receiving objects: 100% (145/145), 9.82 MiB | 4.65 MiB/s, done.
Resolving deltas: 100% (53/53), done.
PS C:\Temp> |
```

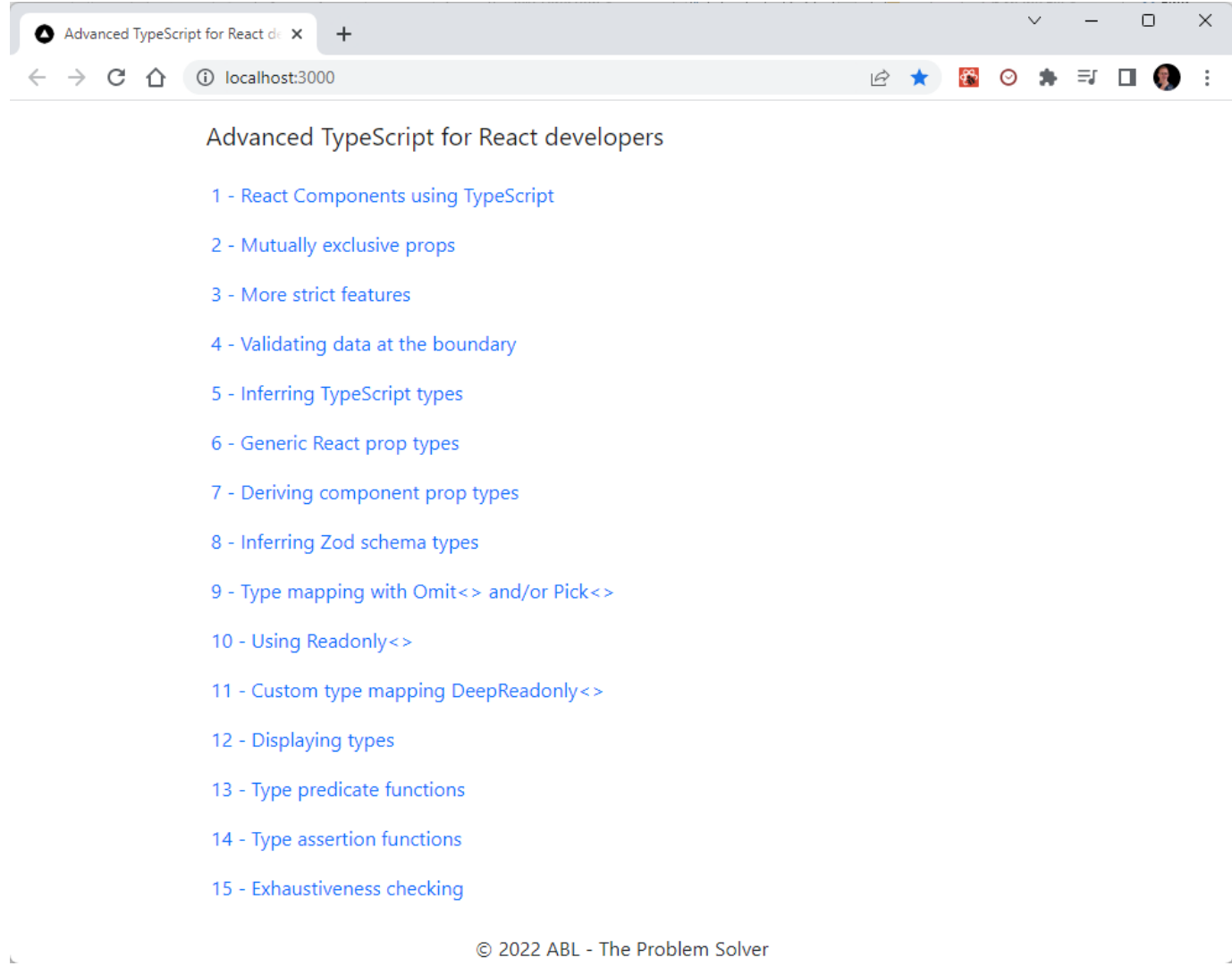
# Install NPM Packages

```
PS C:\Temp> cd .\advanced-react-typescript-2022\  
PS C:\Temp\advanced-react-typescript-2022> npm install  
  
added 251 packages, and audited 252 packages in 5s  
  
81 packages are looking for funding  
  run `npm fund` for details  
  
found 0 vulnerabilities  
PS C:\Temp\advanced-react-typescript-2022> |
```

# Start the application

