TypeScript Cookbook — User Stories Study Plan

October 20, 2025

Contents

TEMPLATE — Replace with short, action-oriented title

Epic / Feature	<epic/feature>
Business Value	<why this matters to stakeholders>
Priority / Estimate	**Priority:** Must/Should/Could **SP:** <story points>
Persona	<who benefits>
Dependencies	<tools, repos, environments>
Assumptions / Risks	<key assumptions and risks>

Story *As a <persona>, I want <capability> so that <value>.*

Non-Functional Performance Security Reliability Accessibility

Acceptance Criteria (BDD)

Scenario Happy path

Given the target repo/project/context is available

When the hands-on objectives for this story are executed

Then the stated outcomes/deliverables are produced and reviewed

- ☐ <Task 1 (concrete, 15–60 minutes)>
- ☐ <Task 2>
- ☐ <Task 3>
- ☐ <Add 5–10 test assertions or type tests>
- ☐ <Document findings in README.md>

TC-01 — Project Setup

Epic / Feature	TypeScript Cookbook Study
Business Value	Establish consistent TS project setup across Node, React, and Deno to accelerate later chapters and reduce configuration risk.
Priority / Estimate	**Priority:** Must **SP:** 3
Persona	Full-stack developer starting a typed JS codebase
Dependencies	Node LTS, pnpm/npm, TypeScript (strict), Vitest/Jest, Vite (for React), Deno
Assumptions / Risks	Strict mode enabled; CI can run `tsc -noEmit`. Risk: ESM/CJS mismatch; different TS versions across tools.

Story *As a developer, I want to create a repeatable TS project baseline so that I can focus on learning the cookbook's patterns without configuration drift.*

Non-Functional Performance Security Reliability Accessibility **Acceptance Criteria (BDD)**

Scenario

Happy path

Given the workstation has Node/Deno installed and a clean repo is available

When the setup scripts are executed for Node ESM, React (Vite), and Deno

Then CI can run type-checks and tests successfully; a README documents the setup and trade-offs

- □ Create `/ch01-setup` with `tsconfig.json` (strict) and `tsconfig.base.json`.
- □ Initialize Node ESM project; add Vitest; set `tsc -noEmit` in CI.
- □ Scaffold Vite React app; enable TS strict mode; run a sample test.
- □ Create a minimal Deno example with `deno.json` and `deno task`.
- □ Document module resolution (ESM/CJS), path aliases, and test runner choices.

TC-02 — Basic Types

Epic / Feature	TypeScript Cookbook Study
Business Value	Model data precisely using primitives, tuples, interfaces/types, and safe unknown handling to prevent runtime class of errors.
Priority / Estimate	**Priority:** Must **SP:** 3
Persona	Backend engineer touching external APIs
Dependencies	Strict null checks, ESLint + TS, type tests
Assumptions / Risks	Incoming data may be untrusted; avoid `any`.

Story *As an engineer, I want to replace loose `any` usage with `unknown` + refinements so that misuse is caught at compile time.*

Non-Functional Performance Security Reliability Accessibility **Acceptance Criteria (BDD)**

Scenario
Happy path

Given a small JSON parsing module exists

When the parser is refactored to use precise tuples and type predicates

Then unit tests assert that unsafe access fails to compile while valid paths compile

- □ Convert an `any`-based parser to `unknown` + narrowing functions.
- □ Model a tuple return type for `parseUrl()` and update call sites.
- □ Introduce `symbol`-keyed registry for plugin lookups.
- □ Add 10 `expect-type` assertions proving safe/unsafe paths.

TC-03 — The Type System

Epic / Feature	TypeScript Cookbook Study
Business Value	Use unions, intersections, discriminated unions, and exhaustiveness checks to model real-world state machines safely.
Priority / Estimate	**Priority:** Must **SP:** 3
Persona	Payments developer
Dependencies	Test runner, `assertNever` helper, strictNullChecks
Assumptions / Risks	Future states may be added; enforce exhaustive switches.

Story *As a payments dev, I want to encode workflow states with discriminated unions so that invalid transitions can't ship.*

Non-Functional Performance Security Reliability Accessibility

Acceptance Criteria (BDD)

Scenario

Happy path

Given a `Payment` union type is defined

When all reducers and handlers switch on the discriminator

Then adding a new state fails compilation until all handlers are updated

- ❑ Define `Payment = Created | Authorized | Captured | Refunded`.
- ❑ Implement handlers with `assertNever` to force exhaustiveness.
- ❑ Create a branded `UserId` nominal type and adapters.
- ❑ Add tests that simulate a new state to prove compile-time breakage.

