Frontend code convention

태그

▼ TOSS Frontend Rule

```
# Frontend Design Guideline

This document summarizes key frontend design principles and rules, s
howcasing
recommended patterns. Follow these guidelines when writing frontend
code.

# Readability

Improving the clarity and ease of understanding code.

## Naming Magic Numbers

**Rule:** Replace magic numbers with named constants for clarity.

**Reasoning:**

- Improves clarity by giving semantic meaning to unexplained values.
- Enhances maintainability.

#### Recommended Pattern:

```typescript
const ANIMATION_DELAY_MS = 300;

async function onLikeClick() {
  await postLike(url);
  await delay(ANIMATION_DELAY_MS); // Clearly indicates waiting for a
```
```

```
nimation
  await refetchPostLike();
}
```

Abstracting Implementation Details

Rule: Abstract complex logic/interactions into dedicated components/HOCs.

Reasoning:

- Reduces cognitive load by separating concerns.
- Improves readability, testability, and maintainability of components.

Recommended Pattern 1: Auth Guard

(Login check abstracted to a wrapper/guard component)

```tsx
// App structure
function App() {
  return (
    <AuthGuard>
      {" "}
      {/* Wrapper handles auth check */}
      <LoginStartPage />
    </AuthGuard>
  );
}

// AuthGuard component encapsulates the check/redirect logic
function AuthGuard({ children }) {
  const status = useCheckLoginStatus();
  useEffect(() => {
    if (status === "LOGGED_IN") {
      location.href = "/home";
    }
```

```
  }, [status]);

  // Render children only if not logged in, otherwise render null (or loadi
ng)
  return status !== "LOGGED_IN" ? children : null;
}

// LoginStartPage is now simpler, focused only on login UI/logic
function LoginStartPage() {
  // ... login related logic ONLY ...
  return <>{/* ... login related components ... */}</>;
}
```

Recommended Pattern 2: Dedicated Interaction Component

(Dialog logic abstracted into a dedicated `InviteButton` component)

```tsx
export function FriendInvitation() {
  const { data } = useQuery(/* ... */);

  return (
    <>
      {/* Use the dedicated button component */}
      <InviteButton name={data.name} />
      {/* ... other UI ... */}
    </>
  );
}

// InviteButton handles the confirmation flow internally
function InviteButton({ name }) {
  const handleClick = async () => {
    const canInvite = await overlay.openAsync(({ isOpen, close }) => (
      <ConfirmDialog
        title={`Share with ${name}`}
        // ... dialog setup ...
```

```
    />
  ));

  if (canInvite) {
    await sendPush();
  }
 };

 return <Button onClick={handleClick}>Invite</Button>;
}
```

Separating Code Paths for Conditional Rendering

Rule: Separate significantly different conditional UI/logic into distinct
components.

Reasoning:

- Improves readability by avoiding complex conditionals within one component.
- Ensures each specialized component has a clear, single responsibility.

Recommended Pattern:

(Separate components for each role)

```tsx
function SubmitButton() {
 const isViewer = useRole() === "viewer";

 // Delegate rendering to specialized components
 return isViewer ? <ViewerSubmitButton /> : <AdminSubmitButton />;
}

// Component specifically for the 'viewer' role
```

```
function ViewerSubmitButton() {
  return <TextButton disabled>Submit</TextButton>;
}

// Component specifically for the 'admin' (or non-viewer) role
function AdminSubmitButton() {
  useEffect(() => {
    showAnimation(); // Animation logic isolated here
  }, []);

  return <Button type="submit">Submit</Button>;
}
```

Simplifying Complex Ternary Operators

Rule: Replace complex/nested ternaries with `if`/`else` or IIFEs for readability.

Reasoning:

- Makes conditional logic easier to follow quickly.
- Improves overall code maintainability.

Recommended Pattern:

(Using an IIFE with `if` statements)

```typescript
const status = (() => {
  if (ACondition && BCondition) return "BOTH";
  if (ACondition) return "A";
  if (BCondition) return "B";
  return "NONE";
})();
```

Reducing Eye Movement (Colocating Simple Logic)

Rule: Colocate simple, localized logic or use inline definitions to red
uce
context switching.

