TypeScript

Last updated by | Daniels, Steve | Oct 23, 2025 at 7:02 PM GMT+5:30

Contents

📖 Best Practices

- TypeScript: Handbook - The TypeScript Handbook The Jest Object · Jest

🖌️ Linting and Formatting

- We use the following for linting and formatting:
  - eslint - npm
  - prettier - npm

🏷️ Naming Conventions

🧠 TypeScript Naming Conventions (from ESLint config)

✅ Interface Naming

- Use **PascalCase** and prefix with `I`:
  interface IUserProfile { ... }
  This helps distinguish interfaces from other types and classes.

✅ Type Alias Naming

- Use **PascalCase** and prefix with `T`:
  type TUserData = { ... }
  This makes type aliases easily identifiable and separate from interfaces.

✅ Variable & Function Naming

- Use **camelCase** for variables and functions:
  const userName = 'Steve';function fetchData() { ... }
  Avoid abbreviations like `btnTxt`; prefer `buttonText` for clarity.

✅ File Naming

- Use **camelCase** for filenames:
  userProfile.tsloginForm.ts
  Enforced by `unicorn/filename-case`.

✅ Import Ordering

- Use grouped and alphabetised imports with newlines between groups:
  import type { TUserData } from './types';import fs from 'fs';import express from 'express';import { getUser } from '../services';import { logger } from './logger';
  Controlled by `import/order`.

✅ Type Imports

- Prefer **separate type imports** using `type` keyword:
  `import type { TUserData } from './types';`
  Enforced by `@typescript-eslint/consistent-type-imports`.

✅ Function Return Types

- Explicitly define return types for functions:
  `function getUser(): TUserData { ... }`
  Warned by `@typescript-eslint/explicit-function-return-type`.

❌ What to Avoid

- Avoid **abbreviations** like `req`, `res`, `params`, `args` — though `unicorn/prevent-abbreviations` is disabled to allow common Express patterns.
- Avoid **empty object types** — though `@typescript-eslint/no-empty-object-type` is turned off.
- Avoid **filename casing inconsistencies** — enforced to be camelCase.

📝 Commenting & Documentation

- If documentation in code is deemed necessary, it **should** conform to TSDoc

〓 Functional Approach

✅ Function Declarations over Arrow Functions

Prefer:

```
function calculateTotal(price: number, tax: number): number {  return price + tax;}``
```

Avoid:

```
const calculateTotal = (price: number, tax: number): number => price + tax;
```

Why prefer `function` syntax?

- Named functions improve stack traces and debugging.
- Easier to hoist and organise code.
- More readable in larger codebases and consistent with traditional FP libraries.

✅ Pure Functions

Functions should:

- Not mutate external state.

- Return the same output for the same input.
  function double(x: number): number { return x * 2;}

✅ Immutability

Use `const` and avoid mutating objects or arrays directly.
const updatedList = [...originalList, newItem];
Use libraries like `immer` or `fp-ts` for deeper immutability patterns.

✅ Higher-Order Functions

Functions that take or return other functions:
function withLogging(fn: (x: number) => number): (x: number) => number { return function (x: number): number { console.log('Input:', x); return fn(x); };}``

✅ Declarative over Imperative

Prefer:
const activeUsers = users.filter(isActive).map(toUserSummary); `Over: const activeUsers: UserSummary[] = [];for (const user of users) { if (user.active) { activeUsers.push(toUserSummary(user)); }}`

✅ Type Safety

Use type aliases and interfaces to clarify function inputs and outputs:

```
type TUser = { name: string; age: number };

function greet(user: TUser): string {
  return `Hello, ${user.name}`;
}
```

Type Assertions

- Must only use the `as` keyword in exceptional cases - likely it is highlighting a gap in the typing implementation or poor design

⛔ Error Handling

⌨️ Try..Catch

- Any code that is not a pure function must have a `try ... catch` block
- All code that has a `async` must have a `try ... catch` block
- `try ... catch` must not be nested
- `try ... catch` should contain all logic within a given function

✔️ Error type assertions

- Starting with TypeScript v4, the `error` property in a `catch` block is typed as `unknown` [1]. Developers **must** therefore assert the type of the `error` before attempting to access properties it may or may not have.

🐥 This doesn't work

```
function someFunc() {
  try {
    // Code that may throw an error
  } catch (error) {
    console.error(error.message)
    // TypeScript Error: 'error' is of type 'unknown'
  }
}
```

There are two different approaches to asserting the type of the error:

`instaceof` the `Error` class

For scenarios where the thrown error is an instance of the built-in `Error` class, a simple `instanceof` assertion should be sufficient, e.g.

```
function someFunc() {
  try {
    // Code that may throw an error
  } catch (error) {
    if (error instanceof Error) {
      console.error(error.message) // ✅ Works
      // Handle error further
      return
    }
    console.error('Unexpected error type', error)
  }
}
```

For errors that are instances of a custom class that has extra props, `instanceof` is also a valid assertion, e.g.

```
function someFunc() {
  try {
    // Code that may throw an error
  } catch (error) {
    if (error instanceof CustomError) {
      console.error(error.customProp) // ✅ Works
      // Handle error further
      return
    }
    console.error('Unexpected error type', error)
  }
}
```

⚠️ Gotcha

Be careful - if a custom error class extends the default `Error` class, the order of the if statements matters and can lead to unreachable conditions.

```
class CustomError extends Error {
  customProp: string

  constructor({ customProp, ...props }) {
    this.customProp = customProp
    super(props)
  }
}

function someFunc() {
  try {
    // Code that may throw an error
  } catch (error) {
    if (error instanceof Error) {
      console.error(error.message) // ✅ Works
      // Handle error further
      return
    } else if (error instanceof CustomError) {
      console.error(error.customProp) // ❌ Unreachable code, as CustomError is also instance of Error
      // Handle error further
      return
    }
    console.error('Unexpected error type', error)
  }
}
```

Using type predicate functions

In some cases, an object is thrown which is *not* an instance of a class and it may therefore be necessary to assert on properties within the object directly. This is a common occurence with third party libraries which are written usuing a purely functional approach.

