
TypeScript in React projects

Table of Contents

Structural typing vs nominal typing

Structural typing

Structural typing means that types with the same structure are treated as compatible regardless of name. TypeScript uses structural typing exclusively.

```
type MacBook = {
  colour: string;
}
type Animal = {
  colour: string;
  // canBeInARoom: boolean;
}
const macBookPro: MacBook = {
  colour: 'gray';
}
const elephant: Animal = {
  colour: 'gray',
}
const listOfMacBooks: MacBook[] = [
  macBookPro,
  elephant, // no error
]
```

Nominal typing

Nominal typing means that types are compatible if they have the same name or are declared as subtypes. More commonly used in mainstream OOP languages like Java. Flow uses nominal typing for class-related types.

Classes

```
public interface IMacBookAnimal {}
public record MacBook(String colour) implements IMacBookAnimal {}
public record Animal(String colour) implements IMacBookAnimal {}
```

Program

```
import java.util.Arrays;
import java.util.List;

public class NominalTypingExample {
  public static void main(String[] args) {
    var macBookPro = new MacBook("gray");
    var elephant = new Animal("gray");
    // List<MacBook> listOfMacBooks = Arrays.asList(macBookPro, elephant); // error
    List<IMacBookAnimal> listOfMacBooks = Arrays.asList(macBookPro, elephant); // no error
  }
}
```

Type modeling

Type modeling

TypesScript provides a multitude of options to model data types that can be used for business logic, props, json files etc.

Example data

```
[
  {
    "id": "123456",
    "firstName": "Peter",
    "lastName": "Peterson",
    "house": "111",
    "street": "Highstreet",
    "city": "London",
    "postcode": "W1 6AA",
    "defaultedPayments": 1,
    "missedPayments": 50
  }
]
```

Composite types from other types

```ts
export type CustomerDefaultFields = {
  id: string;
  firstName: string;
  lastName: string;
};
export type CustomerAddress = {
  house: string;
  street: string;
  city: string;
  postcode: string;
};
export type CustomerPaymentStatus = {
  defaultedPayments: number;
  missedPayments: number;
};
export type Customer =
  & CustomerDefaultFields
  & CustomerAddress
  & CustomerAddress
export type CustomerWithPaymentData =
  & CustomerDefaultFields
  & CustomerPaymentStatus
```

Types can also be created by picking and omitting certain fields from a base type via utility types. Pick allows to choose specific fields that are used to create a new type, while omit specifies the fields, that are to be excluded from the new type.

```
export type CustomerFields = {
  id: string;
  firstName: string;
  lastName: string;
  house: string;
  street: string;
  city: string;
  postcode: string;
  defaultedPayments: number;
  missedPayments: number;
  otherField: never;
};
export type Customer = Pick<CustomerFields, 'id' | 'firstName' | 'lastName' | 'house' | 'street' | 'city' |
'postcode' | 'defaultedPayments' | 'missedPayments'
>;
export type Customer2 = Omit<CustomerFields, 'otherField'>;
export type CustomerWithPaymentData = Pick<CustomerFields, 'id' | 'firstName' | 'lastName' | 'defaultedPayments' |
>;
export type CustomerWithPaymentData2 = Omit<CustomerFields, 'house' | 'street' | 'city' | 'postcode' | 'otherField
```

Pick and Omit can also be used to narrow down the type of a field, by removing the field first and then add it again with a different, e.g. narrower type.

Example

```
export type LoanStringlyTyped = {
  id: string;
  title: string;
  tranche: string;
  available_amount: string;
  annualised_return: string;
  term_remaining: string;
  ltv: string;
  loan_value: string;
};

export type Loan = Omit<LoanStringlyTyped, 'available_amount' | 'annualised_return'
| 'loan_value' | 'term_remaining'> & {
  available_amount: currency,
  annualised_return: currency,
  loan_value: currency
  term_remaining: Date;
};
```

Restrict values for types

Restricting possible values for types prevents accessing undefined or null values without having to resort to manual checks like `Object.hasOwn` or `[].includes`, which helps with readability. Furthermore it makes one's IDE to only show applicable values.

Types

```ts
export type Person = {
  firstName: string,
  lastName: string,
  isPrimeMinister: boolean,
}

export type PersonList = ReadonlyArray<Pick<Person, 'firstName' | 'lastName'> & {
  id: string,
  numberOfCats: number,
}>

export type PersonListFields = keyof PersonList[number]; // firstName | lastName | id | numberOfCats
```

Data

```ts
import { Person, PersonList } from './types';
import { faker } from '@faker-js/faker';

export const people: Person[] = Array.from({ length: 5 }, () => ({
  firstName: faker.name.firstName(),
  lastName: faker.name.lastName(),
  isPrimeMinister: false,
}));

export const peopleList: PersonList = people.map((person) => ({
  ...person,
  id: faker.datatype.uuid(),
  numberOfCats: faker.datatype.number({ min: 0, max: 50, precision: 1 }),
}));
```