Reasoning:

- Allows top-to-bottom reading and faster comprehension.
- Reduces cognitive load from context switching (eye movement).

Recommended Pattern A: Inline `switch`

```tsx
function Page() {
  const user = useUser();

  // Logic is directly visible here
  switch (user.role) {
    case "admin":
      return (
        <div>
          <Button disabled={false}>Invite</Button>
          <Button disabled={false}>View</Button>
        </div>
      );
    case "viewer":
      return (
        <div>
          <Button disabled={true}>Invite</Button> {/* Example for viewer
*/}
          <Button disabled={false}>View</Button>
        </div>
      );
    default:
      return null;
  }
}
```

Recommended Pattern B: Colocated simple policy object

```tsx
function Page() {
  const user = useUser();
  // Simple policy defined right here, easy to see
  const policy = {
    admin: { canInvite: true, canView: true },
    viewer: { canInvite: false, canView: true },
  }[user.role];

  // Ensure policy exists before accessing properties if role might not ma
tch
  if (!policy) return null;

  return (
    <div>
      <Button disabled={!policy.canInvite}>Invite</Button>
      <Button disabled={!policy.canView}>View</Button>
    </div>
  );
}
```

Naming Complex Conditions

Rule: Assign complex boolean conditions to named variables.

Reasoning:

- Makes the _meaning_ of the condition explicit.
- Improves readability and self-documentation by reducing cognitive lo
ad.

Recommended Pattern:

(Conditions assigned to named variables)

```typescript
const matchedProducts = products.filter((product) => {
  // Check if product belongs to the target category
  const isSameCategory = product.categories.some(
    (category) => category.id === targetCategory.id
  );

  // Check if any product price falls within the desired range
  const isPriceInRange = product.prices.some(
    (price) => price >= minPrice && price <= maxPrice
  );

  // The overall condition is now much clearer
  return isSameCategory && isPriceInRange;
});
```

Guidance: Name conditions when the logic is complex, reused, or n
eeds unit
testing. Avoid naming very simple, single-use conditions.

Predictability

Ensuring code behaves as expected based on its name, parameters, an
d context.

Standardizing Return Types

Rule: Use consistent return types for similar functions/hooks.

Reasoning:

- Improves code predictability; developers can anticipate return value s
hapes.
- Reduces confusion and potential errors from inconsistent types.

Recommended Pattern 1: API Hooks (React Query)

```typescript
// Always return the Query object
import { useQuery, UseQueryResult } from "@tanstack/react-query";

// Assuming fetchUser returns Promise<UserType>
function useUser(): UseQueryResult<UserType, Error> {
  const query = useQuery({ queryKey: ["user"], queryFn: fetchUser });
  return query;
}

// Assuming fetchServerTime returns Promise<Date>
function useServerTime(): UseQueryResult<Date, Error> {
  const query = useQuery({
    queryKey: ["serverTime"],
    queryFn: fetchServerTime,
  });
  return query;
}
```

Recommended Pattern 2: Validation Functions

(Using a consistent type, ideally a Discriminated Union)

```typescript
type ValidationResult = { ok: true } | { ok: false; reason: string };

function checkIsNameValid(name: string): ValidationResult {
  if (name.length === 0) return { ok: false, reason: "Name cannot be em
pty." };
  if (name.length >= 20)
    return { ok: false, reason: "Name cannot be longer than 20 character
s." };
  return { ok: true };
}

function checkIsAgeValid(age: number): ValidationResult {
```

```
  if (!Number.isInteger(age))
    return { ok: false, reason: "Age must be an integer." };
  if (age < 18) return { ok: false, reason: "Age must be 18 or older." };
  if (age > 99) return { ok: false, reason: "Age must be 99 or younger." };
  return { ok: true };
}

// Usage allows safe access to 'reason' only when ok is false
const nameValidation = checkIsNameValid(name);
if (!nameValidation.ok) {
  console.error(nameValidation.reason);
}
```

Revealing Hidden Logic (Single Responsibility)

Rule: Avoid hidden side effects; functions should only perform acti
ons
implied by their signature (SRP).