```
PS C:\Temp\advanced-react-typescript-2022> npm run dev  
  
> advanced-react-typescript-2022@0.1.0 dev  
> next dev  
  
ready - started server on 0.0.0.0:3000, url: http://localhost:3000  
event - compiled client and server successfully in 543 ms (335 modules)
```

# The application



# Compiling the code

# Compiling the code

- Quite often TypeScript code is not type checked during development
  - Create React App use Babel
  - Next.js uses SWC
- 👉 Start with the **00-start** branch 👉
  - `git checkout --track origin/00-start`

# package.json

```
{ } package.json M x TS index.tsx 1, M
{ } package.json > ...
5   "scripts": {
6     "dev": "next dev",
7     "compile": "tsc --noEmit",
8     "compile:watch": "tsc --noEmit --watch",
9     "build": "next build",
10    "start": "next start",
11    "lint": "next lint"
12  },
```

npm run compile




```
TERMINAL  PROBLEMS  OUTPUT  DEBUG CONSOLE  AZURE  GITLENS

[14:12:26] File change detected. Starting incremental compilation...

pages/index.tsx:84:7 - error TS2322: Type 'number' is not assignable to type 'string'.
84  const notString: string = 1;
    ~~~~~

[14:12:26] Found 1 error. Watching for file changes.
```





# React Components using TypeScript

# Components and TypeScript

- React components can be **written in different ways**
  - Named functions or arrow functions
  - Just like with ECMAScript
- Create a **type to describe** the component **Props**
  - Either an interface or a type alias
- Annotate the result as a **valid React type**
  - Or let TypeScript infer the resulting type
- Typing with an arrow function is often easier with **React.FC<TProp>**
  - But doesn't work well with generic components

# alert.tsx



```
TS 01-js-to-ts.tsx  TS alert.tsx  X
src > 01-js-to-ts > TS alert.tsx > ...

You, 15 minutes ago | 1 author (You)
1 import { FC } from 'react';           You, 15 minutes ago • React
2 import { useIntl } from 'react-intl';
3
4 type Props = {
5   messageId: string;
6   variant: 'primary' | 'secondary' | 'success' | 'danger';
7 };
8
9 export const Alert: FC<Props> = ({ messageId, variant }) => {
10   const { formatMessage } = useIntl();
11
12   if (!messageId) {
13     throw new Error('The messageId prop is required');
14   }
15
16   return (
17     <div className={`alert alert-${variant}`} role="alert">
18       {formatMessage({ id: messageId })}
19     </div>
20   );
21 };
```



# Mutually exclusive props

# Mutually exclusive props

- Sometimes **not all combinations of props** are allowed
  - Two props might be mutually exclusive.
  - You must to pass one of them but not both
- Use an ***or*** between **multiple prop types**
  - With an optional *never* to prevent illegal combinations



