



TypeScript Style Guide

Introduction

TypeScript Style Guide provides a concise set of conventions and best practices for creating consistent, maintainable code.

Table of Contents

[Expand](#)

About Guide

What

Since "consistency is the key", TypeScript Style Guide strives to enforce the majority of rules using automated tools such as ESLint, TypeScript, Prettier, etc. However, certain design and architectural decisions must still be followed, as described in the conventions below.

Why

- As project grow in size and complexity, maintaining code quality and ensuring consistent practices become increasingly challenging.
- Defining and following a standard approach to writing TypeScript applications leads to a consistent codebase and faster development cycles.
- No need to discuss code styles during code reviews.
- Saves team time and energy.

Disclaimer

Like any code style guide, this one is opinionated, setting conventions (sometimes arbitrary) to govern our code.

You don't have to follow every convention exactly as written, decide what works best for your product and team to maintain consistency in your codebase.

Requirements

This Style Guide requires:

- [TypeScript v5](#)
- [typescript-eslint v8](#) with `strict-type-checked` configuration enabled.

The Style Guide assumes but is not limited to using:

- React for frontend conventions
- Playwright and Vitest for testing conventions

TLDR

- Embrace **const assertions** for type safety and immutability. ↓
- Strive for **data immutability** using types like `readonly` and `readonly[]`.
↓
- Make the **majority of object properties required** (use optional properties sparingly). ↓
- Embrace **discriminated unions**. ↓
- Avoid **type assertions** in favor of proper type definitions. ↓
- Strive for functions to be **pure, stateless**, and have **single responsibility**. ↓
- Maintain **consistent and readable naming conventions** throughout the codebase. ↓
- Use **named exports**. ↓
- Organize code by feature and collocate related code as close as possible. ↓

Types

When creating types, consider how they would best **describe our code**.

Being expressive and keeping types as **narrow as possible** offers several benefits to the codebase:

- Increased Type Safety - Catch errors at compile time, as narrowed types provide more specific information about the shape and behavior of your data.
- Improved Code Clarity - Reduces cognitive load by providing clearer boundaries and constraints on your data, making your code easier for other developers to understand.
- Easier Refactoring - With narrower types, making changes to your code becomes less risky, allowing you to refactor with confidence.
- Optimized Performance - In some cases, narrow types can help the TypeScript compiler generate more optimized JavaScript code.

Type Inference

As a rule of thumb, explicitly declare types only when it helps to narrow them.

Note ⓘ

Explicitly declare types when doing so helps to narrow them:

```
// ✗ Avoid
const employees = new Map(); // Inferred as wide type
'Map<any, any>'
employees.set('Lea', 17);
type UserRole = 'admin' | 'guest';
const [userRole, setUserRole] = useState('admin'); // Inferred
as 'string', not the desired narrowed literal type

// ✅ Use explicit type declarations to narrow the types.
const employees = new Map<string, number>(); // Narrowed to
'Map<string, number>'
employees.set('Gabriel', 32);
type UserRole = 'admin' | 'guest';
const [userRole, setUserRole] = useState<UserRole>('admin');
// Explicit type 'UserRole'
```

Avoid explicitly declaring types when they can be inferred:

```
// ✗ Avoid
const userRole: string = 'admin'; // Inferred as wide type
'string'
const employees = new Map<string, number>([['Gabriel', 32]]); // Redundant type declaration
const [isActive, setIsActive] = useState<boolean>(false); // Redundant, inferred as 'boolean'

// ✅ Use type inference.
const USER_ROLE = 'admin'; // Inferred as narrowed string literal type 'admin'
const employees = new Map([['Gabriel', 32]]); // Inferred as 'Map<string, number>'
const [isActive, setIsActive] = useState(false); // Inferred as 'boolean'
```

Data Immutability

Immutability should be a key principle. Wherever possible, data should remain immutable by leveraging types like `Readonly` and `ReadonlyArray`.

- Using readonly types prevents accidental data mutations and reduces the risk of bugs caused by unintended side effects, ensuring data integrity throughout the application lifecycle.
- When performing data processing, always return new arrays, objects, or other reference-based data structures. To minimize cognitive load for future

developers, strive to keep data objects flat and concise.

- Use mutations sparingly, only in cases where they are truly necessary, such as when dealing with complex objects or optimizing for performance.

```
// ✗ Avoid data mutations
const removeFirstUser = (users: Array<User>) => {
  if (users.length === 0) {
    return users;
  }
  return users.splice(1);
};

// ✓ Use readonly type to prevent accidental mutations
const removeFirstUser = (users: ReadonlyArray<User>) => {
  if (users.length === 0) {
    return users;
  }
  return users.slice(1);
  // Using arr.splice(1) errors – Function 'splice' does not
  // exist on 'users'
};
```

Required & Optional Object Properties

Strive to have the majority of object properties required and use optional properties sparingly.

This approach reflects designing type-safe and maintainable code:

- Clarity and Predictability - Required properties make it explicit which data is always expected. This reduces ambiguity for developers using or consuming the object, as they know exactly what must be present.
- Type Safety - When properties are required, TypeScript can enforce their presence at compile time. This prevents runtime errors caused by missing properties.
- Avoids Overuse of Optional Chaining - If too many properties are optional, it often leads to extensive use of optional chaining (`(?.)`) to handle potential undefined values. This clutters the code and obscures its intent.