Reasoning:

- Leads to predictable behavior without unintended side effects.
- Creates more robust, testable code through separation of concerns (S
RP).

Recommended Pattern:

```typescript
// Function *only* fetches balance
async function fetchBalance(): Promise<number> {
  const balance = await http.get<number>("...");
  return balance;
}

// Caller explicitly performs logging where needed
async function handleUpdateClick() {
  const balance = await fetchBalance(); // Fetch
```

```
  logging.log("balance_fetched"); // Log (explicit action)
  await syncBalance(balance); // Another action
}
```

Using Unique and Descriptive Names (Avoiding Ambiguity)

Rule: Use unique, descriptive names for custom wrappers/functions to avoid
ambiguity.

Reasoning:

- Avoids ambiguity and enhances predictability.
- Allows developers to understand specific actions (e.g., adding auth) directly
  from the name.

Recommended Pattern:

```typescript
// In httpService.ts - Clearer module name
import { http as httpLibrary } from "@some-library/http";

export const httpService = {
  // Unique module name
  async getWithAuth(url: string) {
    // Descriptive function name
    const token = await fetchToken();
    return httpLibrary.get(url, {
      headers: { Authorization: `Bearer ${token}` },
    });
  },
};

// In fetchUser.ts - Usage clearly indicates auth
import { httpService } from "./httpService";
export async function fetchUser() {
```

```
  // Name 'getWithAuth' makes the behavior explicit
  return await httpService.getWithAuth("...");
}
```

Cohesion

Keeping related code together and ensuring modules have a well-defined, single
purpose.

Considering Form Cohesion

Rule: Choose field-level or form-level cohesion based on form requirements.

Reasoning:

- Balances field independence (field-level) vs. form unity (form-level).
- Ensures related form logic is appropriately grouped based on requirements.

Recommended Pattern (Field-Level Example):

```tsx
// Each field uses its own `validate` function
import { useForm } from "react-hook-form";

export function Form() {
  const {
    register,
    formState: { errors },
    handleSubmit,
  } = useForm({
    /* defaultValues etc. */
  });

  const onSubmit = handleSubmit((formData) => {
```

````
    console.log("Form submitted:", formData);
  });

  return (
    <form onSubmit={onSubmit}>
      <div>
        <input
          {...register("name", {
            validate: (value) =>
              value.trim() === "" ? "Please enter your name." : true, // Examp
le validation
          })}
          placeholder="Name"
        />
        {errors.name && <p>{errors.name.message}</p>}
      </div>
      <div>
        <input
          {...register("email", {
            validate: (value) =>
              /^[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}$/i.test(value)
                ? true
                : "Invalid email address.", // Example validation
          })}
          placeholder="Email"
        />
        {errors.email && <p>{errors.email.message}</p>}
      </div>
      <button type="submit">Submit</button>
    </form>
  );
}
```

#### Recommended Pattern (Form-Level Example):

```tsx
// A single schema defines validation for the whole form
````

```
import * as z from "zod";
import { useForm } from "react-hook-form";
import { zodResolver } from "@hookform/resolvers/zod";

const schema = z.object({
  name: z.string().min(1, "Please enter your name."),
  email: z.string().min(1, "Please enter your email.").email("Invalid emai
l."),
});

export function Form() {
  const {
    register,
    formState: { errors },
    handleSubmit,
  } = useForm({
    resolver: zodResolver(schema),
    defaultValues: { name: "", email: "" },
  });

  const onSubmit = handleSubmit((formData) => {
    console.log("Form submitted:", formData);
  });

  return (
    <form onSubmit={onSubmit}>
      <div>
        <input {...register("name")} placeholder="Name" />
        {errors.name && <p>{errors.name.message}</p>}
      </div>
      <div>
        <input {...register("email")} placeholder="Email" />
        {errors.email && <p>{errors.email.message}</p>}
      </div>
      <button type="submit">Submit</button>
    </form>
  );
}
```

```

**Guidance:** Choose **field-level** for independent validation, async checks,
or reusable fields. Choose **form-level** for related fields, wizard forms, or
interdependent validation.

## Organizing Code by Feature/Domain

**Rule:** Organize directories by feature/domain, not just by code type.

**Reasoning:**

- Increases cohesion by keeping related files together.
- Simplifies feature understanding, development, maintenance, and deletion.

#### Recommended Pattern:

(Organized by feature/domain)