```
TS 02-mutually-exclusive.tsx 1 TS dual-alert.tsx M X
src > 02-mutually-exclusive > TS dual-alert.tsx > ...
14 type Props = (
15   | {
16     message: string;
17     messageId?: never;
18   }
19   | {
20     message?: never;
21     messageId: string;
22   }
23 ) & {
24   variant?: Variant;
25 };
26
```

TERMINAL PROBLEMS 1 OUTPUT DEBUG CONSOLE AZURE GITLENS

[14:42:13] File change detected. Starting incremental compilation...

pages/02-mutually-exclusive.tsx:10:8 - error TS2322: Type '{ message: string; messageId: string; }' is not assignable to type 'IntrinsicAttributes & Props'.  
Type '{ message: string; messageId: string; }' is not assignable to type '{ message?: undefined; messageId: string; }'.  
Types of property 'message' are incompatible.  
Type 'string' is not assignable to type 'undefined'.

10 <DualAlert

# More strict features

# More Strict Features

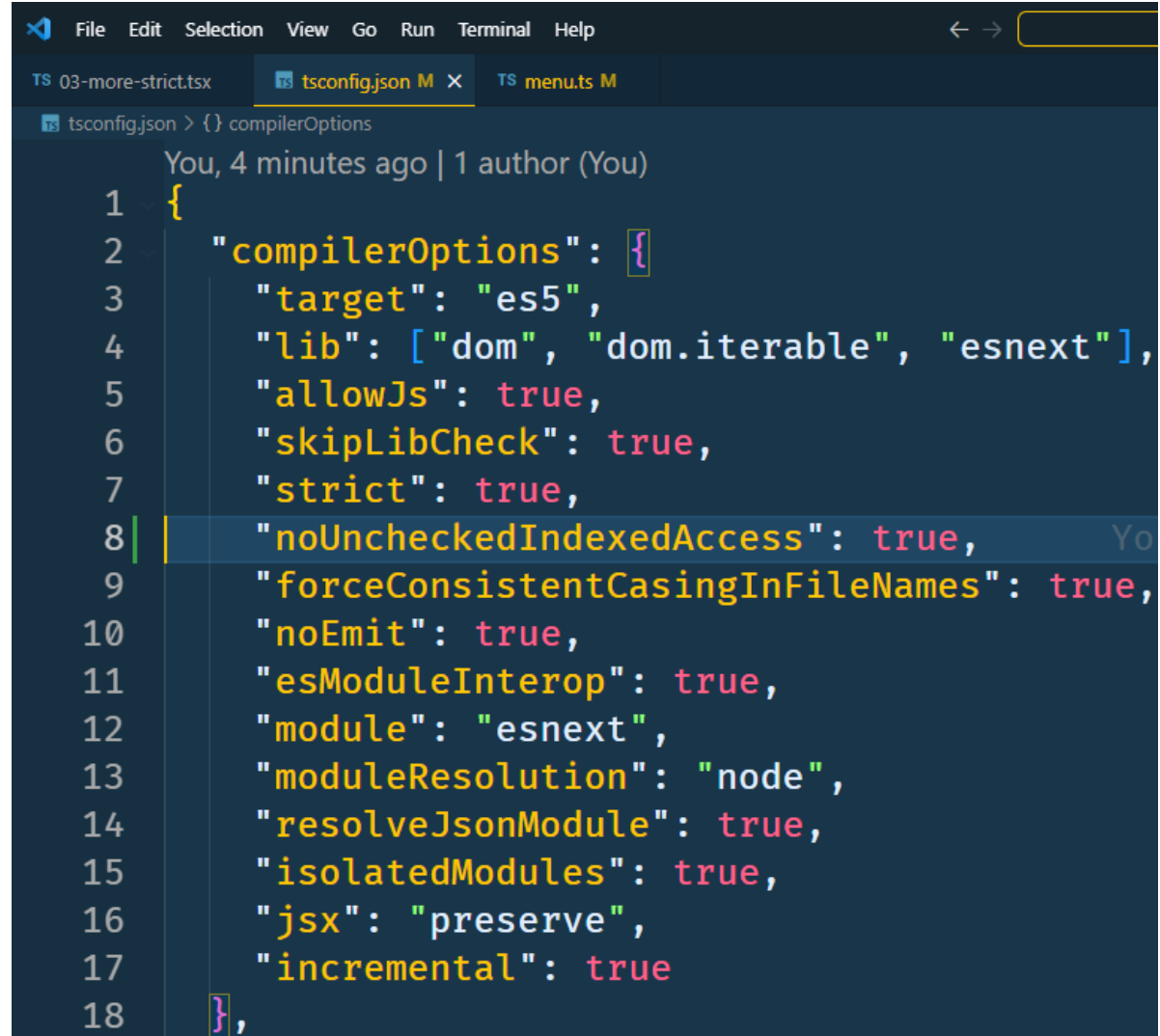
- There are **many more strict settings** not enabled by “strict”
  - allowUnreachableCode
  - allowUnusedLabels
  - exactOptionalPropertyTypes
  - noFallthroughCasesInSwitch
  - noImplicitOverride
  - noImplicitReturns
  - noPropertyAccessFromIndexSignature
  - noUncheckedIndexedAccess
  - noUnusedLocals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



# noUncheckedIndexedAccess

- 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
  - Or element are their type or *undefined*
- Try adding Mushrooms to the Pizza Ai Funghi and observe a runtime error 😞

# tsconfig.json




```
File Edit Selection View Go Run Terminal Help
TS 03-more-strict.tsx tsconfig.json M TS menu.ts M
tsconfig.json > {} compilerOptions
You, 4 minutes ago | 1 author (You)
1 {
2   "compilerOptions": {
3     "target": "es5",
4     "lib": ["dom", "dom.iterable", "esnext"],
5     "allowJs": true,
6     "skipLibCheck": true,
7     "strict": true,
8     "noUncheckedIndexedAccess": true,
9     "forceConsistentCasingInFileNames": true,
10    "noEmit": true,
11    "esModuleInterop": true,
12    "module": "esnext",
13    "moduleResolution": "node",
14    "resolveJsonModule": true,
15    "isolatedModules": true,
16    "jsx": "preserve",
17    "incremental": true
18  },
```

# menu.ts