If introducing many optional properties truly can't be avoided, utilize **discriminated union types**.

```
// ❌ Avoid optional properties when possible, as they
// increase complexity and ambiguity
type User = {
    id?: number;
    email?: string;
    dashboardAccess?: boolean;
    adminPermissions?: ReadonlyArray<string>;
    subscriptionPlan?: 'free' | 'pro' | 'premium';
    rewardsPoints?: number;
    temporaryToken?: string;
};

// ✅ Prefer required properties. If optional properties are
// unavoidable,
// use a discriminated union to make object usage explicit and
// predictable.
type AdminUser = {
    role: 'admin';
    id: number;
    email: string;
    dashboardAccess: boolean;
    adminPermissions: ReadonlyArray<string>;
};

type RegularUser = {
    role: 'regular';
    id: number;
    email: string;
    subscriptionPlan: 'free' | 'pro' | 'premium';
    rewardsPoints: number;
};

type GuestUser = {
    role: 'guest';
    temporaryToken: string;
};

// Discriminated union type 'User' ensures clear intent with
// no optional properties
type User = AdminUser | RegularUser | GuestUser;

const regularUser: User = {
    role: 'regular',
    id: 212,
    email: 'lea@user.com',
    subscriptionPlan: 'pro',
```

```

    rewardsPoints: 1500,
    dashboardAccess: false, // Error: 'dashboardAccess' property
    does not exist
};

```

Discriminated Union

If there's only one TypeScript feature to choose from, embrace discriminated unions.

Discriminated unions are a powerful concept to model complex data structures and improve type safety, leading to clearer and less error-prone code.

You may encounter discriminated unions under different names such as tagged unions or sum types in various programming languages as C, Haskell, Rust (in conjunction with pattern-matching).

Discriminated unions advantages:

- As mentioned in [Required & Optional Object Properties](#), [Args as Discriminated Type](#) and [Props as Discriminated Type](#), discriminated unions remove optional object properties, reducing complexity.
- Exhaustiveness check - TypeScript can ensure that all possible variants of a type are implemented, eliminating the risk of undefined or unexpected behavior at runtime.



```

type Circle = { kind: 'circle'; radius: number };
type Square = { kind: 'square'; size: number };
type Triangle = { kind: 'triangle'; base: number; height: number };

// Create discriminated union 'Shape', with 'kind' property
// to discriminate the type of object.
type Shape = Circle | Square | Triangle;

// TypeScript warns us with errors in calculateArea
// function
const calculateArea = (shape: Shape) => {
  // Error - Switch is not exhaustive. Cases not matched:
  "triangle"
  switch (shape.kind) {
    case 'circle':
      return Math.PI * shape.radius ** 2;
  }
}

```

```

    case 'square':
        return shape.size * shape.width; // Error - Property
        'width' does not exist on type 'square'
    }
};


```

- Avoid code complexity introduced by **flag variables**.
- Clear code intent, as it becomes easier to read and understand by explicitly indicating the possible cases for a given type.
- TypeScript can narrow down union types, ensuring code correctness at compile time.
- Discriminated unions make refactoring and maintenance easier by providing a centralized definition of related types. When adding or modifying types within the union, the compiler reports any inconsistencies throughout the codebase.
- IDEs can leverage discriminated unions to provide better autocompletion and type inference.

Type-Safe Constants With `satisfies`

The `as const satisfies` syntax is a powerful TypeScript feature that combines strict type-checking and immutability for constants. It is particularly useful when defining constants that need to conform to a specific type.

Key benefits:

- Immutability with `as const`
 - Ensures the constant is treated as readonly.
 - Narrows the types of values to their literals, preventing accidental modifications.
- Validation with `satisfies`
 - Ensures the object conforms to a broader type without widening its inferred type.
 - Helps catch type mismatches at compile time while preserving narrowed inferred types.

Array constants:

```

type UserRole = 'admin' | 'editor' | 'moderator' | 'viewer' |
'guest';

// ✗ Avoid constant of wide type

```

```

const DASHBOARD_ACCESS_ROLES: ReadonlyArray<UserRole> =
['admin', 'editor', 'moderator'];

// ✗ Avoid constant with incorrect values
const DASHBOARD_ACCESS_ROLES = ['admin', 'contributor',
'analyst'] as const;

// ✅ Use immutable constant of narrowed type
const DASHBOARD_ACCESS_ROLES = ['admin', 'editor',
'moderator'] as const satisfies ReadonlyArray<UserRole>;

```

Object constants:

```

type OrderStatus = {
  pending: 'pending' | 'idle';
  fulfilled: boolean;
  error: string;
};

// ✗ Avoid mutable constant of wide type
const IDLE_ORDER: OrderStatus = {
  pending: 'idle',
  fulfilled: true,
  error: 'Shipping Error',
};

// ✗ Avoid constant with incorrect values
const IDLE_ORDER = {
  pending: 'done',
  fulfilled: 'partially',
  error: 116,
} as const;

// ✅ Use immutable constant of narrowed type
const IDLE_ORDER = {
  pending: 'idle',
  fulfilled: true,
  error: 'Shipping Error',
} as const satisfies OrderStatus;

```

Template Literal Types

Embrace template literal types as they allow you to create precise and type-safe string constructs by interpolating values. They are a powerful alternative to using the wide string type, providing better type safety.

Adopting template literal types brings several advantages:

- Prevent errors caused by typos or invalid strings.
- Provide better type safety and autocompletion support.
- Improve code maintainability and readability.

Template literal types are useful in various practical scenarios, such as:

- String Patterns - Use template literal types to enforce valid string patterns.

```
// ✗ Avoid
const appVersion = '2.6';
// ✓ Use
type Version = `v${number}.${number}.${number}`;
const appVersion: Version = 'v2.6.1';
```

- API Endpoints - Use template literal types to restrict values to valid API routes.

```
// ✗ Avoid
const userEndpoint = '/api/usersss'; // Type 'string' –
// Typo 'usersss': the route doesn't exist, leading to a
// runtime error.
// ✓ Use
type ApiRoute = 'users' | 'posts' | 'comments';
type ApiEndpoint = `/api/${ApiRoute}`; // Type ApiEndpoint
= "/api/users" | "/api/posts" | "/api/comments"
const userEndpoint: ApiEndpoint = '/api/users';
```

- Internationalization Keys - Avoid relying on raw strings for translation keys, which can lead to typos and missing translations. Use template literal types to define valid translation keys.

```
// ✗ Avoid
const homeTitle = 'translation.homesss.title'; // Type
// 'string' – Typo 'homesss': the translation doesn't exist,
// leading to a runtime error.
// ✓ Use
type LocaleKeyPages = 'home' | 'about' | 'contact';
type TranslationKey =
`translation.${LocaleKeyPages}.${string}`; // Type
TranslationKey = `translation.home.${string}` |
`translation.about.${string}` |
```

```
`translation.contact.${string}`  
const homeTitle: TranslationKey = 'translation.home.title';
```