```
src/
├── components/ # Shared/common components
├── hooks/   # Shared/common hooks
├── utils/   # Shared/common utils
├── domains/
│   ├── user/
│   │   ├── components/
│   │   │   └── UserProfileCard.tsx
│   │   ├── hooks/
│   │   │   └── useUser.ts
│   │   └── index.ts # Optional barrel file
│   ├── product/
│   │   ├── components/
│   │   │   └── ProductList.tsx
```
```

```
│   │   ├── hooks/
│   │   │   └── useProducts.ts
│   │   └── ...
│   └── order/
│       ├── components/
│       │   └── OrderSummary.tsx
│       ├── hooks/
│       │   └── useOrder.ts
│       └── ...
└── App.tsx
```

Relating Magic Numbers to Logic

Rule: Define constants near related logic or ensure names link them clearly.

Reasoning:

- Improves cohesion by linking constants to the logic they represent.
- Prevents silent failures caused by updating logic without updating related
  constants.

Recommended Pattern:

```typescript
// Constant clearly named and potentially defined near animation logic
const ANIMATION_DELAY_MS = 300;

async function onLikeClick() {
  await postLike(url);
  // Delay uses the constant, maintaining the link to the animation
  await delay(ANIMATION_DELAY_MS);
  await refetchPostLike();
}
```

_Ensure constants are maintained alongside the logic they depend on o
r clearly
named to show the relationship._

Coupling

Minimizing dependencies between different parts of the codebase.

Balancing Abstraction and Coupling (Avoiding Premature Abstractio
n)

Rule: Avoid premature abstraction of duplicates if use cases might
diverge;
prefer lower coupling.

Reasoning:

- Avoids tight coupling from forcing potentially diverging logic into one
  abstraction.
- Allowing some duplication can improve decoupling and maintainability
when
  future needs are uncertain.

Guidance:

Before abstracting, consider if the logic is truly identical and likely to
stay identical across all use cases. If divergence is possible (e.g.,
different pages needing slightly different behavior from a shared hook li
ke
`useOpenMaintenanceBottomSheet`), keeping the logic separate initiall
y (allowing
duplication) can lead to more maintainable, decoupled code. Discuss tr
ade-offs
with the team. _[No specific 'good' code example here, as the recomm
endation is
situational awareness rather than a single pattern]._

Scoping State Management (Avoiding Overly Broad Hooks)

Rule: Break down broad state management into smaller, focused hooks/contexts.

Reasoning:

- Reduces coupling by ensuring components only depend on necessar
y state slices.
- Improves performance by preventing unnecessary re-renders from un
related state
  changes.

Recommended Pattern:

(Focused hooks, low coupling)

```typescript
// Hook specifically for cardId query param
import { useQueryParam, NumberParam } from "use-query-params";
import { useCallback } from "react";

export function useCardIdQueryParam() {
  // Assuming 'query' provides the raw param value
  const [cardIdParam, setCardIdParam] = useQueryParam("cardId", Nu
mberParam);

  const setCardId = useCallback(
    (newCardId: number | undefined) => {
      setCardIdParam(newCardId, "replaceIn"); // Or 'push' depending on
desired history behavior
    },
    [setCardIdParam]
  );

  // Provide a stable return tuple
  return [cardIdParam ?? undefined, setCardId] as const;
}
```

```
// Separate hook for date range, etc.
// export function useDateRangeQueryParam() { /* ... */ }
```

Components now only import and use `useCardIdQueryParam` if they need `cardId`,
decoupling them from date range state, etc.