Script

```
function sorted(list: PersonList, sortBy: PersonListFields): PersonList {
  const sortedList = [...list].sort((entryA, entryB) => {
    const fieldA = entryA[sortBy];
    const fieldB = entryB[sortBy];

    if (fieldA > fieldB) return 1;
    if (fieldA < fieldB) return -1;
    return 0;
  });
  return sortedList;
}

function sayIt(list: PersonList): void {
  const listAsString: string[] = list.map((entry) => Object.values(entry).join(' | '));
  console.log(listAsString);
}

const peopleListSortedByNumberOfCats: PersonList = sorted(peopleList, 'numberOfCats');
const peopleListSortedByFirstName: PersonList = sorted(peopleList, 'firstName');

sayIt(peopleListSortedByNumberOfCats);
sayIt(peopleListSortedByFirstName);
```

Differentiating types

Differentiating values

TypeScript can infer values via control flow analysis and allow or disallow certain operations at specific points within the code.

```ts
export type Person = {
  firstName: string;
  lastName: string;
  address?: {
    house: string;
    street: string;
    postcode: string;
    city: string;
  };
};
```

```ts
function showPersonDetails(person: Person): void {
  const { firstName, lastName, address } = person;

  console.log(`First name: ${firstName}`);
  console.log(`Last name: ${lastName}`);

  if (address) {
    const { house, street, postcode,
    city } = address;

    console.log(`House: ${house}`);
    console.log(`Street: ${street}`);
    console.log(`Postcode: ${postcode}`);
    console.log(`City: ${city}`);
  }
}
```

Differentiating custom types

TypeScript can narrow down types similar to values via control flow analysis and.

```ts
export type Person = {
  firstName: string;
  lastName: string;
};

export type PersonWithAddress = Person & {
  address: {
    house: string;
    street: string;
    postcode: string;
    city: string;
  };
};

export type People = Person | PersonWithAddress;
```

```ts
function showPersonDetails(
  person: Person | PersonWithAddress): void {
  const { firstName, lastName } = person;

  console.log(`First name: ${firstName}`);
  console.log(`Last name: ${lastName}`);

  // in-operator narrows down union types
  if ('address' in person) {
    // PersonWithAddress
    const type = person;
    const { house, street, postcode, city } = person
    .address;

    console.log(`House: ${house}`);
    console.log(`Street: ${street}`);
    console.log(`Postcode: ${postcode}`);
    console.log(`City: ${city}`);
  }
}
```

Discriminated unions

Discriminate unions are a way to narrow down union types by using a single field called `discriminant property`.

```ts
export type AnimalSound = 'Meow' | 'Woof';

export type Cat = {
  type: 'Cat';
  name: string;
  sound: AnimalSound;
  isCurrentChiefMouser: boolean;
};

export type Dog = {
  type: 'Dog';
  name: string;
  sound: AnimalSound;
  canBeMistakenForAPony: boolean;
};

export type Animal = Cat | Dog;

export type AnimalList = [Animal, Animal]; // Tuple
```

```ts
function showAnimalDetails(animal: Animal): void {
  const { type, name, sound } = animal;

  console.log(name);
  console.log(sound);

  if (type === 'Cat') {
    console.log(animal.isCurrentChiefMouser);

    return;
  }

  console.log(animal.canBeMistakenForAPony);
}
```

Type predicates & Type guards

Todo

```ts
export function isCustomerWithDefaultedPayments(customer: Customer): customer is CustomerWithDefaultedPayments {
  const { hasDefaultedPayments } = customer;

  return hasDefaultedPayments;
}

export function isCustomerWithMissedPayments(customer: Customer): customer is CustomerWithMissedPayments {
  const { hasDefaultedPayments, hasMissedPayments } = customer;

  if (hasDefaultedPayments) {
    return false;
  }

  return hasMissedPayments;
}

export function isCustomerRegular(customer: Customer): customer is CustomerRegular {
  const { hasDefaultedPayments, hasMissedPayments } = customer;

  return !hasDefaultedPayments && !hasMissedPayments;
}
```

React and TSX

Deprecate props

TypeScript's `never` type can be used to hard deprecate component props. This is most useful for library or styleguide maintainers. Trying to use the deprecated prop, will result in a type error. Alternatively a soft deprecation can be achieved by using the JSDoc deprecated tag, e.g. `@deprecated Please use x instead`.

