
TypeScript in React projects

Table of Contents

1. Structural typing vs nominal typing
2. Modeling types
3. Using types
4. React and TSX
5. Migrating to TypeScript

Structural typing vs nominal typing

Structural typing

Structural typing means that types with the same structure are compatible regardless of name. TypeScript uses structural typing by default.

```ts
type MacBook = {
  name: string;
  colour: string;
  numberOfFeet: number;
  hasFans: boolean;
  canBeInARoom: boolean;
};

type Animal = {
  name: string;
  colour: string;
  numberOfFeet: number;
  hasFans: boolean;
  canBeInARoom: boolean;
};
```

```ts
const macBookPro: MacBook = {
  name: 'Mac',
  colour: 'gray',
  numberOfFeet: 4,
  hasFans: true,
  canBeInARoom: true,
};

const elephant: Animal = {
  name: 'Benjamin',
  colour: 'gray',
  numberOfFeet: 4,
  hasFans: true,
  canBeInARoom: true,
};

const listOfMacBooks: MacBook[] = [
  macBookPro,
  elephant, // no error
];
```

Nominal typing

Nominal typing means that each type is unique and types can not be used interchangeably even when structurally identically. Compatibility can be created via interfaces or inheritance.

Nominal Typing is widely used in languages like Java or C#. Flow uses a mix of nominal and structural typing. Nominal Typing can be emulated in TypeScript via Branded Types.

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

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
  }
}
```

Modeling types

TypesScript provides a very powerful type system to model types that can be used to model parameters, return values, json files etc.

Example payload from an API-endpoint

```
[
  {
    "id": "B122",
    "firstName": "Sherlock",
    "lastName": "Holmes",
    "house": "221B",
    "street": "Baker Street",
    "city": "London",
    "postcode": "NW16XE",
    "defaultedPayments": 1,
    "missedPayments": 50
  }
]
```

Corresponding type

```
type Customer = {
  id: string;
  firstName: string;
  lastName: string;
  house: string;
  street: string;
  city: string;
  postcode: string;
  defaultedPayments: number;
  missedPayments: number;
};

type CustomerList = Customer[];
```

When using types it is possible to split them up into logical units and composite new types from other types via **Intersection types.**

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

The same result can be achieved via `extend` when using interfaces.

```
interface CustomerDefaultFields {
  id: string;
  firstName: string;
  lastName: string;
}
interface CustomerAddress {
  house: string;
  street: string;
  city: string;
  postcode: string;
}
interface CustomerPaymentStatus {
  defaultedPayments: number;
  missedPayments: number;
}
interface Customer extends
  CustomerDefaultFields,
  CustomerAddress {}
interface CustomerWithPaymentData extends
  CustomerDefaultFields,
  CustomerAddress,
  CustomerPaymentStatus {}
```

Types can also be created by including or excluding certain fields from a complex base type via **Pick** or **Omit**.

```
type CustomerFields = {
  id: string;
  firstName: string;
  lastName: string;
  house: string;
  street: string;
  city: string;
  postcode: string;
  defaultedPayments: number;
  missedPayments: number;
};
type Customer = Pick<CustomerFields,
  'id' | 'firstName' | 'lastName' |
  'house' | 'street' | 'city' | 'postcode'
>;
type Customer2 = Omit<CustomerFields,
  'defaultedPayments' | 'missedPayments'
>;
```

Pick and Omit can also be used when changing the type of certain fields, by removing the fields first and then adding them again with a different, e.g. narrower type. This comes in handy when working with JSON files.

```
import currency from 'currency.js';

type LoanLooselyTyped = {
  id: string;
  title: string;
  tranche: string;
  available_amount: string;
  annualised_return: string;
  term_remaining: string;
  ltv: number;
  loan_value: string;
};

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

Union types can be used to allow a set of different values for a type. While often used as string unions, union types are not restricted to primitives. They are also used with objects that have common fields.

```ts
type AnimalSound = 'Meow' | 'Woof' | 'Moo' |
  'Oink';

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

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

type Cow = {
  name: string;
  sound: AnimalSound;
};

type Animal = Cat | Dog | Cow;
type AnimalList = Animal[];
```

```ts
const cat: Cat = {
  name: 'Mittens',
  sound: 'Meow',
  isCurrentChiefMouser: false,
};

const dog: Dog = {
  name: 'Ben',
  sound: 'Woof',
  canBeMistakenForAPony: true,
};

const cow: Cow = {
  name: 'Gertie',
  sound: 'Meow', // no error
};

const animals: AnimalList = [cat, dog, cow];
```