Eliminating Props Drilling with Composition

Rule: Use Component Composition instead of Props Drilling.

Reasoning:

- Significantly reduces coupling by eliminating unnecessary intermediate
  dependencies.
- Makes refactoring easier and clarifies data flow in flatter component trees.

Recommended Pattern:

```tsx
import React, { useState } from "react";

// Assume Modal, Input, Button, ItemEditList components exist

function ItemEditModal({ open, items, recommendedItems, onConfirm, onClose }) {
  const [keyword, setKeyword] = useState("");

  // Render children directly within Modal, passing props only where needed
  return (
    <Modal open={open} onClose={onClose}>
      {/* Input and Button rendered directly */}
      <div
        style={{
```

```
          display: "flex",
          justifyContent: "space-between",
          marginBottom: "1rem",
        }}
      >
        <Input
          value={keyword}
          onChange={(e) => setKeyword(e.target.value)} // State managed
here
          placeholder="Search items..."
        />
        <Button onClick={onClose}>Close</Button>
      </div>
      {/* ItemEditList rendered directly, gets props it needs */}
      <ItemEditList
        keyword={keyword} // Passed directly
        items={items} // Passed directly
        recommendedItems={recommendedItems} // Passed directly
        onConfirm={onConfirm} // Passed directly
      />
    </Modal>
  );
}

// The intermediate ItemEditBody component is eliminated, reducing co
upling.
```

Formatter ⇒ AirBnB Prettier

1. 설계 원칙 (Design Principles)

추상화와 구현 상세의 분리 (Abstraction)

복잡한 로직이나 상호작용은 전용 컴포넌트나 훅(Wrapper/HOC)으로 감싸서, 상위 컴포넌트는 **'어떻게(How)'가 아니라 '무엇(What)'을 하는지**만 보여주도록 합니다.

- 인증 체크, 다이얼로그 처리 등 복잡한 로직은 래퍼(Wrapper) 컴포넌트로 추상화하여 인지 부하를 줄입니다.

```
// 인증 체크 로직을 AuthGuard라는 래퍼 컴포넌트에 위임
function App() {
  return (
    <AuthGuard>
      <HomePage />
    </AuthGuard>
  );
}
```

조건부 렌더링의 분리 (Separating Code Paths)

역할(Role)이나 상태에 따라 UI나 로직이 크게 달라진다면, 하나의 컴포넌트 안에서 `if/else` 나 삼항 연산자로 처리하기보다 **별도의 컴포넌트로 분리**합니다.

- **규칙:** 서로 다른 책임(예: 관리자 vs 뷰어)을 가진 UI는 별도 컴포넌트로 분기하여 가독성을 높입니다.

```
function SubmitButton() {
  const isViewer = useRole() === "viewer";
  // 역할에 맞는 전용 컴포넌트를 렌더링
  return isViewer ? <ViewerSubmitButton /> : <AdminSubmitButton />;
}

// 각 컴포넌트는 자신의 역할에만 집중
function ViewerSubmitButton() { return <TextButton disabled>Submit</TextButton>; }
function AdminSubmitButton() { return <Button type="submit">Submit</Button>; }
```

상태 관리의 스코핑 (State Scoping)

상태 관리를 비대하게 만들지 않고, **필요한 상태만 관리하는 작은 단위의 훅(Hook)**으로 쪼갭니다.

- **규칙:** 컴포넌트가 불필요한 상태(데이터)에 의존하지 않도록, 상태 관리 훅을 기능별로 잘게 나눕니다.

```
// 특정 쿼리 파라미터(cardId)만 관리하는 전용 훅 사용
export function useCardIdQueryParam() {
  const [cardIdParam, setCardIdParam] = useQueryParam("cardId", Numbe
rParam);
  // ...
  return [cardIdParam, setCardId];
}
// 컴포넌트는 정확히 필요한 훅만 가져다 씀
```

Props Drilling 방지 (Composition)

Props를 여러 단계 거쳐 전달해야 한다면, 중간 컴포넌트를 거치지 않고 **컴포넌트 합성 (Composition)**을 활용합니다.