Props

```
export type CustomButtonProps = Partial<Pick<HTMLButtonElement, 'autofocus' | 'ariaDisabled'>> &
{
  onClick: (event?: MouseEvent<HTMLButtonElement>) => void,
  oldProp?: never, // is deprecated
  newProp: string,
};
```

```tsx
import React, { PropsWithChildren } from 'react';
import { CustomButtonProps } from './library';

function CustomButton({ children, onClick, newProp }: PropsWithChildren<CustomButtonProps>) {
  return (
    <button type="button" onClick={onClick} data-testid={newProp}>
      {children}
    </button>
  );
}

export default function App() {
  function handleClick(): void {}

  return (
    <CustomButton
      onClick={() => handleClick()}
      newProp="test"
      // oldProp="test" // error
    >
      Button
    </CustomButton>
  );
}
```

Disallow prop combinations

TypeScript's `never` type can also be used to prohibit certain combination of props. The syntax can get a bit complex for certain type of props, e.g. `boolean` as those type of props can either be set to false or not passed at all.

Component

```
function ButtonLink({ onClick, type, disabled, href, children }: PropsWithChildren<ButtonLinkProps>) {
  const TagType = type; // either a or button
  if (type === 'button') {
    console.log(href); // undefined
    console.log(disabled);
  }
  if (type === 'a') {
    console.log(href); // string
    console.log(disabled);
  }
  return (
    <TagType className={style.ButtonLink} onClick={onClick}>
      {children}
    </TagType>
  );
}
export default ButtonLink;
```

We use the type field to distinguish between various types and allow TypeScript to narrow down the type.

Props

```
type ButtonLinkCommonProps = {
  onClick: (() => void);
}

type ButtonLinkButtonProps = {
  type: 'button';
  disabled?: boolean;
  href?: never; // button should never have a href attribute
}

type ButtonLinkLinkProps = {
  type: 'a';
  disabled?: false; // link should never have a disabled attribute
  href: string;
}

export type ButtonLinkProps = ButtonLinkCommonProps & (ButtonLinkButtonProps | ButtonLinkLinkProps);
```

Prohibit child components

TypeScript's `never` type can also be used to prevent child components. This is useful for library authors to prevent composition and use a component as is.

App

```
import Child from './Child';
import Childless from './Childless';

function App() {
  return (
    <div className="wrapper">
      <h1>Restrict child elements</h1>
      <Child>
        <h2>Can have child content</h2>
      </Child>
      <Childless />
    </div>
  );
}
```

Component with children

```tsx
import { PropsWithChildren } from 'react';

export type ChildProps = PropsWithChildren<{
  // same as PropsWithChildren<>
  // children?: ReactNode | undefined;
  otherProp: string;
}>;
```

Component without children

```tsx
export type ChildlessProps = {
  children?: never;
  otherProp: string;
};
```

Migrating to TypeScript

Soft migration

Soft migration allows a gradual migration in parallel to regular feature work, albeit at a slower pace.

1. Add a `jsconfig.json` file and set `checkJs` to `false` to disable type checking in JS files
2. Set `checkJs` to `true`, when working on the migration or enable it on a per file basis via adding a `// @ts-check` line at the start of a JS file
3. Fix all errors that are fixable without type annotations and disable the remaining ones by prepending each with `// @ts-ignore`
4. Install TypeScript dependencies and necessary type packages like `@types/react-dom`
5. Create a script, that runs the TypeScript compiler in no-emit mode to check remaining errors: `npx -`

```
{
  "compilerOptions": {
    "target": "ES2022", // ES3 is default
    "checkJs": false,
    "allowJs": true,
    "noEmit": true,
  },
  "exclude": [
    "node_modules"
  ],
  "include": [
    "./**/*.js"
  ]
}
```

Example `jsconfig.json` in project root

Soft migration (part 2)

6. Add TypeScript to build pipelines
7. Create a `tsconfig.json` file via `npx -p typescript tsc --init` and disable strict mode, strict null checks and no implicit any
8. Rename all `.js` files to `.ts` and `.tsx`
9. Reenable previously disabled errors and fix them
10. Add TypeScript-specific linter rules, f.e. `eslint-config-airbnb-typescript` and adapt code to new linter rules
11. Enable strict mode, strict null checks and no implicit any again
12. Fix TypeScript errors caused by stricter TypeScript settings

```
{
  "compilerOptions": {
    "target": "ES2022", // ES3 is default
    "strict": false,
    "strictNullChecks": false,
    "noImplicityAny": false
  },
   "lib": [
    "dom",
    "dom.iterable"
  ],
  "exclude": [
    "node_modules"
  ],
  "include": [
    "./**/*.tsx",
    "./**/*.ts"
  ]
}
```

Example `tsconfig.json` in project root

Big bang migration

Allows a faster migration by converting all files at once at the expense of stopping feature work for a certain period of time. It might require freezing the code base for the duration of the migration.

1. Install TypeScript dependencies and necessary type packages like `@types/react-dom`
2. Create a `tsconfig.json` file via `npx -p typescript tsc --init` and disable strict mode, strict null checks and no implicit any
3. Rename all `.js` files to `.ts` and `.tsx`
4. Fix all TypeScript errors
5. Add TypeScript to build pipelines
6. Enable strict mode, strict null checks and no implicit any again
7. Fix TypeScript errors caused by stricter TypeScript settings
8. Improve types for components, tests etc. further