```
File Edit Selection View Go Run Terminal Help
TS 03-more-strict.tsx TSconfig.json M TS menu.ts M x
src > 03-more-strict > TS menu.ts > ...
• 55 export const getExtraIngredient = (name: string): ExtraIngredient => {
56     const extraIngredient = extraIngredients[name] ?? { name, price: 0 };
57
58     return extraIngredient;
59 };
```



# Validating data at the boundary

# Validating Data

- The type definitions are **used at compile time**
- They **might not match the runtime** behavior
  - Specially when doing AJAX requests or reading JSON files
- Try adding Mushrooms to the Pizza Ai Funghi and observe another runtime error 😞

# schemas.ts

```
TS 04-validating-ajax-data.tsx TS schemas.ts U X TS pizza-shop-data-loader.tsx M
src > 04-validating-ajax-data > TS schemas.ts > ...
1 import { z } from 'zod';
2
3 export const pizzaSchema = z.object({
4   name: z.string(),
5   ingredients: z.string().array(),
6   price: z.number(),
7   extras: z.string().array(),
8 });
9
10 export const pizzasSchema = z.array(pizzaSchema);
11
12 export const extraIngredientSchema = z.object({
13   name: z.string(),
14   price: z.number(),
15 });
16
17 export const extraIngredientsSchema = z.record(extraIngredientSchema);
```

# pizza-shop- data-loader.tsx



```
TS 04-validating-ajax-data.tsx TS schemas.ts U TS pizza-shop-data-loader.tsx M X
src > 04-validating-ajax-data > TS pizza-shop-data-loader.tsx > ...

 6 import { PizzaShop } from './pizza-shop';
 7 import { extraIngredientsSchema, pizzasSchema } from './schemas';
 8
 9 const server = 'http://localhost:3000';
10
11 export const PizzaShopDataLoader: FC = () => {
12   const { data: pizzas, error: pizzasError } = useSWR<Pizza[]>(
13     '/api/pizzas.json',
14     (resource, init) =>
15       fetch(`${server}${resource}`, init)
16         .then((res) => res.json())
17         .then(pizzasSchema.parse)
18   );
19
20   const { data: extraIngredients, error: extraIngredientsError } =
21     useSWR<ExtraIngredients>(
22       '/api/good-extra-ingredients.json',
23       (resource, init) =>
24         fetch(`${server}${resource}`, init)
25           .then((res) => res.json())
26           .then(extraIngredientsSchema.parse)
27     );

```

# Inferring TypeScript types



# Inferring TypeScript types

- In many cases **TypeScript can infer types** from existing objects
  - Not just the object shape but also valid keys
- Use the ***extends* keyword** to limit a generic type argument
  - One generic argument can be used to infer a second etc.

# configuration.tsx

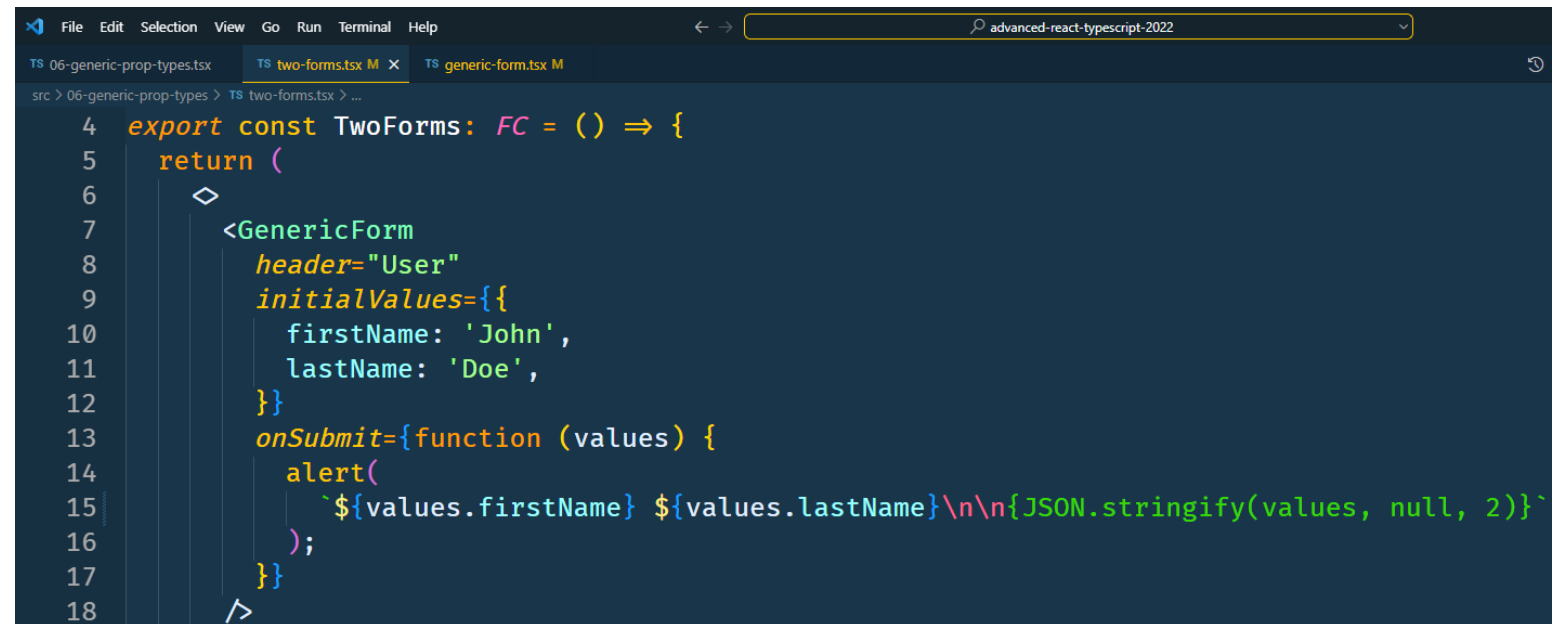