- CSS Utilities - Avoid raw strings for color values, which can lead to invalid or non-existent colors. Use template literal types to enforce valid color names and values.

```
// ✗ Avoid  
const color = 'blue-450'; // Type 'string' – Color 'blue-  
450' doesn't exist, leading to a runtime error.  
// ✓ Use  
type BaseColor = 'blue' | 'red' | 'yellow' | 'gray';  
type Variant = 50 | 100 | 200 | 300 | 400;  
type Color = `${BaseColor}-${Variant}` | `#${string}`; //  
Type Color = "blue-50" | "blue-100" | "blue-200" ... |  
"red-50" | "red-100" ... | `${string}  
const iconColor: Color = 'blue-400';  
const customColor: Color = '#AD3128';
```

- Database queries - Avoid using raw strings for table or column names, which can lead to typos and invalid queries. Use template literal types to define valid tables and column combinations.

```
// ✗ Avoid  
const query = 'SELECT name FROM usersss WHERE age > 30'; //  
Type 'string' – Typo 'usersss': table doesn't exist, leading  
to a runtime error.  
// ✓ Use  
type Table = 'users' | 'posts' | 'comments';  
type Column<TTableName extends Table> =  
  TTableName extends 'users' ? 'id' | 'name' | 'age' :  
  TTableName extends 'posts' ? 'id' | 'title' | 'content' :  
  TTableName extends 'comments' ? 'id' | 'postId' | 'text' :  
  never;  
  
type Query<TTableName extends Table> = `SELECT  
${Column<TTableName>} FROM ${TTableName} WHERE ${string}`;  
const userQuery: Query<'users'> = 'SELECT name FROM users  
WHERE age > 30'; // Valid query  
const invalidQuery: Query<'users'> = 'SELECT title FROM users  
WHERE age > 30'; // Error: 'title' is not a column in 'users'  
table.
```

Type any & unknown

`any` data type must not be used as it represents literally "any" value that TypeScript defaults to and skips type checking since it cannot infer the type. As such, `any` is dangerous, it can mask severe programming errors.

When dealing with ambiguous data type use `unknown`, which is the type-safe counterpart of `any`.

`unknown` doesn't allow dereferencing all properties (anything can be assigned to `unknown`, but `unknown` isn't assignable to anything).

```
// ✗ Avoid any
const foo: any = 'five';
const bar: number = foo; // no type error

// ✅ Use unknown
const foo: unknown = 5;
const bar: number = foo; // type error - Type 'unknown' is not
assignable to type 'number'

// Narrow the type before dereferencing it using:
// Type guard
const isNumber = (num: unknown): num is number => {
    return typeof num === 'number';
};
if (!isNumber(foo)) {
    throw Error(`API provided a fault value for field
'foo': ${foo}. Should be a number!`);
}
const bar: number = foo;

// Type assertion
const bar: number = foo as number;
```

Type & Non-nullability Assertions

Type assertions `user as User` and non-nullability assertions `user!.name` are unsafe. Both only silence TypeScript compiler and increase the risk of crashing application at runtime.

They can only be used as an exception (e.g. third party library types mismatch, dereferencing `unknown` etc.) with a strong rational for why it's introduced into the codebase.

```
type User = { id: string; username: string; avatar: string |
    null };
```

```
// ✗ Avoid type assertions
const user = { name: 'Nika' } as User;
// ✗ Avoid non-nullability assertions
renderUserAvatar(user!.avatar); // Runtime error

const renderUserAvatar = (avatar: string) => {...}
```

Type Errors

When a TypeScript error cannot be mitigated, use `@ts-expect-error` as a last resort to suppress it.

This directive notifies the compiler when the suppressed error no longer exists, ensuring errors are revisited once they're obsolete, unlike `@ts-ignore`, which can silently linger even after the error is resolved.

- Always use `@ts-expect-error` with a clear description explaining why it is necessary.
- Avoid `@ts-ignore`, as it does not track suppressed errors.



```
// ✗ Avoid @ts-ignore as it will do nothing if the following
line is error-free.
// @ts-ignore
const newUser = createUser('Gabriel');

// ✅ Use @ts-expect-error with description.
// @ts-expect-error: This library function has incorrect type
definitions - createUser accepts string as an argument.
const newUser = createUser('Gabriel');
```

Type Definition

TypeScript provides two options for defining types: `type` and `interface`. While these options have some functional differences, they are interchangeable in most cases. To maintain consistency, choose one and use it consistently.

Define all types using type alias 

Note ⓘ

```
// ✗ Avoid interface definitions
interface UserRole = 'admin' | 'guest'; // Invalid -
interfaces can't define type unions
```

```

interface UserInfo {
  name: string;
  role: 'admin' | 'guest';
}

// ✅ Use type definition
type UserRole = 'admin' | 'guest';

type UserInfo = {
  name: string;
  role: UserRole;
};

```

When performing declaration merging (e.g. extending third-party library types), use `interface` and disable the lint rule where necessary.

```

// types.ts
declare namespace NodeJS {
  // eslint-disable-next-line @typescript-eslint/consistent-type-definitions
  export interface ProcessEnv {
    NODE_ENV: 'development' | 'production';
    PORT: string;
    CUSTOM_ENV_VAR: string;
  }
}

// server.ts
app.listen(process.env.PORT, () => {...}

```

Array Types

Array types should be defined using generic syntax  Rule

Note ⓘ

```

// ✖️ Avoid
const x: string[] = ['foo', 'bar'];
const y: readonly string[] = ['foo', 'bar'];

// ✅ Use
const x: Array<string> = ['foo', 'bar'];
const y: ReadonlyArray<string> = ['foo', 'bar'];

```

Type Imports and Exports

TypeScript allows specifying a `type` keyword on imports to indicate that the export exists only in the type system, not at runtime.