TC-04 — Generics

Epic / Feature	TypeScript Cookbook Study
Business Value	Generalize utilities while keeping inference ergonomic to reduce duplication and improve DX.
Priority / Estimate	**Priority:** Must **SP:** 3
Persona	Library author
Dependencies	Generics playground, `ThisType`, assertion signatures
Assumptions / Risks	Over-generalization hurts inference; keep APIs readable.

Story *As a library author, I want generic helpers like `compose()` and typed builders so that consumers get strong inference with minimal annotations.*

Non-Functional Performance Security Reliability Accessibility

Acceptance Criteria (BDD)

Scenario
Happy path

Given utility function skeletons exist

When generic constraints and assertion signatures are added

Then type tests show correct inference across several call shapes

- □ Implement `compose()` and `mapValues()` with generics.
- □ Write `assertIsNonEmptyArray<T>` using assertion signatures.
- □ Prototype a fluent builder using `ThisType`.
- □ Add `tsd` tests for inference edge cases.

TC-05 — Conditional Types

Epic / Feature	TypeScript Cookbook Study
Business Value	Transform types using `infer`, conditional distribution, and never-filtering to model advanced APIs.
Priority / Estimate	**Priority:** Must **SP:** 3
Persona	API designer
Dependencies	Type utilities workspace
Assumptions / Risks	Conditional types can become unreadable; keep helpers focused.

Story *As an API designer, I want `PickByValue`/`OmitByValue` and shape-sensitive return types so that my APIs adapt to input flags.*

Non-Functional Performance Security Reliability Accessibility **Acceptance Criteria (BDD)**

Scenario

Happy path

Given base utility types are created

When conditional versions are implemented with `infer`

Then tests prove behavior for unions, `any`, and `unknown`

- ❑ Implement `PickByValue<T,V>` and `OmitByValue<T,V>`.
- ❑ Build a flag-sensitive function with conditional return types.
- ❑ Create a `GroupByKind<T>` to partition unions via `infer`.
- ❑ Add edge-case tests for distribution behavior.

TC-06 — String Template Literal Types

Epic / Feature	TypeScript Cookbook Study
Business Value	Leverage template literal types to type event names, routes, and formatters while extracting parameters safely.
Priority / Estimate	**Priority:** Must **SP:** 3
Persona	Frontend engineer
Dependencies	Vite React app; router skeleton
Assumptions / Risks	Recursive templates can hit depth limits; keep patterns shallow.

Story *As a frontend engineer, I want typed event and route keys so that handler payloads and path params are inferred correctly.*

Non-Functional Performance Security Reliability Accessibility **Acceptance Criteria (BDD)**

Scenario Happy path

Given event and router modules exist

When template literal types are added for keys and params

Then handlers receive inferred payloads/params without casting

- ☐ Create `EventMap` with keys like `user:created`.
- ☐ Implement a `format()` enforcing named placeholders.
- ☐ Model `/users/:id` to infer `{ id: string }`.
- ☐ Type tests proving inference across several routes.

TC-07 — Variadic Tuple Types

Epic / Feature	TypeScript Cookbook Study
Business Value	Use variadic tuples to type curry/compose and to transform callback-style APIs into promises.
Priority / Estimate	**Priority:** Must **SP:** 3
Persona	Node developer
Dependencies	Utility lib playground
Assumptions / Risks	Complex parameter lists can confuse inference; keep overloads minimal.

Story *As a Node dev, I want fully typed `curry()` and `Promisify<F>` so that consumers get safe parameter and result inference.*

Non-Functional Performance Security Reliability Accessibility **Acceptance Criteria (BDD)**

Scenario
Happy path

Given function utilities are scaffolded

When variadic tuple types are applied

Then test cases verify parameter splitting and return inference

- ❑ Implement `curry()` supporting multiple arg lengths.
- ❑ Write `Promisify<F>` for Node-style callbacks.
- ❑ Derive a union/enum from a tuple of literals.
- ❑ Add compile-time assertion tests.

TC-08 — Helper Types

Epic / Feature	TypeScript Cookbook Study
Business Value	Create reusable helper types to express “at least one”, “exactly one”, nested partial/required, and union-to-intersection.
Priority / Estimate	**Priority:** Must **SP:** 3
Persona	Library consumer/author
Dependencies	Type playground; `type-fest` for comparison
Assumptions / Risks	Helpers must be documented; prefer well-known names to reduce cognitive load.