```
File Edit Selection View Go Run Terminal Help
TS 05-inferring-types.tsx TS configuration.tsx M X
src > 05-inferring-types > TS configuration.tsx > ...
4 function getConfigItem<
5   TSection extends keyof typeof config,
6   TItem extends keyof typeof config[TSection]
7 >(section: TSection, item: TItem) {
8   const config = {
9     user: {
10       firstName: 'John',
11       lastName: 'Doe',
12       birthDate: new Date(1990, 6, 10),
13     },
14     address: {
15       street: 'Main St',
16       houseNumber: 123,
17       city: 'New York',
18     },
19   };
20
21   return config[section][item];
22 }
23
24 export const Configuration: FC = () => {
25   const firstName = getConfigItem('user', 'firstName');
26   const lastName = getConfigItem('user', 'lastName');
27   const birthDate = getConfigItem('user', 'birthDate').toLocaleDateString();
28
29   // const employer = getConfigItem('employer', 'name');
```

# Generic React prop types

# Generic React prop types

- React component **prop types** can also be **generic**
  - To ensure that various props have matching type definitions
- The generic type can be specified when the component is used
  - Or will automatically inferred if not
- Very powerful to create reusable, flexible but fully typed components



```
4 export const TwoForms: FC = () => {
5   return (
6     <
7       <GenericForm
8         header="User"
9         initialValues={{
10           firstName: 'John',
11           lastName: 'Doe',
12         }}
13         onSubmit={function (values) {
14           alert(
15             `${values.firstName} ${values.lastName}\n\n${JSON.stringify(values, null, 2)}`
16           );
17         }}
18     />
```



```
File Edit Selection View Go Run Terminal Help
TS 06-generic-prop-types.tsx TS two-forms.tsx M TS generic-form.tsx M X
src > 06-generic-prop-types > TS generic-form.tsx > ...

4 type Props<TValues> = {
5   header: string;
6   initialValues: TValues;
7   onSubmit: (values: TValues) => void;
8 };
9
10 export function GenericForm<TValues extends Record<string, any>>({
11   header,
12   initialValues,
13   onSubmit,
14 }): Props<TValues>) {
15   const [values, setValues] = useState(initialValues);
16
17   return (
18     <form className="mb-5" onSubmit={() => onSubmit(values)}>
19       <h3 className="mb-3">{header}</h3>
20
21       {Object.keys(values).map((key) => (
22         <LabeledInput
23           key={key}
24           value={values[key]}
25           onChange={(e) => setValues({ ...values, [key]: e.target.value })}
26         >
```



# Deriving component prop types

# Deriving component prop types

- **Infer a component Prop type**
  - Using *React.ComponentProps<typeof Component>*
- **No need to publicly export** all those prop definitions
  - Just in case they are needed
- Very useful when you want to export the **nested component props**
  - Use type mappings to modify the type as needed



# pizza-on-menu.tsx

```
TS 07-deriving-prop-types.tsx TS pizza-on-menu.tsx M X TS labeled-checkbox.tsx M
src > 07-deriving-prop-types > TS pizza-on-menu.tsx > ...
38 <div className="mb-3">
39   <h5 className="card-title">Extras</h5>
40   {pizza.extras.map((extra) => (
41     <LabeledCheckbox
42       key={extra}
43       checked={extras.includes(extra)}
44       disabled={extra === 'mushrooms'}
45       id={`${pizza.name}-${extra}`}
46       onChange={() => {
```

# labeled- checkbox.tsx