Type imports must always be separated:

- Tree Shaking and Dead Code Elimination: If you use `import` for types instead of `import type`, the bundler might include the imported module in the bundle unnecessarily, increasing the size. Separating imports ensures that only necessary runtime code is included.
- Minimizing Dependencies: Some modules may contain both runtime and type definitions. Mixing type imports with runtime imports might lead to accidental inclusion of unnecessary runtime code.
- Improves code clarity by making the distinction between runtime dependencies and type-only imports explicit.



```
// ✗ Avoid using `import` for both runtime and type
import { MyClass } from 'some-library';

// Even if MyClass is only a type, the entire module might be
// included in the bundle.

// ✅ Use `import type`
import type { MyClass } from 'some-library';

// This ensures only the type is imported and no runtime code
// from "some-library" ends up in the bundle.
```

Services & Types Generation

Documentation becomes outdated the moment it's written, and worse than no documentation is wrong documentation. The same applies to types when describing the modules your app interacts with, such as APIs, messaging protocols and databases.

For external services, such as REST, GraphQL, and MQ it's crucial to generate types from their contracts, whether they use Swagger, schemas, or other sources (e.g. `openapi-ts`, `graphql-config`). Avoid manually declaring and maintaining types, as they can easily fall out of sync.

As an exception, only manually declare types when no options are available, such as when there is no documentation for the service, data cannot be fetched to retrieve a contract, or the database cannot be accessed to infer types.

Functions

Function conventions should be followed as much as possible (some of the conventions derive from functional programming basic concepts):

General

Function:

- should have single responsibility.
- should be stateless where the same input arguments return same value every single time.
- should accept at least one argument and return data.
- should not have side effects, but be pure. Implementation should not modify or access variable value outside its local environment (global state, fetching etc.).

Single Object Arg

To keep function readable and easily extensible for the future (adding/removing args), strive to have single object as the function arg, instead of multiple args.

As an exception this does not apply when having only one primitive single arg (e.g. simple functions isNumber(value), implementing currying etc.).

```
// ✗ Avoid having multiple arguments
transformUserInput('client', false, 60, 120, null, true,
2000);

// ✓ Use options object as argument
transformUserInput({
  method: 'client',
  isValidated: false,
  minLines: 60,
  maxLines: 120,
  defaultInput: null,
  shouldLog: true,
  timeout: 2000,
});
```

Required & Optional Args

Strive to have majority of args required and use optional sparingly.

If the function becomes too complex, it probably should be broken into smaller pieces.

An exaggerated example where implementing 10 functions with 5 required args each, is better than implementing one "can do it all" function that accepts 50 optional args.

Args as Discriminated Type

When applicable use **discriminated union type** to eliminate optional properties, which will decrease complexity on function API and only required properties will be passed depending on its use case.

```
// ✗ Avoid optional properties as they increase complexity  
// and ambiguity in function APIs  
type StatusParams = {  
    data?: Products;  
    title?: string;  
    time?: number;  
    error?: string;  
};  
  
// ✓ Prefer required properties. If optional properties are  
// unavoidable,  
// use a discriminated union to represent distinct use cases  
// with required properties.  
type StatusSuccessParams = {  
    status: 'success';  
    data: Products;  
    title: string;  
};  
  
type StatusLoadingParams = {  
    status: 'loading';  
    time: number;  
};  
  
type StatusErrorParams = {  
    status: 'error';  
    error: string;  
};  
  
// Discriminated union 'StatusParams' ensures predictable  
// function arguments with no optional properties  
type StatusParams = StatusSuccessParams | StatusLoadingParams  
| StatusErrorParams;
```

```
export const parseStatus = (params: StatusParams) => { ... }
```

Return Types

Requiring explicit return types improves safety, catches errors early, and helps with long-term maintainability. However, excessive strictness can slow development and add unnecessary redundancy.

As a rule of thumb, be explicit on the outside, implicit on the inside. For example, when building APIs or libraries, always type everything explicitly to avoid accidental breaking changes. For internal logic, let TypeScript infer its defaults, which will provide strong type safety without added verbosity.

Consider the advantages of explicitly defining the return type of a function:

- **Improves Readability:** Clearly specifies what type of value the function returns, making the code easier to understand for those calling the function.
- **Avoids Misuse:** Ensures that calling code does not accidentally attempt to use an undefined value when no return value is intended.
- **Surfaces Type Errors Early:** Helps catch potential type errors during development, especially when code changes unintentionally alter the return type.
- **Simplifies Refactoring:** Ensures that any variable assigned to the function's return value is of the correct type, making refactoring safer and more efficient.
- **Encourages Design Discussions:** Similar to Test-Driven Development (TDD), explicitly defining function arguments and return types promotes discussions about a function's functionality and interface ahead of implementation.
- **Optimizes Compilation:** While TypeScript's type inference is powerful, explicitly defining return types can reduce the workload on the TypeScript compiler, improving overall performance.

As context matters, use explicit return types when they add clarity and safety.

Explicitly defining the return type of a function is encouraged, although not required



Variables

Const Assertion

Strive declaring constants using const assertion `as const`:

Constants are used to represent values that are not meant to change, ensuring reliability and consistency in a codebase. Using `const` assertions further enhances type safety and immutability, making your code more robust and predictable.

- Type Narrowing - Using `as const` ensures that literal values (e.g., numbers, strings) are treated as exact values instead of generalized types like `number` or `string`.
- Immutability - Objects and arrays get readonly properties, preventing accidental mutations.