Discriminated unions

When union types share a common field (usually a string), that field can be used to automatically differentiate members of the union type. They are called **Discriminated unions**. The shared field is called **discriminant property**.

```ts
type AnimalSound = 'Meow' | 'Woof' | 'Moo' |
  'Oink';

type Cat = {
  type: 'Cat'; // discriminant property
  name: string;
  sound: AnimalSound;
  isCurrentChiefMouser: boolean;
};

type Dog = {
  type: 'Dog'; // discriminant property
  name: string;
  sound: AnimalSound;
  canBeMistakenForAPony: boolean;
};

type Animal = Cat | Dog;
```

```ts
// TS narrows Animal down to Cat
const cat: Animal = {
  type: 'Cat',
  name: 'Mittens',
  sound: 'Meow',
  isCurrentChiefMouser: false,
  // canBeMistakenForAPony: false,
  // Error ('canBeMistakenForAPony' does not exist
  // in type 'Cat')
};

// TS narrows Animal down to Dog
const dog: Animal = {
  type: 'Dog',
  name: 'Ben',
  sound: 'Woof',
  canBeMistakenForAPony: true,
};
```

Logical union type combinations

Union types can also be modeled in such a way, that unique logical combinations are created. TypeScript can use those to automatically differentiate members of the union type.

```
type DatePickerBase = {
  day: number;
  month: number;
  year: number;
};
type DatePickerRegular = {
  maxNumberOfDays?: number; // optional
  showTimePicker: boolean; // unique
};
type DatePickerDateOfBirthMode = {
  minAge?: number; // optional
  maxAge?: number; // optional
  showIAmOldEnoughBox: boolean; // unique
};
type DatePicker = DatePickerBase
  & (DatePickerRegular
    | DatePickerDateOfBirthMode);
```

Mapped types can be used to iterate over fields of one or more types to create a new type. This is useful with keeping the type declarations DRY.

```
type CustomerTypeBackend = {
  id: string;
  first_name: string;
  last_name: string;
  defaulted_payments: number;
  missed_payments: number;
  // additional_field_1: number[];
}

type GlobalKeyMap = {
  id: 'id',
  first_name: 'firstName',
  last_name: 'lastName',
  house: 'house',
  street: 'street',
  city: 'city',
  post_code: 'postCode',
  defaulted_payments: 'defaultedPayments',
  missed_payments: 'missedPayments',
  additionalField2: 'additional_field_2';
}
```

```
type CustomerTypeFrontendWithoutMismatchingData = {
  [K in keyof CustomerTypeBackend as
    GlobalKeyMap[K]]: CustomerTypeBackend[K]
}
```

Output:

```
type CustomerTypeFrontendWithoutMismatchingData = {
  id: string;
  firstName: string;
  lastName: string;
  defaultedPayments: number;
  missedPayments: number;
}
```

```ts
type CustomerTypeBackend = {
  id: string;
  first_name: string;
  last_name: string;
  defaulted_payments: number;
  missed_payments: number;
  additional_field_1: number[];
}
```

```ts
type GlobalKeyMap = {
  id: 'id',
  first_name: 'firstName',
  last_name: 'lastName',
  ...
  defaulted_payments: 'defaultedPayments',
  missed_payments: 'missedPayments',
  additionalField2: 'additional_field_2';
}
```

```ts
type CustomerTypeFrontend = {
  [K in keyof Pick<CustomerTypeBackend, Extract<keyof CustomerTypeBackend, keyof GlobalKeyMap>>
    as GlobalKeyMap[K]]: CustomerTypeBackend[K];
};
```

Output

```ts
type CustomerTypeFrontend = {
  id: string;
  firstName: string;
  lastName: string;
  defaultedPayments: number;
  missedPayments: number;
}
```

Mapped types can also iterate over fields of multiple types to create new types.

```ts
type CurrencyAbbreviations = 'GBP' | 'EUR' | 'USD'
  | 'CAD' | 'AUD';

type CurrencyNames = 'Pound' | 'Euro' | 'Dollar'
  | 'Canadian Dollar' | 'Australian Dollar';

type CurrencySymbols = '£' | '€' | '$';

type Currencies = {
  [K in CurrencyAbbreviations]?: {
    name: CurrencyNames;
    symbol: CurrencySymbols;
  };
};

// Alternative approach using Record and Partial
type CurrenciesAlternative = Partial<Record<
  CurrencyAbbreviations, {
    name: CurrencyNames;
    symbol: CurrencySymbols;
  }
>>;
```

```ts
const currencies: Currencies = {
  GBP: {
    name: 'Pound',
    symbol: '£',
  },
  EUR: {
    name: 'Euro',
    symbol: '€',
  },
};
```