```
TS 07-deriving-prop-types.tsx TS pizza-on-menu.tsx M TS labeled-checkbox.tsx M X
src > components > TS labeled-checkbox.tsx > ...
You, 2 minutes ago | 1 author (You)
1 import { ComponentProps, FC, useId } from 'react';
2 import { Checkbox } from './checkbox';
3 import { Label } from './label';
4
5 type Props = ComponentProps<typeof Checkbox>;
6
7 export const LabeledCheckbox: FC<Props> = ({ children, id, ...props }) => {
8   const internalId = useId();
9
10  return (
11    <div className="form-check">
12      <Checkbox id={id ?? internalId} { ...props } />
13      <Label htmlFor={id ?? internalId}>{children}</Label>
14    </div>
15  );
16 };
```



# Inferring Zod schema types

# Inferring Types

- Maintaining a **Zod schema** and a **TypeScript interface** is tedious
  - Both have to be kept in sync
- The TypeScript types can be **inferred** from the Zod schema
  - Using “z.infer(typeof schema)”

types.ts



```
TS 08-inferring-schema-types TS schemas.ts TS types.ts M X
src > 08-inferring-schema-types > TS types.ts > ...
You, 1 minute ago | 1 author (You)
1 import { z } from 'zod';
2 import {
3   extraIngredientSchema,
4   extraIngredientsSchema,
5   pizzaSchema,
6 } from './schemas';
7
8 export type Pizza = z.infer<typeof pizzaSchema>;
9
10 export type ExtraIngredients = z.infer<typeof extraIngredientsSchema>;
11
12 export type ExtraIngredient = z.infer<typeof extraIngredientSchema>;
13
14 export type PizzaOnOrder = {
15   name: string;
16   price: number;
17   extraIngredients: ExtraIngredient[];
18 };
```

# Type mapping with Omit<> and/or Pick<>

# Type mapping with `Omit<>` and/or `Pick<>`

- **Type mapping** lets you **create a new type** based on an existing type
  - With one or more modifications
- There are many **build in type mappings**
  - **Omit**: Create a new type by removing one or more props
  - **Pick**: Create a new type with just the specified props
- A type mapping can contain **conditional logic** to alter a part of the type

# types.ts



```
TS 09-type-mapping.tsx TS types.ts M X
src > 09-type-mapping > TS types.ts > ...
8  export type Pizza = z.infer<typeof pizzaSchema>;
9  export type ExtraIngredients = z.infer<typeof extraIngredientsSchema>;
10 export type ExtraIngredient = z.infer<typeof extraIngredientSchema>;
11
12 export type PizzaOnOrder = Pick<Pizza, 'name' | 'price'> & {
13   extraIngredients: ExtraIngredient[];
14 };
```



# 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
- 💡 Recommended for **function arguments** to show intent 💡
  - And AJAX responses etc.

types.ts

```
TS 10-using-readonly.tsx TS types.ts M X TS pizza-shop.tsx M
src > 10-using-readonly > TS types.ts > [e] Pizza
10 export type ExtraIngredient = Readonly<{
11     name: string;
12     price: number;
13 }>;
```

# pizza-shop.tsx



```
TS 10-using-readonly.tsx  TS types.ts M  TS pizza-shop.tsx M X
src > 10-using-readonly > TS pizza-shop.tsx > ...
14   const onPlaceOrder = () => {
15     const extrasForAEuro = order.flatMap((pizza) =>
16       pizza.extraIngredients.filter((extra) => extra.price === 1)
17     );
18     console.log('Extras for a Euro', extrasForAEuro);
```



# 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/1c2923f91643a16agode638e76bceoab>

# types.ts

```
TS 11-using-deep-readonly.tsx TS types.ts M x TS menu.ts M TS pizza-shop.tsx M TS ordered-pizza.tsx M
src > 11-using-deep-readonly > TS types.ts > ...
You, 4 minutes ago | 1 author (You)
1 export type DeepReadonly<T> = {
2   readonly [P in keyof T]: DeepReadonly<T[P]>;
3 };
4
5 export type Pizza = {
6   name: string;
7   ingredients: string[];
8   price: number;
9   extras: string[];
10 };
11
12 export type ExtraIngredients = Record<string, ExtraIngredient>;
13
14 export type ExtraIngredient = {
15   name: string;
16   price: number;
17 };
```

# menu.ts