Examples:

- Objects

```
// ✗ Avoid
const FOO_LOCATION = { x: 50, y: 130 }; // Type { x: number; y: number; }
FOO_LOCATION.x = 10;

// ✓ Use
const FOO_LOCATION = { x: 50, y: 130 } as const; // Type '{ readonly x: 50; readonly y: 130; }'
FOO_LOCATION.x = 10; // Error
```

- Arrays

```
// ✗ Avoid
const BAR_LOCATION = [50, 130]; // Type number[]
BAR_LOCATION.push(10);

// ✓ Use
const BAR_LOCATION = [50, 130] as const; // Type 'readonly [10, 20]'
BAR_LOCATION.push(10); // Error
```

- Template Literals

```
// ✗ Avoid
const RATE_LIMIT = 25;
const RATE_LIMIT_MESSAGE = `Max number of requests/min is ${RATE_LIMIT}.`; // Type string

// ✓ Use
```

```
const RATE_LIMIT = 25;
const RATE_LIMIT_MESSAGE = `Max number of requests/min is
${RATE_LIMIT}.` as const; // Type 'Rate limit exceeded! Max
number of requests/min is 25.'
```

Enums & Const Assertion

Enums are discouraged in the TypeScript ecosystem due to their runtime cost and quirks.

The TypeScript documentation outlines several [pitfalls](#), and recently introduced the `--erasableSyntaxOnly` flag to disable runtime-generating features like enums altogether.



As rule of a thumb, prefer:

- Literal types whenever possible.
- Const assertion arrays when looping through values.
- Const assertion objects when enumerating arbitrary values.

Examples:

- Use literal types to avoid runtime objects and reduce bundle size.

```
// ✗ Avoid using enums as they increase the bundle size
enum UserRole {
  GUEST = 'guest',
  MODERATOR = 'moderator',
  ADMINISTRATOR = 'administrator',
}

// Transpiled JavaScript
('use strict');
var UserRole;
(function (UserRole) {
  UserRole['GUEST'] = 'guest';
  UserRole['MODERATOR'] = 'moderator';
  UserRole['ADMINISTRATOR'] = 'administrator';
})(UserRole || (UserRole = {}));

// ✓ Use literal types – Types are stripped during
transpilation
type UserRole = 'guest' | 'moderator' | 'administrator';
```

```
const isGuest = (role: UserRole) => role === 'guest';
```

- Use const assertion arrays when looping through values.

```
// ✗ Avoid using enums
enum USER_ROLES {
    guest = 'guest',
    moderator = 'moderator',
    administrator = 'administrator',
}

// ✓ Use const assertions arrays
const USER_ROLES = ['guest', 'moderator', 'administrator']
as const;
type UserRole = (typeof USER_ROLES)[number];

const seedDatabase = () => {
    USER_ROLES.forEach((role) => {
        db.roles.insert(role);
    })
}
const insert = (role: UserRole) => { ... }

const UsersRoleList = () => {
    return (
        <div>
            {USER_ROLES.map((role) => (
                <Item key={role} role={role} />
            )));
        </div>
    );
}
const Item = ({ role }: { role: UserRole }) => { ... }
```

- Use const assertion objects when enumerating arbitrary values.

```
// ✗ Avoid using enums
enum COLORS {
    primary = '#B33930',
    secondary = '#113A5C',
    brand = '#9C0E7D',
}

// ✓ Use const assertions objects
const COLORS = {
```

```

primary: '#B33930',
secondary: '#113A5C',
brand: '#9C0E7D',
} as const;

type Colors = typeof COLORS;
type ColorKey = keyof Colors; // Type "primary" |
"secondary" | "brand"
type ColorValue = Colors[ColorKey]; // Type "#B33930" |
"#113A5C" | "#9C0E7D"

const setColor = (color: ColorValue) => { ...

setColor(COLORS.primary);
setColor('#B33930');

```

Type Union & Boolean Flags

Embrace type unions, especially when type union options are mutually exclusive, instead multiple boolean flag variables.

Boolean flags have a tendency to accumulate over time, leading to confusing and error-prone code, since they hide the actual app state.

```

// ✗ Avoid introducing multiple boolean flag variables
const isPending, isProcessing, isConfirmed, isExpired;

// ✅ Use type union variable
type UserStatus = 'pending' | 'processing' | 'confirmed' |
'expired';
const userStatus: UserStatus;

```

When boolean flags are used and the number of possible states grows quickly, it often results in unhandled or ambiguous states. Instead, take advantage of [discriminated unions](#) to better manage and represent your application's state.

Null & Undefined

In TypeScript types `null` and `undefined` many times can be used interchangeably.

Strive to:

- Use `null` to explicitly state it has no value - assignment, return function type etc.

- Use `undefined` assignment when the value doesn't exist. E.g. exclude fields in form, request payload, database query ([Prisma differentiation](#)) etc.

Naming

Strive to keep naming conventions consistent and readable, with important context provided, because another person will maintain the code you have written.

Named Export

Named exports must be used to ensure that all imports follow a uniform pattern



Rule

This keeps variables, functions etc. names consistent across the entire codebase. Named exports have the benefit of erroring when import statements try to import something that hasn't been declared.

Naming Conventions

While it's often hard to find the best name, aim to optimize code for consistency and future readers by following these conventions:

Variables

- **Locals**

Camel case

`products`, `productsFiltered`

- **Booleans**

Prefixed with `is`, `has` etc.

`isDisabled`, `hasProduct`



Rule

- **Constants**

Capitalized

```
const FEATURED_PRODUCT_ID = '8f47d2a1-b13e-4d5a-a7d8-  
6ef1234';
```

- **Object & Array Constants**

Singular, capitalized with const assertion.

```
const IDLE_ORDER = {  
  pending: 'idle',
```

```

    fulfilled: true,
    error: 'Shipping Error',
} as const;

const DASHBOARD_ACCESS_ROLES = ['admin', 'editor',
'moderator'] as const;

```

If a type exists use [Type-Safe Constants With Satisfies](#).

```

// Type OrderStatus is predefined (e.g. generated from database schema, API)
type OrderStatus = {
    pending: 'pending' | 'idle';
    fulfilled: boolean;
    error: string;
};

const IDLE_ORDER = {
    pending: 'idle',
    fulfilled: true,
    error: 'Shipping Error',
} as const satisfies OrderStatus;

// Type UserRole is predefined
type UserRole = 'admin' | 'editor' | 'moderator' | 'viewer' |
'guest';

const DASHBOARD_ACCESS_ROLES = ['admin', 'editor',
'moderator'] as const satisfies ReadonlyArray<UserRole>;

```

Functions

Camel case

`filterProductsByType`, `formatCurrency`

Types

Pascal case

`OrderStatus`, `ProductItem`



Rule

Generics

A generic type parameter must start with the capital letter T followed by a descriptive name `TRequest`, `TFooBar`.

