

TypeScript in React projects

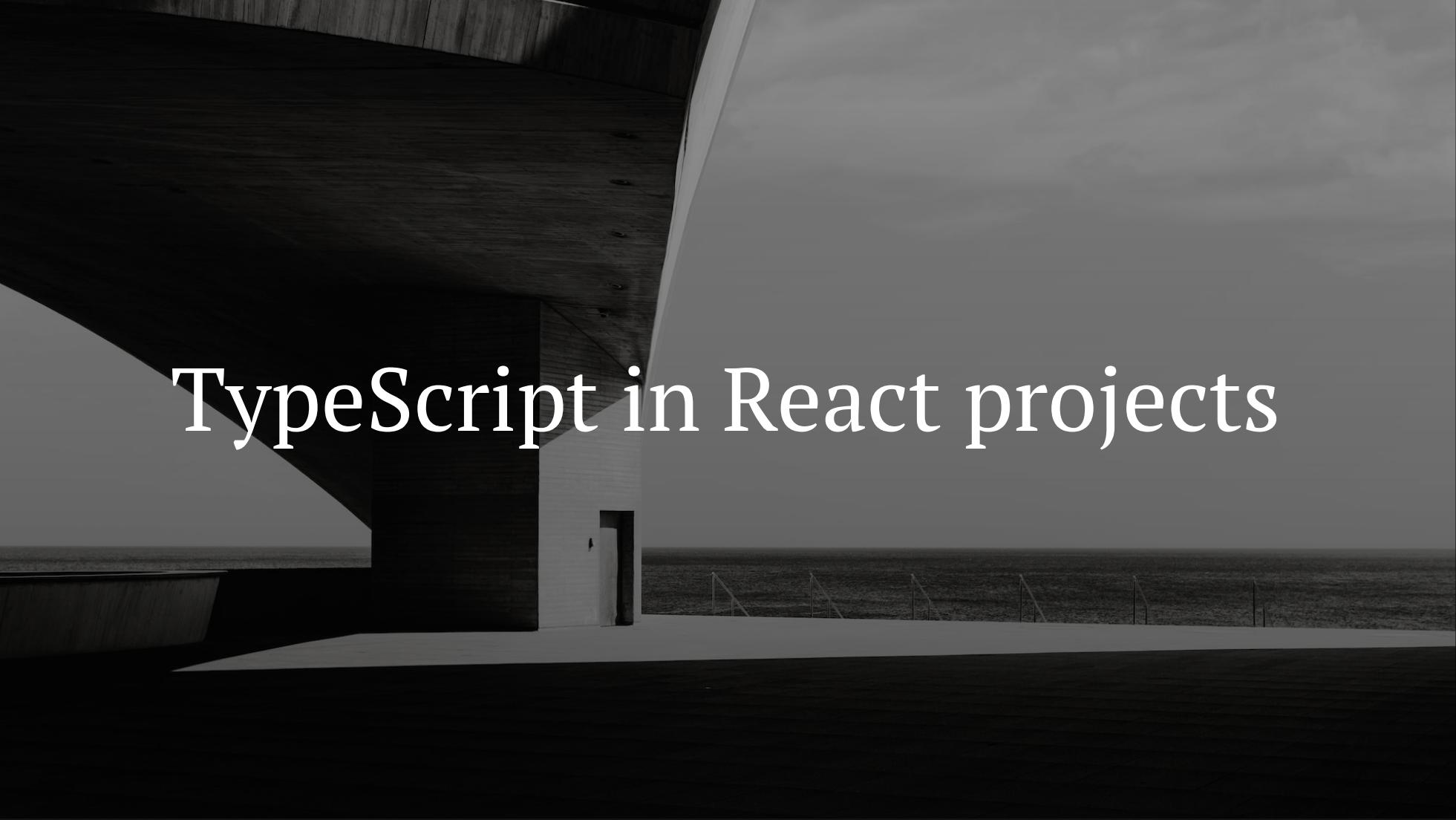


Table of Contents

1. Structural typing
2. Nominal typing
3. Restrict values for types
4. Type modeling
5. Deprecate props
6. Disallow prop combinations
7. Prohibit child components
8. Discriminated unions
9. Type predicates & Type guards
10. Leverage built in types

Structural typing

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

```
1 type MacBook = {  
2   colour: string;  
3 }  
4 type Animal = {  
5   colour: string;  
6   // canBeInARoom: boolean;  
7 }  
8 const macBookPro: MacBook = {  
9   colour: 'gray';  
10 }  
11 const elephant: Animal = {  
12   colour: 'gray',  
13 }  
14 const listOfMacBooks: MacBook[] = [  
15   macBookPro,  
16   elephant, // no error  
17 ]
```