Since mapped types provide the current type within the map function, a type lookup can be performed, where the current type acts as the key, similar to looking up values in an object.

```
type CurrencyAbbreviations = 'GBP' | 'EUR' | 'USD'
  | 'CAD' | 'AUD';
type CurrencyNames = {
  GBP: 'Pound' | 'Pound Sterling';
  EUR: 'Euro';
  USD: 'Dollar' | 'US Dollar';
  CAD: 'Canadian Dollar';
  AUD: 'Australian Dollar' | 'Dollarydoos';
};
type CurrencySymbols = {
  GBP: '£';
  EUR: '€';
  USD: '$';
  CAD: '$';
  AUD: '$';
};
type Currencies = {
  [K in CurrencyAbbreviations]?: [
    name: CurrencyNames[K],
    symbol: CurrencySymbols[K],
  ];
};
```

```
const currencies: Currencies = {
  EUR: ['Euro', '€'],
  GBP: ['Pound', '£'],
  USD: ['Dollar', '$'],
  CAD: ['Canadian Dollar', '$'],
  AUD: ['Dollarydoos', '$'],
  // CAD: ['Euro', '$'], // Error
  // GBP: ['Pound Sterling', '$'], // Error
};
```

Using types

Differentiating values in types

TypeScript can infer values via control flow analysis and may allow or disallow certain operations on members of a type. For example optional values can only be accessed after checking if they are defined.

```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 multiple types

TypeScript can also narrow down multiple types via control flow analysis. Checking types directly in the code doesn't work, as types are removed during compilation (type erasure).

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

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

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}`);

  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 in the real world

Since TypeScript is able to narrow down a union type via a discriminant property, discriminant unions are widely used for type checking in conditionals to ensure type safety and for dealing with API-requests.

```
type ActionNamesTypes = 'UPDATE_COUNTER' |
  'UPDATE_DATETIME';

type State = {
  counter: Counter;
  dateTime: DateTime;
};

type CounterAction = {
  type: Extract<ActionNamesTypes,'UPDATE_COUNTER'>
  payload: Counter;
};

type DateTimeAction = {
  type: Extract<ActionNamesTypes,'UPDATE_DATETIME'>
  payload: DateTime;
};

type Actions = CounterAction | DateTimeAction;
```

```
const reducers = (state: State = initialState,
  action: Actions): State => {
  switch (action.type) {
    case ActionNames.UPDATE_COUNTER: {
      return {
        ...state,
        counter: action.payload, // Counter
      };
    }
    case ActionNames.UPDATE_DATETIME: {
      return {
        ...state,
        dateTime: action.payload, // DateTime
      };
    }
    default:
      return {
        ...state,
      };
  }
};
```

Discriminated unions in the real world (part 2)

Discriminant unions also work with Generics to make Code more flexible.

```ts
type Loading = {
  state: 'loading';
};
type Failed = {
  state: 'failed';
  statusCode: number;
  errorMessage?: string;
};
type Success<T> = {
  state: 'success';
  statusCode: number;
  data: T;
};

type APIRequestStatus<T> = Loading | Failed |
  Success<T>;

type Cat = {
  type: 'Cat';
  name: string;
};
```

```ts
async function sendAPIRequest<T>(): Promise<void> {
  const requestStatus = await getResponse<T>();
  switch (requestStatus.state) {
    case 'loading': {
      const { state } = requestStatus;
      return;
    }
    case 'failed': {
      const { state, statusCode, errorMessage }
      = requestStatus;
      return;
    }
    case 'success': {
      const { state, statusCode, data }
      = requestStatus;
      return;
    }
    default: {}
  }
}
sendAPIRequest<Cat>();
```

Type predicates & Type guards

Todo

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

  return hasDefaultedPayments;
}

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

  if (hasDefaultedPayments) {
    return false;
  }

  return hasMissedPayments;
}

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`.

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

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

function App() {
  function handleClick(): void {
    console.log('Click');
  }

  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>
  );
}
```

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;
}

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

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

Component with children

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

type ChildProps = PropsWithChildren<{
  otherProp: string;
}>;
```

Component without children

```ts
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