Story *As a library author, I want a small helper set so that teams share a common vocabulary for optionality and exclusivity.*

Non-Functional Performance Security Reliability Accessibility **Acceptance Criteria (BDD)**

Scenario
Happy path

Given helper skeletons exist

When helpers are implemented and documented

Then tests confirm semantics vs `type-fest` equivalents

- ❑ Implement `AtLeastOne<T>` and `ExactlyOne<T>`.
- ❑ Implement `DeepPartial<T>` and `DeepRequired<T>`.
- ❑ Add `UnionToIntersection<U>` with tests.
- ❑ Compare ergonomics with `type-fest` in README.

TC-09 — Standard Library and External Type Definitions

Epic / Feature	TypeScript Cookbook Study
Business Value	Augment stdlib gaps (e.g., `Object.keys`) and third-party packages to preserve literal types and improve safety.
Priority / Estimate	**Priority:** Must **SP:** 3
Persona	App developer
Dependencies	Ambient `*.d.ts` typings folder, module augmentation example
Assumptions / Risks	Augmentations should be minimal and discoverable.

Story *As an app dev, I want safe wrappers and augmentations so that key/values and imports retain useful types.*

Non-Functional Performance Security Reliability Accessibility **Acceptance Criteria (BDD)**

Scenario

Happy path

Given a `typings` folder exists

When safe `objectKeys<T>()` wrapper and module augmentation are added

Then React can import `*.svg` with types; unit tests compile

- ❑ Write `objectKeys<T>()` preserving key literals.
- ❑ Augment a third-party module to fix a missing type.
- ❑ Add `declare module '*.svg'` and React usage example.
- ❑ Add tests demonstrating safer usage vs raw stdlib calls.

TC-10 — TypeScript and React

Epic / Feature	TypeScript Cookbook Study
Business Value	Type hooks, polymorphic components, contexts, and `forwardRef` to improve safety and reuse.
Priority / Estimate	**Priority:** Must **SP:** 3
Persona	Frontend engineer working in React
Dependencies	Vite React app, Storybook (optional)
Assumptions / Risks	Prop generics can overfit; prefer straightforward props where possible.

Story *As a React engineer, I want typed hooks and components so that consumers get safe props and proper ref types with minimal annotation.*

Non-Functional Performance Security Reliability Accessibility **Acceptance Criteria (BDD)**

Scenario

Happy path

Given component library skeleton exists

When generic `forwardRef`, `useAsync<T>()`, and a polymorphic `Text` are implemented

Then storybook or tests verify prop inference and refs

- □ Implement `useAsync<T>()` with discriminated states.
- □ Create `List<T>` using `forwardRef` and render prop.
- □ Add polymorphic `Text` that forwards props safely.
- □ Add prop-level tests and a minimal Storybook.

TC-11 — Classes

Epic / Feature	TypeScript Cookbook Study
Business Value	Use visibility modifiers, `override`, strict init, and decorators to build robust class-based utilities.
Priority / Estimate	**Priority:** Should **SP:** 2
Persona	Services engineer
Dependencies	Decorator experiment (optional), strictPropertyInitialization
Assumptions / Risks	Prefer composition where appropriate; document design choices.

Story *As a services engineer, I want a typed* `EventEmitter` *and safe overrides so that misuse is prevented at compile time.*

Non-Functional Performance Security Reliability Accessibility

Acceptance Criteria (BDD)

Scenario
Happy path

Given class skeletons exist

When visibility and override rules are applied; decorator added for timing or memoization

Then tests validate behavior and typing

- □ Implement `EventEmitter<TMap>` with generic payloads.
- □ Enforce `override` on subclass methods.
- □ Add a method decorator for timing or memoization (optional).
- □ Document trade-offs vs functional patterns.

TC-12 — Type Development Strategies

Epic / Feature	TypeScript Cookbook Study
Business Value	Develop a sustainable approach to authoring types, using `satisfies`, type tests, and runtime validation (e.g., Zod).
Priority / Estimate	**Priority:** Must **SP:** 3
Persona	Library maintainer
Dependencies	tsd/expect-type, Zod
Assumptions / Risks	Balance type power and maintainability; avoid clever one-offs.

Story *As a maintainer, I want a small* `@you/ts-utils` *package with tests so that types evolve safely with semantic versioning.*

Non-Functional Performance Security Reliability Accessibility **Acceptance Criteria (BDD)**

Scenario

Happy path

Given a utilities package skeleton exists

When helpers from earlier chapters are consolidated with type tests and Zod schemas

Then the package is published locally and documented with a CHANGELOG

- □ Create `@you/ts-utils` with 6–8 helpers.
- □ Add `tsd` tests and a few Zod schemas.
- □ Publish to local registry (e.g., Verdaccio) and write CHANGELOG.
- □ Draft “when to stop typing” checklist in README.