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

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

Program

```
1  import java.util.Arrays;
2  import java.util.List;
3
4  public class NominalTypingExample {
5      public static void main(String[] args) {
6          var macBookPro = new MacBook("gray");
7          var elephant = new Animal("gray");
8          // List<MacBook> listOfMacBooks = Arrays.asList(macBookPro, elephant); // error
9          List<IMacBookAnimal> listOfMacBooks = Arrays.asList(macBookPro, elephant); // no error
10     }
11 }
```

Restrict values for types

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

Types

```
1  export type Person = {  
2    firstName: string,  
3    lastName: string,  
4    isPrimeMinister: boolean,  
5  }  
6  
7  export type PersonList = ReadonlyArray<Pick<Person, 'firstName' | 'lastName'> & {  
8    id: string,  
9    number_of_cats: number,  
10   }>  
11  
12  export type PersonListFields = keyof PersonList[number]; // firstName | lastName | id | number_of_cats
```

Data

```
1 import { Person, PersonList } from './types';
2 import { faker } from '@faker-js/faker';
3
4 export const people: Person[] = Array.from({ length: 5 }, () => ({
5   firstName: faker.name.firstName(),
6   lastName: faker.name.lastName(),
7   isPrimeMinister: false,
8 }));
9
10 export const peopleList: PersonList = people.map((person) => ({
11   ...person,
12   id: faker.datatype.uuid(),
13   numberOfCats: faker.datatype.number({ min: 0, max: 50, precision: 1 }),
14 }));
```

Script

```
1  function sorted(list: PersonList, sortBy: PersonListFields): PersonList {
2    const sortedList = [...list].sort((entryA, entryB) => {
3      const fieldA = entryA[sortBy];
4      const fieldB = entryB[sortBy];
5
6      if (fieldA > fieldB) return 1;
7      if (fieldA < fieldB) return -1;
8      return 0;
9    });
10   return sortedList;
11 }
12
13 function sayIt(list: PersonList): void {
14   const listAsString: string[] = list.map((entry) => Object.values(entry).join(' | '));
15   console.log(listAsString);
16 }
17
18 const peopleListSortedByNumberOfCats: PersonList = sorted(peopleList, 'numberOfCats');
19 const peopleListSortedByFirstName: PersonList = sorted(peopleList, 'firstName');
20
21 sayIt(peopleListSortedByNumberOfCats);
22 sayIt(peopleListSortedByFirstName);
```

Type modeling

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

Example data

```
1  [
2    {
3      "id": "123456",
4      "firstName": "Peter",
5      "lastName": "Peterson",
6      "house": "111",
7      "street": "Highstreet",
8      "city": "London",
9      "postcode": "W1 6AA",
10     "defaultedPayments": 1,
11     "missedPayments": 50
12   }
13 ]
```





Composite types from other types

```
1  export type CustomerDefaultFields = {
2    id: string;
3    firstName: string;
4    lastName: string;
5  };
6  export type CustomerAddress = {
7    house: string;
8    street: string;
9    city: string;
10   postcode: string;
11 };
12 export type CustomerPaymentStatus = {
13   defaultedPayments: number;
14   missedPayments: number;
15 };
16 export type Customer =
17   & CustomerDefaultFields
18   & CustomerAddress
19   & CustomerAddress
20 export type CustomerWithPaymentData =
21   & CustomerDefaultFields
22   & 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.

```
1  export type CustomerFields = {
2    id: string;
3    firstName: string;
4    lastName: string;
5    house: string;
6    street: string;
7    city: string;
8    postcode: string;
9    defaultedPayments: number;
10   missedPayments: number;
11   otherField: never;
12 };
13 export type Customer = Pick<CustomerFields, 'id' | 'firstName' | 'lastName' | 'house' | 'street' | 'city' |
14 'postcode' | 'defaultedPayments' | 'missedPayments'
15 >;
16 export type Customer2 = Omit<CustomerFields, 'otherField'>;
17 export type CustomerWithPaymentData = Pick<CustomerFields, 'id' | 'firstName' | 'lastName' | 'defaultedPayments' |
18 >;
19 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 narrower type. This is not strictly necessary, since fields override previous fields with the same name (just like in JavaScript), but it helps distinguishing types at a glance.

