TypeScript Style Guide

Introduction

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

Table of Contents

Expand

About Guide

What

Since "consistency is the key", TypeScript Style Guide strives to enforce majority of the rules by using automated tooling as ESLint, TypeScript, Prettier, etc.
Still certain design and architectural decisions must be followed which are described with 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 way to write TypeScript applications brings a consistent codebase and faster development cycles.
- No need to discuss code styles in code reviews.
- Saves team time and energy.

Disclaimer

As any code style guide is opinionated, this is no different as it tries to set conventions (sometimes arbitrary) that govern our code.

You don't have to follow every convention exactly as it is written in guide, decide what works best for your product and team to stay consistent with your codebase.

Requirements

Style Guide requires you to use:

- TypeScript v5
- typescript-eslint v7 with `strict-type-checked` configuration enabled.

Style Guide assumes using, but is not limited to:

- React as UI library for frontend conventions.
- Playwright and Testing Library for testing conventions.

TLDR

- Embrace const assertions. ↓
- Strive for data immutability (`Readonly`, `ReadonlyArray`). ↓
- Embrace discriminated unions. ↓
- Majority of function arguments should be required (use optional sparingly). ↓
- Avoid type assertions. ↓
- Strive for functions to be pure, stateless and have single responsibility. ↓
- Strong emphasis to keep naming conventions consistent and readable. ↓
- Use named exports. ↓
- Code is organized and grouped by feature. Collocate code as close as possible to where it's relevant. ↓

Types

When creating types we try to think of how they would best **describe our code**. Being expressive and keeping types as **narrow as possible** brings benefits to the codebase:

- Increased Type Safety - Catch errors at compile-time, since narrowed types provide more specific information about the shape and behavior of your data.
- Improved Code Clarity - Cognitive load is reduced by providing clearer boundaries and constraints on your data which makes your code easier to understand by other developers.
- Easier Refactoring - Refactor with confidence, since types are narrow, making changes to your code becomes less risky.
- Optimized Performance - In some cases, narrow types can help the TypeScript compiler generate more optimized JavaScript code.

Type Inference

As rule of thumb, explicitly declare a type when it help narrows it.

Note ⓘ

```
// ✗ Avoid - Don't explicitly declare a type, it can be inferred.
const userRole: string = 'admin'; // Type 'string'
const employees = new Map<string, number>([['Gabriel', 32]]);
const [isActive, setIsActive] = useState<boolean>(false);
```

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


// ❌ Avoid – Don't infer a (wide) type, it can be narrowed.
const employees = new Map(); // Type 'Map<any, any>'
employees.set('Lea', 17);
type UserRole = 'admin' | 'guest';
const [userRole, setUserRole] = useState('admin'); // Type
'string'

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

Data Immutability

Majority of the data should be immutable with use of `Readonly`, `ReadonlyArray`.

Using readonly type prevents accidental data mutations, which reduces the risk of introducing bugs related to unintended side effects.

When performing data processing always return new array, object etc. To keep cognitive load for future developers low, try to keep data objects small.
As an exception mutations should be used sparingly in cases where truly necessary: complex objects, performance reasoning etc.

```
// ❌ 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'
};
```

Return Types

Including return type annotations is highly encouraged, although not required (eslint rule).

Consider benefits when explicitly typing the return value of a function:

- Return values makes it clear and easy to understand to any calling code what type is returned.
- In the case where there is no return value, the calling code doesn't try to use the undefined value when it shouldn't.
- Surface potential type errors faster in the future if there are code changes that change the return type of the function.
- Easier to refactor, since it ensures that the return value is assigned to a variable of the correct type.
- Similar to writing tests before implementation (TDD), defining function arguments and return type, gives you the opportunity to discuss the feature functionality and its interface ahead of implementation.
- Although type inference is very convenient, adding return types can save TypeScript compiler a lot of work.

Discriminated union

If there is only one TypeScript feature to choose, 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 Args as Discriminated Union, discriminated union eliminates optional args which decreases complexity on function API.

- Exhaustiveness check - TypeScript can ensure that all possible variants of a type are implemented, eliminating the risk of undefined or unexpected behavior at runtime. (eslint rule)

```
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.

Template Literal Types

Embrace using template literal types, instead of just (wide) `string` type. Template literal types have many applicable use cases e.g. API endpoints, routing, internationalization, database queries, CSS typings ...

```
// ❌ Avoid
const userEndpoint = '/api/usersss'; // Type 'string' - Since typo 'usersss', route doesn't exist and results in 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';

// ❌ Avoid
const homeTitle = 'translation.homesss.title'; // Type 'string' - Since typo 'homesss', translation doesn't exist and results in 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';

// ❌ Avoid
const color = 'blue-450'; // Type 'string' - Since color 'blue-450' doesn't exist and results in 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';
```

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 strong rational why being introduced into 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 Error

If TypeScript error can't be mitigated, as last resort use `@ts-expect-error` to suppress it (eslint rule). If at any future point suppressed line becomes error-free, TypeScript compiler will indicate it.
`@ts-ignore` is not allowed, where `@ts-expect-error` must be used with provided description (eslint rule).