Consider the following example:

```
import { someLibraryFunction } from 'some-library'

async function someFunc() {
  try {
    // ...
    await someLibraryFunction({ some: 'thing' }) // throws CustomErrorType
  } catch (error) {
    if (error instanceof CustomErrorType) {
      // ❌ Does not work, as CustomErrorType is not a class
      console.error(error.customProp)
      return
    }
    console.error('Unexpected error type', error)
  }
}
```

Writing assertions can quickly become tedious if the same assertions have to be made repeatedly on the object. We can use a type predicate function to assert that something is valid as a type, e.g.

```typescript
// If a library has type definitions, use them to help
import type { CustomErrorType } from 'some-library'

import { someLibraryFunction } from 'some-library'

// In reality, this function might exist in another file, but is included here for clarity
export function isCustomErrorType(error: unknown): error is CustomErrorType {
  return (
    error !== undefined &&
    typeof error === 'object' &&
    error !== null &&
    'customProp' in error &&
    typeof error.customProp === 'string'
  )
}

async function someFunc() {
  try {
    // ...
    await someLibraryFunction({ some: 'thing' }) // throws CustomErrorType
  } catch (error) {
    if (isCustomErrorType(error)) {
      console.error(error.customProp) // ✅ Works, customProp is now accessible
      // Handle error further
      return
    }
    console.error('Unexpected error type', error)
  }
}
```

Consult the TypeScript documentation for more information on Using Type Predicates

👽 3rd Party Libraries

When considering whether to use a 3rd party to achieve something, a developer **should** consider if they could achieve the same functionality themselves using native functionality for the platform they are developing for.

Consider the following common examples:

- axios - A promise-basedlibrary for making HTTP requests in the browser and Node.
  - Node has a native implementation of the fetch API based on Undici
  - Most modern browsers implement the fetch API natively
- date-fns - A library for manipulating Dates and Times
  - The JavaScript-native Date class usually has enough functionality to be used on it's own
- lodash - A set of utility functions for manipulating objects and arrays
  - Modern ECMAScript standards include array and object methods which can achieve the same effects as lodash utilities.

In general, you won't see libraries such as those above in new code. Developers **should** avoid adding new dependencies without consultation with a senior or lead developer.

Usage of any new third-party library as a production dependency is subject to security review before it can be used.

📄 Licensing

- Must follow Licensing Guidelines

🧪 Testing

- Test suites **must** be written in Jest.
- Test suites **must** be broken up into logical blocks using `describe()`, i.e. all `it()`s **must** be contained within a `describe()` block
- Skipped tests (`xit()`, `it.skip()`, etc.) and test blocks (`xdescribe()`, `describe.skip()`, etc.) **should** not be committed.
- Test block and test names **should** follow a descriptive BDD style where possible.
- Express APIs **should** be tested using supertest to wrap the express app.
- Requests to third-party HTTP APIs **should** be mocked using msw

🤡 Mocking

- When mocking modules, developers **should** use `jest.spyOn()` to create properly typed mocks which can be reused between tests in the same block.

🐥 Don't do this

```
import { operation } from 'module'

import { myFunction } from '.'

jest.mock('module')

describe('myFunction', () => {
  afterEach(() => {
    jest.resetAllMocks()
  })

  it('returns a value if operation succeeds', () => {
    ;(operation as jest.Mock).mockImplementation(() => {
      return 'success'
    })
    expect(myFunction()).toReturn(expect.any(String))
    expect(operation).toHaveBeenCalledWith('something')
  })

  it('throws an error if operation fails', () => {
    ;(operation as jest.Mock).mockImplementation(() => {
      return 'failure'
    })
    expect(() => myFunction()).toThrowError('OPERATION_ERROR')
    expect(operation).toHaveBeenCalledWith('something')
  })
})
```

✅ Do this

```
/* Import the whole module, if spying on one of it's named exports directly */
import * as Module from 'module'

import { myFunction } from '.'

/* Provide a factory to jest mock, and only override the properties you need to
   modify in the test, and spread requireActual() for all other properties */
jest.mock('module', () => ({
  ...jest.requireActual('module'),
  operation: jest.fn(),
}))

describe('myFunction', () => {
  /*
    Instantiate a spy to reuse across all tests in this describe block, including a default implementation.
    mockImplementation (and other mock functions) are now properly typed.
  */
  const operationSpy = jest.spyOn(Module, 'operation').mockImplementation(() => {
    return 'success'
  })

  afterEach(() => {
    /* clear rather than reset/restore, so the default implementation is maintained between tests */
    jest.clearAllMocks()
  })

  it('returns a value if operation succeeds', () => {
    expect(myFunction()).toReturn(expect.any(String))
    /* Assert on the spy (which now has all the proper spy properties), rather than the original function */
    expect(operationSpy).toHaveBeenCalledWith('something')
    /* You can now access the mock itself, e.g. to query it's calls */
    expect(operationSpy.mock.calls[0][0]).toBe('something')
  })

  it('throws an error if operation fails', () => {
    /* Use mock*Once which provide an override which is discarded after this test is run */
    operationSpy.mockImplementationOnce(() => {
      return 'failure'
    })
    expect(() => myFunction()).toThrowError('OPERATION_ERROR')
    expect(operationSpy).toHaveBeenCalledWith('something')
  })
})
```

Logging & Monitoring

References

[1] TypeScript: Documentation - TypeScript 4.0