Key reasons and benefits:

- Complex types often involve generics, where clear naming improves readability and maintainability.
- Single letter generics like `T`, `K`, `U` are disallowed, the more parameters we introduce, the easier it is to mistake them.
- Prefixing with `T` makes it immediately obvious that it's a generic type parameter, not a regular type.
- A common scenario is when a generic parameter shadows an existing type due to having the same name e.g. `<Request extends Request>`

Rule

```
// ✗ Avoid naming generic parameters with one letter
const createPair = <T, K extends string>(first: T, second: K):
[T, K] => {
  return [first, second];
};
const pair = createPair(1, 'a');

// ✓ Use descriptive names starting with capital T
const createPair = <TFirst, TSecond extends string>(first:
TFirst, second: TSecond): [TFirst, TSecond] => {
  return [first, second];
};
const pair = createPair(1, 'a');

// ✗ Avoid naming generic parameters without a prefix - which
'Request' is which?
const handle = <Request extends Request>(req: Request): void
=> { ...

// ✓ Prefix generic parameter with capital T
const handle = <TRequest extends Request>(req: TRequest): void
=> { ...
```

Abbreviations & Acronyms

Treat acronyms as whole words, with capitalized first letter only.

```
// ✗ Avoid
const FAQList = ['qa-1', 'qa-2'];
const generateUserURL(params) => { ... }

// ✓ Use
```

```
const FaqList = ['qa-1', 'qa-2'];
const generateUserUrl(params) => {....}
```

In favor of readability, strive to avoid abbreviations, unless they are widely accepted and necessary.

```
// ✗ Avoid
const GetWin(params) => {....}

// ✅ Use
const GetWindow(params) => {....}
```

React Components

Pascal case

`ProductItem`, `ProductsPage`

Prop Types

React component name following "Props" postfix

`[ComponentName]Props` - `ProductItemProps`, `ProductsPageProps`

Callback Props

Event handler (callback) props are prefixed as `on*` - e.g. `onClick`.

Event handler implementation functions are prefixed as `handle*` - e.g. `handleClick`.

Rule

```
// ✗ Avoid inconsistent callback prop naming
<Button click={actionClick} />
<MyComponent userSelected0ccurred={triggerUser} />

// ✅ Use prop prefix 'on*' and handler prefix 'handle*'
<Button onClick={handleClick} />
<MyComponent onUserSelected={handleUserSelected} />
```

React Hooks

Camel case, prefixed as 'use' 

Symmetrically convention as `[value, setValue] = useState()` 

```
// ✗ Avoid inconsistent useState hook naming
const [userName, setUser] = useState();
```

```
const [color, updateColor] = useState();
const [isActive, setActive] = useState();

// ✅ Use
const [name, setName] = useState();
const [color, setColor] = useState();
const [isActive, setIsActive] = useState();
```

Custom hook must always return an object

```
// ✗ Avoid
const [products, errors] = useGetProducts();
const [fontSizes] = useTheme();

// ✅ Use
const { products, errors } = useGetProducts();
const { fontSizes } = useTheme();
```

Comments

Comments can quickly become outdated, leading to confusion rather than clarity.

Favor expressive code over comments by using meaningful names and clear logic.
Comments should primarily explain "why," not "what" or "how."

Use comments when:

- The context or reasoning isn't obvious from the code alone (e.g. config files, workarounds)
- Referencing related issues, PRs, or planned improvements

```
// ✗ Avoid
// convert to minutes
const m = s * 60;
// avg users per minute
const myAvg = u / m;

// ✅ Use - Prefer expressive code by naming things what they are
const SECONDS_IN_MINUTE = 60;
const minutes = seconds * SECONDS_IN_MINUTE;
const averageUsersPerMinute = noOfUsers / minutes;

// ✅ Use - Reference planned improvements
// TODO: Move filtering to the backend once API v2 is
```

```
released.
// Issue/PR – https://github.com/foo/repo/pulls/55124
const filteredUsers = frontendFiltering(selectedUsers);

// ✓ Use – Add context to explain why
// Use Fourier transformation to minimize information loss –
https://github.com/dntj/jsfft#usage
const frequencies = signal.FFT();
```

TSDoc Comments

TSDoc comments enhance code maintainability and developer experience by providing structured documentation that IDEs, TypeScript, and API tools (e.g., Swagger, OpenAPI, GraphQL) can interpret.

Use TSDoc comments when documenting APIs, libraries, configurations or reusable code.

```
/** 
 * Configuration options for the Web3 SDK.
 */
export type Web3Config = {
  /** Ethereum network chain ID. */
  chainId: number;

  /**
   * Gas price strategy for transactions:
   * - `fast`: Higher fees, faster confirmation
   * - `standard`: Balanced
   * - `slow`: Lower fees, slower confirmation
   */
  gasPriceStrategy: 'fast' | 'standard' | 'slow';

  /** Maximum gas limit per transaction. */
  maxGasLimit?: number;

  /**
   * Enables event listening for smart contract interactions.
   */
  enableEventListener?: boolean;
};
```

Source Organization

Code Collocation

- Every application or package in monorepo has project files/folders organized and grouped by **feature**.
- **Collocate code as close as possible to where it's relevant.**
- Deep folder nesting should not represent an issue.

Imports

Import paths can be relative, starting with `./` or `../`, or they can be absolute `@common/utils`.