```
// ❌ Avoid @ts-ignore
// @ts-ignore
const newUser = createUser('Gabriel');

// ✅ Use @ts-expect-error with description
// @ts-expect-error: The library type definition is wrong,
createUser accepts string as an argument.
const newUser = createUser('Gabriel');
```

Type Definition

TypeScript offers two options for type definitions - `type` and `interface`. As they come with some functional differences in most cases they can be used interchangeably. We try to limit syntax difference and pick one for consistency.

All types must be defined with `type` alias (eslint rule).

Note ⓘ

```
// ❌ Avoid interface definitions
interface UserRole = 'admin' | 'guest'; // invalid - interface
can't define (commonly used) type unions

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

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

type UserInfo = {
  name: string;
```

```
  role: UserRole;
};
```

In case of declaration merging (e.g. extending third-party library types) use `interface` and disable lint rule.

```
// 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 must be defined with generic syntax (eslint 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'];
```

Services

Documentation becomes outdated the moment it's written, and worse than no documentation is wrong documentation. The same can be said about types when describing modules your app interacts with (APIs, messaging systems, databases).

For external API services e.g. REST, GraphQL etc. it's crucial to generate types from their contracts, whether it's Swagger, schemas, or other sources (e.g. openapi-ts,

graphql-config...). Avoid manually declaring and maintaining them, as it's easy to get them out of sync.

As exception manually declare types only when their is truly no documentation provided by external service.

Functions

Function conventions should be followed as much as possible (some of the conventions derives 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. It's 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 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 function becomes to complex it probably should be broken into smaller pieces. An exaggerated example where implementing 10 functions with 5 required args each, is better then implementing one "can do it all" function that accepts 50 optional args.

Args as Discriminated Union

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

```
// ❌ Avoid optional args as they increase complexity of function API
type StatusParams = {
  data?: Products;
  title?: string;
  time?: number;
  error?: string;
};

// ✅ Strive to have majority of args required, if that's not possible,
// use discriminated union for clear intent on function usage
type StatusSuccessParams = {
  status: 'success';
  data: Products;
  title: string;
};

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

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

type StatusParams = StatusSuccessParams | StatusLoadingParams | StatusErrorParams;
```

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

Variables

Const Assertion

Strive to use const assertion `as const`:

- type is narrowed
- object gets `readonly` properties
- array becomes `readonly` tuple

```
// ❌ Avoid declaring constants without const assertion
const FOO_LOCATION = { x: 50, y: 130 }; // Type { x:
number; y: number; }
FOO_LOCATION.x = 10;

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

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



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

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

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

Const assertion must be used over enum.

While enums can still cover use cases as const assertion would, we tend to avoid it. Some of the reasonings as mentioned in TypeScript documentation - Const enum pitfalls, Objects vs Enums, Reverse mappings...

```
// .eslintrc.js
'no-restricted-syntax': [
  'error',
  {
    selector: 'TSEnumDeclaration',
    message: 'Use const assertion or a string union type
instead.',
  },
],
```

```
// ❌ Avoid using enums
enum UserRole {
  GUEST,
  MODERATOR,
  ADMINISTRATOR,
}

enum Color {
  PRIMARY = '#B33930',
  SECONDARY = '#113A5C',
  BRAND = '#9C0E7D',
}

// ✅ Use const assertion
const USER_ROLES = ['guest', 'moderator', 'administrator'] as
const;
type UserRole = (typeof USER_ROLES)[number]; // Type "guest" |
"moderator" | "administrator"

// Use satisfies if UserRole type is already defined - e.g.
database schema model
type UserRoleDB = ReadonlyArray<'guest' | 'moderator' |
'administrator'>;
const AVAILABLE_ROLES = ['guest', 'moderator'] as const
satisfies UserRoleDB;

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

```
type Color = typeof COLOR;
type ColorKey = keyof Color; // Type "PRIMARY" | "SECONDARY" |
"BRAND"
type ColorValue = Color[ColorKey]; // Type "#B33930" |
"#113A5C" | "#9C0E7D"
```

Type Union & Boolean Flags

Strive to use type union variable instead multiple boolean flag variables.

Flags tend to compound over time and become hard to maintain, 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 maintaining code that makes the number of possible states grows quickly because use of flags, there are almost always unhandled states, utilize discriminated union.

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)...

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 (eslint rule).
This keeps variables, functions... 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.

In case of exceptions e.g. Next.js pages, disable rule:

```
// .eslintrc.js
overrides: [
  {
    files: ["src/pages/**/*"],
    rules: { "import/no-default-export": "off" },
  },
],
```

Naming Conventions

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

Variables

- **Locals**
  Camel case
  `products`, `productsFiltered`
- **Booleans**
  Prefixed with `is`, `has` etc.
  `isDisabled`, `hasProduct`

  Eslint rule implements:

```
// .eslintrc.js
'@typescript-eslint/naming-convention': [
  'error',
  {
    selector: 'variable',
    types: ['boolean'],
    format: ['PascalCase'],
    prefix: ['is', 'should', 'has', 'can', 'did', 'will'],
  },
],
```

- **Constants**

  Capitalized
  `PRODUCT_ID`

- **Object constants**

  Singular, capitalized with const assertion.