```
TS 11-using-deep-readonly.tsx TS types.ts M TS menu.ts M x TS pizza-shop.tsx M TS ordered-pizza.tsx M
src > 11-using-deep-readonly > TS menu.ts > ...
8 export const pizzas: DeepReadonly<Pizza[]> = [
9   {
10     name: 'Pizza Margherita',
11     ingredients: ['tomato sauce', 'mozzarella', 'basil'],
12     price: 7.95,
13     extras: ['olives'],
14   },
```



# pizza-shop.tsx

```
TS 11-using-deep-readonly.tsx TS types.ts M TS menu.ts M TS pizza-shop.tsx M X TS ordered-pizza.tsx M
src > 11-using-deep-readonly > TS pizza-shop.tsx > ...
 9 export const PizzaShop: FC = () => {
10   const { formatNumber } = useIntl();
11   const [order, setOrder] = useState<DeepReadonly<PizzaOnOrder[]>>([]);
12   const totalPrice = useMemo(() => calculateTotalPrice(order), [order]);
13
14   const onPlaceOrder = () => {
15     const extrasForAEuro = order.flatMap((pizza) =>
16       pizza.extraIngredients.filter((extra) => extra.price === 1)
17     );
18     console.log('Extras for a Euro', extrasForAEuro);
19
20     // pizzas.push({
21     //   name: `New Pizza ${new Date().toLocaleTimeString()}`,
22     //   price: 10,
23     //   extras: ['cheese'],
24     //   ingredients: ['tomato sauce'],
25     // });
26
27     // if (pizzas[0]) {
28     //   pizzas[0].price *= 10;
29     // }
```

ordered-pizza.tsx



```
TS 11-using-deep-readonly.tsx TS types.ts M TS menu.ts M TS pizza-shop.tsx M TS ordered-pizza.tsx M X
src > 11-using-deep-readonly > TS ordered-pizza.tsx > ...
6 type Props = {
7   pizza: DeepReadonly<PizzaOnOrder>;
8 };
9
10 export const OrderedPizza: FC<Props> = ({ pizza }) => {
```



# Displaying types

# Displaying Types

- A **disadvantage of mapped types** is that the type definition in tooltips becomes **hard to read**
  - It shows how a type is constructed instead of the resulting type
- The `Resolve<T>` turns this into **the resulting type** instead
  - Source:  
<https://effectivetypescript.com/2022/02/25/gentips-4-display/>

# types.ts

```
TS types.ts x
src > 12-displaying-types > TS types.ts > DeepReadonly
19 export type PizzaOnOrder = Pick<Pizza, 'name' | 'price'> & {
20   extraIngredients: string[];
21 };
22   type PizzaOnOrder = Pick<Readonly<{
    name: string;
    ingredients: string[];
    price: number;
    extras: string[];
  }>, "name" | "price"> & {
    extraIngredients: ExtraIngredient[];
  }
```

# types.ts



```
TS types.ts M x
src > 12-displaying-types > TS types.ts > [e]
18
19 // Taken from /25/gentips-4-display/
20 type Resolve<T, K> = {
21   readonly name: string;
22   readonly price: number;
23   extraIngredients: ExtraIngredient[];
24 }
25 export type PizzaOnOrder = Resolve<
26   Pick<Pizza, 'name' | 'price'> & {
27     extraIngredients: ExtraIngredient[];
28   }>;
```

# Type Predicate Functions

# Type Predicate Functions

- Often a **TypeScript cast** is used when types don't quite line up
  - But that is **just silencing the compiler**
    - ⚠ Casting via "unknown" will even allow any (invalid) type cast ⚠
  - There is no runtime checking or guarantee
- A **type predicate** can do a cast in a runtime safe manner
  - 💡 Checks **both at runtime and compile time** 💡
  - A **function that returns a "boolean"** to indicate if the type matches



# types.ts

```
TS 13-predicates-assertions.tsx TS types.ts M X TS item-on-sale.tsx M
src > 13-predicates-assertions > TS types.ts > [e] Book
17 export type ItemsOnSale = Book | Magazine | Pen;
18
19 export function isBook(item: ItemsOnSale): item is Book {
20     return item.type === 'book';
21 }
22
23 export function isMagazine(item: ItemsOnSale): item is Magazine {
24     return item.type === 'magazine';
25 }
26
27 export function isPen(item: ItemsOnSale): item is Pen {
28     return item.type === 'pen';
29 }
```

item-on-sale.tsx