To make import statements more readable and easier to understand:

- **Relative** imports `./sortItems` must be used when importing files within the same feature, that are 'close' to each other, which also allows moving feature around the codebase without introducing changes in these imports.
- **Absolute** imports `@common/utils` must be used in all other cases.
- **All** imports must be auto sorted by tooling e.g. `prettier-plugin-sort-imports`, `eslint-plugin-import` etc.

```
// ✗ Avoid
import { bar, foo } from '../../../../../distant-folder';

// ✅ Use
import { locationApi } from '@api/locationApi';

import { foo } from '../foo';
import { bar } from '../bar';
import { baz } from './baz';
```

Project Structure

Example frontend monorepo project, where every application has file/folder grouped by feature:

```
apps/
  |- product-manager/
    |- common/
      |- components/
        |- Button/
        |- ProductTitle/
        ...
        |- index.tsx
      |- consts/
        |- paths.ts
```

```

    |   |   └ ...
    |   └ hooks/
    └ types/
    └ modules/
        └ HomePage/
        └ ProductAddPage/
        └ ProductPage/
        └ ProductsPage/
            └ api/
                └ useGetProducts/
            └ components/
                └ ProductItem/
                └ ProductsStatistics/
                └ ...
            └ utils/
                └ filterProductsByType/
        └ index.tsx
    └ ...
    └ index.tsx
    └ eslint.config.mjs
    └ package.json
    └ tsconfig.json
    └ warehouse/
    └ admin-dashboard/
    └ ...

```

- `modules` folder is responsible for implementation of each individual page, where all custom features for that page are being implemented (components, hooks, utils functions etc.).
- `common` folder is responsible for implementations that are truly used across application. Since it's a "global folder" it should be used sparingly.
If same component e.g. `common/components/ProductTitle` starts being used on more than one page, it shall be moved to common folder.

In case using frontend framework with file-system based router (e.g. Nextjs), `pages` folder serves only as a router, where its responsibility is to define routes (no business logic implementation).

Example backend project structure with file/folder grouped by feature:

```

product-manager/
└ dist/
    └ database/
        └ migrations/

```

```
|- 20220102063048_create_accounts.ts
|   ...
|- seeders/
|   |- 20221116042655-feeds.ts
|   ...
|- docker/
|- logs/
|- scripts/
|- src/
|   |- common/
|   |   |- consts/
|   |   |- middleware/
|   |   |- types/
|   |   ...
|   |- dao/
|   |   |- user/
|   |   ...
|   |- modules/
|   |   |- admin/
|   |   |   |- account/
|   |   |   |   |- account.model.ts
|   |   |   |   |- account.controller.ts
|   |   |   |   |- account.route.ts
|   |   |   |   |- account.service.ts
|   |   |   |   |- account.validation.ts
|   |   |   |   |- account.test.ts
|   |   |   |- index.ts
|   |   ...
|   |   |- general/
|   |   |   |- general.model.ts
|   |   |   |- general.controller.ts
|   |   |   |- general.route.ts
|   |   |   |- general.service.ts
|   |   |   |- general.validation.ts
|   |   |   |- general.test.ts
|   |   |- index.ts
|   |
|   |- ...
|   |- index.tsx
|
|- ...
|- eslint.config.mjs
|- package.json
|- tsconfig.json
```

Appendix - React

Since React components and hooks are also functions, respective **function conventions** applies.

Required & Optional Props

Strive to have majority of props required and use optional props sparingly.

Especially when creating new component for first/single use case majority of props should be required. When component starts covering more use cases, introduce optional props.

There are potential exceptions, where component API needs to implement optional props from the start (e.g. shared components covering multiple use cases, UI design system components - button `isDisabled` etc.)

If component/hook becomes to complex it probably should be broken into smaller pieces.

An exaggerated example where implementing 10 React components with 5 required props each, is better then implementing one "can do it all" component that accepts 50 optional props.

Props as Discriminated Type

When applicable, use **discriminated types** to eliminate optional props. This approach reduces complexity in the component API and ensures that only the required props are passed based on the specific use case.

```
// ✗ Avoid optional props as they increase complexity and
// ambiguity in component APIs
type StatusProps = {
  data?: Products;
  title?: string;
  time?: number;
  error?: string;
};

// ✓ Prefer required props. If optional props are
// unavoidable,
// use a discriminated union to represent distinct use cases
// with required props.
type StatusSuccess = {
  status: 'success';
  data: Products;
  title: string;
};
```

```

type StatusLoading = {
  status: 'loading';
  time: number;
};

type StatusError = {
  status: 'error';
  error: string;
};

// Discriminated union 'StatusProps' ensures predictable
// component props with no optionals
type StatusProps = StatusSuccess | StatusLoading |
StatusError;

export const Status = (status: StatusProps) => {
  switch (props.status) {
    case 'success':
      return <div>Title {props.title}</div>;
    case 'loading':
      return <div>Loading {props.time}</div>;
    case 'error':
      return <div>Error {props.error}</div>;
  }
};

```

Props To State

In general avoid using props to state, since component will not update on prop changes. It can lead to bugs that are hard to track, with unintended side effects and difficulty testing.

When there is truly a use case for using prop in initial state, prop must be prefixed with `initial` (e.g. `initialProduct`, `initialSort` etc.)

```

// ✗ Avoid using props to state
type FooProps = {
  productName: string;
  userId: string;
};

export const Foo = ({ productName, userId }: FooProps) => {
  const [productName, setProductName] = useState(productName);
  ...

// ✅ Use prop prefix `initial`, when there is a rational use

```

```

case for it
type FooProps = {
  initialProductName: string;
  userId: string;
};

export const Foo = ({ initialProductName, userId }: FooProps) => {
  const [productName, setProductName] =
useState(initialProductName);
  ...
}

```

Props Type

```

// ✗ Avoid using React.FC type
type FooProps = {
  name: string;
  score: number;
};

export const Foo: React.FC<FooProps> = ({ name, score }) => {

// ✓ Use props argument with type
type FooProps = {
  name: string;
  score: number;
};

export const Foo = ({ name, score }: FooProps) => { ...
}

```

Component Types

Container

- All container components have postfix "Container" or "Page" `[ComponentName]Container|Page`. Use "Page" postfix to indicate component is an actual web page.
- Each feature has a container component (`AddUserContainer.tsx`, `EditProductContainer.tsx`, `ProductsPage.tsx` etc.)
- Includes business logic.
- API integration.
- Structure:

```

ProductsPage/
└─ api/