- **규칙:** 불필요한 중간 전달자를 없애기 위해 `children`이나 직접 렌더링 방식을 사용하여 결합도를 낮춥니다

```
// ✅ Modal 내부에 필요한 Input을 직접 렌더링하여 Props 전달 단계 삭제
<Modal open={open} onClose={onClose}>
  <div style={{...}}>
    <Input
      value={keyword}
      onChange={(e) => setKeyword(e.target.value)} // 상태를 직접 주입
    />
    <Button onClick={onClose}>Close</Button>
  </div>
</Modal>
```

폼 응집도 고려 (Form Cohesion)

폼의 요구사항에 따라 필드 단위(Field-Level)와 폼 단위(Form-Level) 중 적절한 응집도 전략을 선택합니다.

- **Field-Level (필드 단위):** 유효성 검사 로직이 서로 독립적이거나 재사용성이 중요한 경우, 각 입력 필드 컴포넌트 내부에 검증 로직을 위치시킵니다.
- **Form-Level (폼 단위):** 필드 간 의존성이 있거나(예: 비밀번호 확인), 전체 폼의 흐름이 중요한 경우 `Zod`와 같은 스키마 라이브러리를 사용하여 한곳에서 중앙 집중적으로 관

리합니다.

```
// Form-Level 예시 (Zod 스키마 사용)
const schema = z.object({
  name: z.string().min(1, "이름을 입력해주세요."),
  email: z.string().email("유효한 이메일이 아닙니다."),
});
// 스키마 하나로 폼 전체의 응집도를 높임
```

---

2. 네이밍 규칙 (Naming Conventions)

camelCase

첫 단어는 소문자로 시작, 이후 단어의 첫 글자는 대문자로 씁니다.

- **변수명:** `const isHistoryPage = ...;`
- **함수명:** `const handleClickResultButton = () => {};`
- **React 커스텀 훅:** `const useCabinetData = () => {};`
- **개체 속성명 (interface, type):**

```
interface UserData {
  name: string | null;
  isVisible: boolean;
}
```

- **폴더/파일 명 (api, hooks, icons 등 유틸리티):**
  - `fetchInterceptApi.tsx` (api 폴더)
  - `angleDown.svg` (icons 폴더)
  - `useLogin.tsx` (hooks 폴더)

PascalCase

모든 단어의 첫 글자를 대문자로 씁니다.

- **객체 명 (Interface, Type 정의):** `interface UserData { ... }`
- **리액트 컴포넌트 명:** `const BuildingSelectButton = () => {};`

- **폴더/파일 명 (페이지, 컴포넌트):**
  - `BuildingSelectButton.tsx` (컴포넌트)
  - `ProfilePage.tsx` (페이지)

SNAKE_CASE

대문자로 작성하며 단어 간에는 `_` 로 구분합니다.

- **상수 및 환경 변수:**

```
const BASE_URL = 'https://api.example.com';
const API_KEY = process.env.API_KEY;
```

---

3. 함수 및 변수 (Function & Variables)

함수 작명 규칙

함수의 이름은 의미 있는 동사로 시작하여, 이름만 보고도 역할을 알 수 있어야 합니다.

- `handle` : 이벤트를 핸들링하는 내부 함수 (`handleLoginButton`)
- `on` : Props로 전달되는 이벤트 리스너 함수 (`onSubmit`)
- `is` : 반환값이 Boolean인 함수 (`isNot` 등 부정형 지양)
- `get` : 값을 계산하거나 가져와서 반환하는 함수 (`getStatusColor`)
- `fetch` : API 요청 등 비동기 함수 (`fetchCabinetDetailInformation`)