```
TS 13-predicates-assertions.tsx TS types.ts M TS item-on-sale.tsx M X
src > 13-predicates-assertions > TS item-on-sale.tsx > ...
7  type Props = {
8    item: ItemsOnSale;
9  };
10
11  export const ItemOnSale: FC<Props> = ({ item }) => {
12    if (isBook(item)) {
13      return <BookOnSale book={item} />;
14    } else if (isMagazine(item)) {
15      return <MagazineOnSale magazine={item} />;
16    }
17    // case 'pen':
18    //   return <PenOnSale pen={item as Pen} />;
19
20    return null;
21  };
```

# Type Assertion Functions

# Type Assertion Functions

- **Type assertion functions** can be even easier
  - Throw an error if the type doesn't match
- Often a **better alternative** then a cast
  - The code will not continue if the assumption is wrong

# types.ts

```
TS 13-predicates-assertions.tsx TS types.ts M X TS item-on-sale.tsx M
src > 13-predicates-assertions > TS types.ts > Book
31 export function assertBook(item: ItemsOnSale): asserts item is Book {
32     if (!isBook(item)) {
33         throw new Error('Item is not a book');
34     }
35 }
36
37 export function assertMagazine(item: ItemsOnSale): asserts item is Magazine {
38     if (!isMagazine(item)) {
39         throw new Error('Item is not a magazine');
40     }
41 }
42
43 export function assertPen(item: ItemsOnSale): asserts item is Pen {
44     if (!isPen(item)) {
45         throw new Error('Item is not a pen');
46     }
47 }
```

# item-on-sale.tsx



```
TS 13-predicates-assertions.tsx TS types.ts M TS item-on-sale.tsx M X
src > 13-predicates-assertions > TS item-on-sale.tsx > ...
 7  type Props = {
 8    item: ItemsOnSale;
 9  };
10
11  export const ItemOnSale: FC<Props> = ({ item }) => {
12    switch (item.type) {
13      case 'book':
14        assertBook(item);
15        return <BookOnSale book={item} />;
16      case 'magazine':
17        assertMagazine(item);
18        return <MagazineOnSale magazine={item} />;
19      // case 'pen':
20      //   return <PenOnSale pen={item as Pen} />;
21    }
22
23    return null;
24  };
```

# Exhaustiveness Checking

# Exhaustiveness Checking

- The TypeScript compiler doesn't tell us if **every case** is provided
  - It's easy to forget to add a switch case when an enumeration is expanded
- The "**never**" type is a great way to make sure
  - A compile error if the default case can be reached
  - 💡 Make sure to add an exception or error logging at runtime 💡



types.ts

```
TS 13-predicates-assertions.tsx TS types.ts M X TS item-on-sale.tsx M
src > 13-predicates-assertions > TS types.ts > [Book] Book
1 export type Book = {
2   type: 'book';
3   title: string;
4   description: string;
5 };
6
7 export type Magazine = {
8   type: 'magazine';
9   title: string;
10 };
11
12 export type Pen = {
13   type: 'pen';
14   color: string;
15 };
```

# types.ts

```
TS 13-predicates-assertions.tsx TS types.ts M X TS item-on-sale.tsx M
src > 13-predicates-assertions > TS types.ts > [🔍] Book
49 | export function assertNever(value: never): never {
50 |     throw new Error('Unexpected value: ' + JSON.stringify(value, null, 2));
51 | }
```

# item-on-sale.tsx



```
TS 13-predicates-assertions.tsx TS types.ts M TS item-on-sale.tsx M X
src > 13-predicates-assertions > TS item-on-sale.tsx > ...
7 type Props = {
8   item: ItemsOnSale;
9 };
10
11 export const ItemOnSale: FC<Props> = ({ item }) => {
12   switch (item.type) {
13     case 'book':
14       return <BookOnSale book={item} />;
15     case 'magazine':
16       return <MagazineOnSale magazine={item} />;
17     case 'pen':
18       return <PenOnSale pen={item} />;
19     default:
20       assertNever(item);
21   }
22
23   return null;
24 };
```

# Conclusion

- TypeScript's **strict settings** help catch many errors
  - Make sure to turn on the additional strict features as well
- TypeScript offers a lot of features for **React component props**
  - Infer or mutate prop types and detect invalid combinations of values
- **Validate all data at boundaries**
  - Not just from the user, also from API's
- Use **mapped types** to create new types
  - The possibilities are almost endless
- Use **type predicates and assertions** both at compile and run-time
  - Instead of just casting at compile time
- Enable **exhaustiveness checking** with the "never" type
  - Make sure to log unexpected cases at runtime

Maurice de Beijer

@mauricedb

maurice.de.beijer  
@gmail.com