```

```
  └ useGetProducts/  
  └ components/  
    └ ProductItem/  
  └ utils/  
    └ filterProductsByType/  
  └ index.tsx
```

UI - Feature

- Representational components that are designed to fulfill feature requirements.
- Nested inside container component folder.
- Should follow **functions** conventions as much as possible.
- No API integration.
- Structure:

```
ProductItem/  
└ index.tsx  
└ ProductItem.stories.tsx  
└ ProductItem.test.tsx
```

UI - Design system

- Global Reusable/shared components used throughout whole codebase.
- Structure:

```
Button/  
└ index.tsx  
└ Button.stories.tsx  
└ Button.test.tsx
```

Store & Pass Data

- Pass only the necessary props to child components rather than passing the entire object.
- Utilize storing state in the URL, especially for filtering, sorting etc.
- Don't sync URL state with local state.
- Consider passing data simply through props, using the URL, or composing children. Use global state (Zustand, Context) as a last resort.
- Use React compound components when components should belong and work together: **menu**, **accordion**, **navigation**, **tabs**, **list**, etc.
Always export compound components as:

```
// PriceList.tsx
const PriceListRoot = ({ children }) => <ul>{children}
</ul>;
const PriceListItem = ({ title, amount }) => <li>Name: {name} - Amount: {amount}</li>;

// ✗
export const PriceList = {
  Container: PriceListRoot,
  Item: PriceListItem,
};

// ✗
PriceList.Item = Item;
export default PriceList;

// ✅
export const PriceList = PriceListRoot as typeof
PriceListRoot & {
  Item: typeof PriceListItem;
};
PriceList.Item = PriceListItem;

// App.tsx
import { PriceList } from "./PriceList";

<PriceList>
  <PriceList.Item title="Item 1" amount={8} />
  <PriceList.Item title="Item 2" amount={12} />
</PriceList>
```

- UI components should show derived state and send events, nothing more (no business logic).
- As in many programming languages functions args can be passed to the next function and on to the next etc.
React components are no different, where prop drilling should not become an issue.
If with app scaling prop drilling truly becomes an issue, try to refactor render method, local states in parent components, using composition etc.
- Data fetching is only allowed in container components.
- Use of server-state library is encouraged ([react-query](#), [apollo client](#) etc.).
- Use of client-state library for global state is discouraged.
Reconsider if something should be truly global across application, e.g.

`themeMode`, `Permissions` or even that can be put in server-state (e.g. user settings - `/me` endpoint). If still global state is truly needed use [Zustand](#) or [Context](#).

Appendix - Tests

What & How To Test

Automated test comes with benefits that helps us write better code and makes it easy to refactor, while bugs are caught earlier in the process.

Consider trade-offs of what and how to test to achieve confidence application is working as intended, while writing and maintaining tests doesn't slow the team down.

 Do:

- Implement test to be short, explicit, and pleasant to work with. Intent of a test should be immediately visible.
- Strive for AAA pattern, to maintain clean, organized, and understandable unit tests.
 - Arrange - Setup preconditions or the initial state necessary for the test case. Create necessary objects and define input values.
 - Act - Perform the action you want to unit test (invoke a method, triggering an event etc.). **Strive for minimal number of actions.**
 - Assert - Validate the outcome against expectations. **Strive for minimal number of asserts.**

A rule "unit tests should fail for exactly one reason" doesn't need to apply always, but it can indicate a code smell if there are tests with many asserts in a codebase.

- As mentioned in [function conventions](#) try to keep them pure, and impure one small and focused.

It makes them easy to test, by passing args and observing return values, since we will **rarely need to mock dependencies**.

- Strive to write tests in a way your app is used by a user, meaning test business logic.

E.g. For a specific user role or permission, given some input, we receive the expected output from the process.

- Make tests as isolated as possible, where they don't depend on order of execution and should run independently with its own local storage, session storage, data, cookies etc. Test isolation speeds up the test run, improves reproducibility, makes debugging easier and prevents cascading test failures.

- Tests should be resilient to changes.
 - Black box testing - Always test only implementation that is publicly exposed, don't write fragile tests on how implementation works internally.
 - Query HTML elements based on attributes that are unlikely to change. Order of priority must be followed as specified in [Testing Library - role, label, placeholder, text contents, display value, alt text, title, test ID](#).
 - If testing with a database then make sure you control the data. If test are run against a staging environment make sure it doesn't change.

✖ Don't:

- Don't test implementation details. When refactoring code, tests shouldn't change.
- Don't re-test the library/framework.
- Don't mandate 100% code coverage for applications.
- Don't test third-party dependencies. Only test what your team controls (package, API, microservice etc.). Don't test external sites links, third party servers, packages etc.
- Don't test just to test.

```
// ✖ Avoid
it('should render the user list', () => {
  render(<UserList />);
  expect(screen.getByText('Users
List')).toBeInTheDocument();
});
```

Test Description

All test descriptions must follow naming convention as `it('should ... when ...')`.



```
// ✖ Avoid
it('accepts ISO date format where date is parsed and formatted
as YYYY-MM');
it('after title is confirmed user description is rendered');

// ✅ Name test description as it('should ... when ...')
it('should return parsed date as YYYY-MM when input is in ISO
```

```
date format');
it('should render user description when title is confirmed');
```

Test Tooling

Besides running tests through scripts, it's highly encouraged to use [Vitest Runner](#) and [Playwright Test](#) VS code extension alongside.

With extension any single [unit/integration](#) or [E2E](#) test can be run instantly, especially if testing app or package in larger monorepo codebase.

```
code --install-extension vitest.explorer
code --install-extension ms-playwright.playwright
```

Snapshot

Snapshot tests are discouraged to avoid fragility, which leads to a "just update it" mindset to make all tests pass.

Exceptions can be made, with strong rationale behind them, where test output has short and clear intent about what's actually being tested (e.g., design system library critical elements that shouldn't deviate).