```
const ORDER_STATUS = {
  pending: 'pending',
  fulfilled: true,
  error: 'Shipping Error',
} as const;
```

  If object type exist use `satisfies` operator in conjunction with const assertion, to conform object matches its type.

```
// OrderStatus type is already defined - e.g. generated from database schema model, API etc.
type OrderStatus = {
  pending: 'pending' | 'idle';
  fulfilled: boolean;
  error: string;
};

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

Functions

Camel case
`filterProductsByType`, `formatCurrency`

Types

Pascal case
`OrderStatus`, `ProductItem`

Eslint rule implements:

```
// .eslintrc.js
'@typescript-eslint/naming-convention': [
```

```
  'error',
  {
    selector: 'typeAlias',
    format: ['PascalCase'],
  },
],
```

Generics

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

Creating more complex types often include generics, which can make them hard to read and understand, that's why we try to put best effort when naming them. Naming generics using popular convention with one letter `T`, `K` etc. is not allowed, the more variables we introduce, the easier it is to mistake them.

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

// ✅ Name starts with the capital letter T followed by a
descriptive name
const createPair = <TFirst, TSecond extends string>(first:
TFirst, second: TSecond): [TFirst, TSecond] => {
  return [first, second];
};
const pair = createPair(1, 'a');
```

Eslint rule implements:

```
// .eslintrc.js
'@typescript-eslint/naming-convention': [
  'error',
  {
    // Generic type parameter must start with letter T,
followed by any uppercase letter.
    selector: 'typeParameter',
    format: ['PascalCase'],
    custom: { regex: '^T[A-Z]', match: true },
```

```
  },
],
```

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` (eslint rule).

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

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

React Hooks

Camel case, prefixed as 'use' (eslint rule), symmetrically convention as `[value, setValue] = useState()` (eslint rule)

```
// ❌ 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

In general try to avoid comments, by writing expressive code and name things what they are.

Use comments when you need to add context or explain choices that cannot be expressed through code (e.g. config files).
Comments should always be complete sentences. As rule of thumb try to explain `why` in comments, not `how` and `what`.

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

```
const myAvg = u / m;

// ✅ Use - Expressive code and name things what they are
const SECONDS_IN_MINUTE = 60;
const minutes = seconds * SECONDS_IN_MINUTE;
const averageUsersPerMinute = noOfUsers / minutes;

// ✅ Use - Add context to explain why in comments
// TODO: Filtering should be moved to the backend once API changes are released.
// Issue/PR - https://github.com/foo/repo/pulls/55124
const filteredUsers = frontendFiltering(selectedUsers);
// Use Fourier transformation to minimize information loss - https://github.com/dntj/jsfft#usage
const frequencies = signal.FFT();
```

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...

```
// ❌ 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
│  ├ eslintrc.js
│  ├ 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/
│  │  └─ ...
│  ├─ 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
│   └── ...
├── ...
├── eslintrc.js
├── 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 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 type** to eliminate optional props, which will decrease complexity on component API and only necessary/required props will be passed depending on its use case.

```
// ✗ Avoid optional props as they increase complexity of component API
type StatusProps = {
```

```
  data?: Products;
  title?: string;
  time?: number;
  error?: string;
};

// ✅ Strive to have majority of props required, if that's not possible,
// use discriminated union for clear intent on component usage
type StatusSuccess = {
  status: 'success';
  data: Products;
  title: string;
};

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

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

type StatusProps = StatusSuccess | StatusLoading | StatusError;

export const Status = (status: StatusProps) => {...
```

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`,...
  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...).
- 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 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 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 ...')`.

Eslint rule implements regex:

```
// .eslintrc.js
'jest/valid-title': [
  'error',
  {
    mustMatch: { it: [/should.*when/u.source, "Test title must
include 'should' and '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

Tests can be run through scripts, but it's highly encouraged to use Jest 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 firsttris.vscode-jest-runner
code --install-extension ms-playwright.playwright
```

Snapshot

Snapshot tests are discouraged in order to avoid fragility, which leads to "just update it" turn of mind, to achieve all the tests pass.
Exceptions can be made, with strong rational behind it, where test output has short and clear intent, what's actually being tested (e.g. design system library critical elements that shouldn't deviate).