선언 방식

- **화살표 함수**를 기본으로 사용합니다.

```
const sum = (a, b) => a + b;
```

변수

- `var` 는 사용하지 않으며, `const` 와 `let` 을 사용합니다.
- 배열 변수는 끝에 `List` 를 붙입니다. (예: `userList`)
- 지나친 줄임말은 지양합니다. (예: `statNm` (X) → `stationName` (O))

- **상태 변수:** 길어져도 의미를 명확히 합니다.

```
const [selectedBuilding, setSelectedBuilding] = useState();
```

---

4. React & TypeScript

기본 규칙

- **Any 지양:** 난해한 경우가 아니라면 `any` 타입을 사용하지 않습니다.
- **Props 정의:** `interface`를 사용하여 명확히 정의합니다.
- **함수 타입:** 매개변수와 반환값 타입을 명시합니다.
- **컴포넌트 선언:** `React.FC` 대신 직접 정의하는 방식을 지향합니다.

```
interface LoginApiProps {
  studentNumber: string;
  password: string;
}
const LoginApi = ({ studentNumber, password }: LoginApiProps) => { ... };
```

리턴 타입의 표준화 (Standardizing Return Types)

유사한 기능을 하는 함수나 훅은 일관된 반환 타입 구조를 유지하여 예측 가능성을 높입니다.

- **규칙:** 성공/실패 여부나 데이터를 포함하는 표준화된 객체(Discriminated Union 등)를 반환하도록 설계합니다.
- **이유:** 개발자가 반환값의 형태를 미리 예측할 수 있어 혼란과 에러를 줄일 수 있습니다

```
type ValidationResult = { ok: true } | { ok: false; reason: string };

function checkIsAgeValid(age: number): ValidationResult {
  if (age < 18) return { ok: false, reason: "18세 이상이어야 합니다." };
  return { ok: true };
}

// 사용하는 쪽에서 ok 여부에 따라 안전하게 처리 가능
const result = checkIsAgeValid(age);
```

```
if (!result.ok) {
  console.error(result.reason);
}
```

Client vs Server (Next.js)

- **Client Component:** `useState`, `useEffect`, 브라우저 이벤트(`onClick`) 등을 사용하는 컴포넌트는 파일 최상단에 `'use client';`를 명시합니다.
- Server Component:
  - SSR(ServerSide Rendering)의 경우 파일 최상단에 `'import 'server-only'` 를 통해서 Client Component의 사용을 제한합니다.
- 파일 명명은 다음과 같이 접두사로 진행합니다
  - ServerComponent
    - `SCUserBox.tsx`
  - ClientComponent
    - `CCUserTask.tsx`

---

5. 프로젝트 구조 (Next.js App Router)

명명 규칙

- **kebab-case:** `app` 디렉토리 내부의 라우팅 폴더 (URL 경로와 일치해야 함)
- **camelCase:** `api`, `hooks`, `icons`, `lib` 등 기능/유틸리티 폴더
- **PascalCase:** `Components`, `Domains` 등 UI 및 비즈니스 로직 포함 폴더

구조 원칙

- **응집도(Cohesion):** 특정 기능(`Game`, `User`)과 관련된 모든 코드(UI, Hook, API)는 한 폴더에 모아둡니다.
- **재사용성(Reusability):** 2곳 이상의 도메인에서 공통으로 사용되는 것만 `Global` 영역(`src/Components`, `src/hooks`)으로 승격시킵니다.
- **Next.js 15:** 서버 액션(Server Actions)과 API 로직도 해당 도메인 폴더 내에 위치시킵니다.
  - **app 폴더:** 페이지와 레이아웃을 정의합니다. (소문자 폴더명 = URL)

- **Domains 폴더:** 기능/도메인별로 코드(컴포넌트, 훅 등)를 응집시킵니다.
- **Components 폴더:** 특정 도메인에 종속되지 않는 **공용 디자인 시스템** 컴포넌트만 위치합니다.