Example

```
1  export type LoanStringlyTyped = {
2    id: string;
3    title: string;
4    tranche: string;
5    available_amount: string;
6    annualised_return: string;
7    term_remaining: string;
8    ltv: string;
9    loan_value: string;
10 };
11
12 export type Loan = Omit<LoanStringlyTyped, 'available_amount' | 'annualised_return'
13 | 'loan_value' | 'term_remaining'> & {
14   available_amount: currency,
15   annualised_return: currency,
16   loan_value: currency
17   term_remaining: Date;
18 }
```

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

```
1  export type CustomButtonProps = Partial<Pick<HTMLButtonElement, 'autofocus' | 'ariaDisabled'>> &
2  {
3    onClick: (event?: MouseEvent<HTMLButtonElement>) => void,
4    oldProp?: never, // is deprecated
5    newProp: string,
6  };
```

```
1 import React, { PropsWithChildren } from 'react';
2 import { CustomButtonProps } from './library';
3
4 function CustomButton({ children, onClick, newProp }: PropsWithChildren<CustomButtonProps>) {
5   return (
6     <button type="button" onClick={onClick} data-testid={newProp}>
7       {children}
8     </button>
9   );
10 }
11
12 export default function App() {
13   function handleClick(): void {}
14
15   return (
16     <CustomButton
17       onClick={() => handleClick()}
18       newProp="test"
19       // oldProp="test" // error
20     >
21       Button
22     </CustomButton>
23   );
24 }
```

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

```
1  function ButtonLink({ onClick, type, disabled, href, children }: PropsWithChildren<ButtonLinkProps>) {
2    const TagType = type; // either a or button
3    if (type === 'button') {
4      console.log(href); // undefined
5      console.log(disabled);
6    }
7    if (type === 'a') {
8      console.log(href); // string
9      console.log(disabled);
10   }
11   return (
12     <TagType className={style.ButtonLink} onClick={onClick}>
13       {children}
14     </TagType>
15   );
16 }
17 export default ButtonLink;
```

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

Props

```
1  type ButtonLinkCommonProps = {
2    onClick: (() => void);
3  }
4
5  type ButtonLinkButtonProps = {
6    type: 'button';
7    disabled?: boolean;
8    href?: never; // button should never have a href attribute
9  }
10
11 type ButtonLinkLinkProps = {
12   type: 'a';
13   disabled?: false; // link should never have a disabled attribute
14   href: string;
15 }
16
17 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

```
1 import Child from './Child';
2 import Childless from './Childless';
3
4 function App() {
5   return (
6     <div className="wrapper">
7       <h1>Restrict child elements</h1>
8       <Child>
9         <h2>Can have child content</h2>
10      </Child>
11      <Childless />
12    </div>
13  );
14}
```





Component with children

```
1 import { PropsWithChildren } from 'react';
2
3 export type ChildProps = PropsWithChildren<{
4   // same as PropsWithChildren<>
5   // children?: ReactNode | undefined;
6   otherProp: string;
7 }>;
```

Component without children

```
1 export type ChildlessProps = {
2   children?: never;
3   otherProp: string;
4 };
```

Discriminated unions

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

Types

```
1  export type AnimalSound = 'Meow' | 'Woof';
2
3  export type Cat = {
4      type: 'Cat'; // discriminant property
5      name: string;
6      sound: AnimalSound;
7      isCurrentChiefMouserToTheCabinetOffice: boolean;
8  };
9
10 export type Dog = {
11     type: 'Dog'; // discriminant property
12     name: string;
13     sound: AnimalSound;
14 };
15
16 export type Animal = Cat | Dog;
17
18 export type AnimalList = [Animal, Animal];
```

Function

```
1  function showAnimalDetails(animal: Animal) {
2      // can't access isCurrentChiefMouserToTheCabinetOffice here, since it doesn't exist on all animal types
3      // console.log(animal.isCurrentChiefMouserToTheCabinetOffice);
4
5      if (animal.type === 'Cat') {
6          // animal has been narrowed down to Cat
7          // Can access isCurrentChiefMouserToTheCabinetOffice here, since typescript has narrowed the type down to Cat
8          console.log(animal.isCurrentChiefMouserToTheCabinetOffice);
9
10     return;
11 }
12
13 // animal has been narrowed down to Dog
14 // can't access isCurrentChiefMouserToTheCabinetOffice here, since it doesn't exist on the Dog type
15 // console.log(animal.isCurrentChiefMouserToTheCabinetOffice);
16 }
```

Type predicates & Type guards

Todo

```
1  export function isCustomerWithDefaultedPayments(customer: Customer): customer is CustomerWithDefaultedPayments {
2      const { hasDefaultedPayments } = customer;
3
4      return hasDefaultedPayments;
5  }
6
7  export function isCustomerWithMissedPayments(customer: Customer): customer is CustomerWithMissedPayments {
8      const { hasDefaultedPayments, hasMissedPayments } = customer;
9
10     if (hasDefaultedPayments) {
11         return false;
12     }
13
14     return hasMissedPayments;
15 }
16
17 export function isCustomerRegular(customer: Customer): customer is CustomerRegular {
18     const { hasDefaultedPayments, hasMissedPayments } = customer;
19
20     return !hasDefaultedPayments && !hasMissedPayments;
21 }
```

Leverage built in types

ReadonlyArray, Tuples, Record,