Tree 구조 예시

```
ROOT
├── public                # [Static Assets] 외부에서 URL로 직접 접근 가능한 파일들
│   ├── images            # 예: og-image.png, robots.txt, favicon.ico
│   └── fonts             # (Local Font를 쓸 경우)
│
├── src
│   ├── app
│   │   ├── layout.tsx
│   │   ├── page.tsx
│   │   └── globals.css     # [Global CSS] Tailwind Directive (@tailwind base...) 포함
│   │
│   ├── assets            # [Shared Assets] 컴포넌트에서 import해서 쓰는 공용 이미지/아이콘
│   │   ├── images          # 예: logo.png, placeholder.jpg
│   │   └── icons         # (SVG 파일들)
│   │
│   ├── Components          # [Shared UI]
│   │   └── Button        # 예: className을 통해 Tailwind 적용
│   │
│   ├── Domains           # [Features]
│   │   ├── Game
│   │   │   ├── assets      #  Game 도메인 전용 이미지/아이콘
│   │   │   │   └── card-sprite.png
│   │   │   ├── components
│   │   │   └── ...
│   │   └── User
│   │
│   └── lib               # [Utils]
│       └── utils.ts        # Tailwind 병합 유틸리티 (cn 함수)
│
```

```
├── tailwind.config.ts      # [Config] Tailwind 설정 (Design Token 정의)
├── postcss.config.js       # [Config] PostCSS 설정
└── ...
```

Import 순서

`React 내장` → `라이브러리(Next.js 포함)` → `내부 모듈` → `컴포넌트` → `스타일/이미지` 순으로 작성합니다.

```
// 1. React 내장
import { useState, useEffect } from "react";

// 2. 라이브러리
import { useRouter } from "next/navigation";
import { useForm } from "react-hook-form";

// 3. 내부 모듈 (@ 경로 맵핑 활용)
import { userDataApi } from "@/api/userDataApi";
import { useCabinetData } from "@/hooks/useCabinetData";

// 4. 컴포넌트
import CabinetFooterMenuButton from "@/components/CabinetFooterMen
uButton";

// 5. 스타일 및 이미지
import SearchSVG from "@/icons/search.svg?react";
```

---

6. 스타일 및 기타 규칙

CSS

- **단위:** 반응형을 고려하여 `px` 대신 `rem` 또는 `em`을 권장합니다.
- **Tailwind:** 반응형 접두사(`ss` , `sm`)와 유틸리티 클래스를 적극 활용합니다.
- **레이아웃:** `flex` , `grid` , `gap` 을 사용하여 유연하게 구성합니다.

문자열

- 기본적으로 **작은따옴표(')**를 사용합니다.

```
const message = 'Hello, World!';
```

- **큰따옴표(")** 사용 예외:
  - HTML 속성: `<input type="text" />`
  - JSON 문자열: `'{"name": "user"}'`
  - 작은따옴표가 포함된 문자열: `"It's a good day"`

하드 코딩 방지

- URL, 매직 넘버 등은 상수(`const`)나 환경 변수(`process.env`)로 관리합니다.
- **이유:** 유지보수성 향상 및 값의 목적 파악 용이.

에러 처리

- 에러 처리는 비동기로 수행하며, `console.error`로 로깅합니다.
- 상태 코드와 에러 메시지를 통해 관리자가 쉽게 파악할 수 있도록 합니다.

분기 처리

- **switch-case:** 3개 이상의 값 평가 시 권장.
- **if / 삼항연산자:** 복잡한 조건은 `if`, 간단한 값 할당은 삼항연산자 사용.
- **조건부 렌더링:** `&&` 연산자보다는 삼항연산자를 권장합니다 (0이 렌더링되는 실수 방지).
  - **단, 중첩된 삼항 연산자는 가독성을 해치므로 지양하고 if문이나 별도 컴포넌트로 분리합니다.**

```
<div>{isLoggedIn ? <Dashboard /> : <Login />}</div>
```

기타

- URL 경로는 소문자(`lowercase`)를 사용합니다.
- TypeScript `enum` 대신 Union Type이나 객체(const assertion)를 사용합니다.
- 불필요한 `<div>` 래퍼 대신 Fragment `<>`를 사용합니다.

⇒ 공통 컴포넌트에 대한 네이밍