Component-Based Architecture (CBA)

▼ Description

การแยกโครงสร้างแบบนี้ โดยที่คุณมี

1. **UI Primitives** (Atoms)
2. **Feature Modules** (Molecules/Organisms ตาม Atomic Design)
3. **Layout Components** (Templates)
4. **Page Containers** (Pages)

โดยรวมแล้วเค้าเรียกว่า **Component-Based Architecture** หรือถ้าเจาะลึกลงไปก็คือ **Atomic Design** ผสมกับ **Feature-First (Domain-Driven) Folder Structure** ครับ

- **Component-Based Architecture**

  คือแนวคิดที่แอปถูกแบ่งออกเป็น “ชิ้นส่วน” (components) ย่อย ๆ แต่ละชิ้นดูแลหน้าที่เดียว (Single Responsibility) ทำให้โค้ดอ่านง่าย ขยายต่อได้

- **Atomic Design**

  แบ่งระดับ component เป็น Atoms → Molecules → Organisms → Templates → Pages

  (ในโปรเจกต์เล็กคุณอาจใช้แค่ Atoms + Molecules + Pages)

- **Feature-First (Domain-Driven) Structure**

  คือการจัดโฟลเดอร์ตามฟีเจอร์ (เช่น `/todo`, `/user`) ทำให้แต่ละฟีเจอร์มีทั้ง Container, Components, Hook ของมันเอง

สรุปคือคุณกำลังใช้ **Component-Based Architecture** ในรูปแบบ **Atomic + Feature-Driven** นั่นเองครับ 😊

▼ CBA คืออะไร

สถาปัตยกรรมแบบ **Component-Based Architecture** (CBA) นั้นคือแนวคิดในการออกแบบซอฟต์แวร์ระดับ **Architecture** ที่เน้นการแบ่งระบบออกเป็นหน่วยย่อยๆ เรียกว่า **Component** แต่ละตัวมีหน้าที่ชัดเจน สื่อสารกันด้วย Interface ที่กำหนดไว้ล่วงหน้า ซึ่งช่วยให้

- **Reuse & Maintain**

  นำ component ไปใช้ซ้ำในหลายจุด ปรับปรุง component ใด component หนึ่งได้โดยไม่กระทบส่วนอื่น

- **Loose Coupling / High Cohesion**

  แต่ละ component ทำงานร่วมกันผ่าน contract (props, events, API) แต่ภายใน component จะดูแล logic ของตัวเองจนจบ

- **Testability**

  ทดสอบแต่ละ component แยกกันได้ง่าย (unit test)

CBA คือการออกแบบอะไร

1. **การออกแบบโครงสร้างซอฟต์แวร์ (Software Architecture)**

   – ระบุว่าแต่ละ **Feature / Domain** จะแบ่งเป็น component ใดบ้าง

   – กำหนด interface (public API) ระหว่าง component

2. **การออกแบบโครงสร้างโค้ด (Code Organization)**

   – โฟลเดอร์ ไฟล์ จัดเรียงตาม component boundaries

   – แต่ละ component มีทั้ง logic, style, tests ของตัวเอง

3. **การออกแบบการพัฒนา (Development Workflow)**

   – ทีมหรือคุณในฐานะ solo dev สามารถทำงานบน component ต่างคนต่างส่วนได้

   – ใช้ CI/CD, versioning แยก component ได้ (ถ้าใช้ mono-repo หรือ multi-repo)

CBA ≠

- **ไม่ใช่แค่โครงสร้างไฟล์**

  (ถึงจะสะท้อนใน folder/file structure แต่หัวใจคือ “แยกความรับผิดชอบ” ของแต่ละ component มากกว่า)

- **ไม่ใช่แค่ UI design**

  (แม้จะนิยมใช้ใน React/Next.js แต่ CBA ครอบคลุม backend, API, service layer ก็ได้)

- **ไม่ใช่แค่ pattern ในไฟล์เดียว**

(แต่เป็นแนวทางจัด module ข้ามทั้งระบบ มากกว่าแค่ function หรือ class)

สรุป

> Component-Based Architecture คือแนวคิดออกแบบระดับสถาปัตยกรรมซอฟต์แวร์ ที่แบ่งระบบเป็นหน่วยย่อย (components) แต่ละอันมี interface ชัดเจน ทำให้โค้ด reuse ได้ ดูแลง่าย แยกทีมทำงานได้ เหมาะกับการพัฒนาต่อยอดระยะยาว

จากนั้นถึงจะ “สะท้อน” มาในรูปโครงสร้างโฟลเดอร์ ไฟล์ และวิธีการเรียกใช้ component เหล่านั้นในโปรเจ็คของคุณครับ!

Architecture Detail

"Use the CBA pattern with clearly separated UI primitives, feature modules, containers, and pages. Store reusable logic in `hooks`, context in `contexts`, business logic in `lib`, and API clients in `services`. Page routes are handled under `/app`. All components must follow single responsibility. Type definitions go in `/types`. Prefer named exports except for `page.tsx`."

solo dev ที่อยากได้โครงสร้างที่...

- **ง่ายต่อการจัดการ**
- **นำไปใช้ซ้ำ** (reuse)
- **ต่อยอดได้เรื่อยๆ**
- **ประสิทธิภาพดี** ทั้งฝั่ง dev และ UX

แนวทางจัด module ทั้งระบบ

ใช้แนวทาง **Hybrid “Domain-Driven + UI Primitives”** แบบนี้ครับ:

การแยกโครงสร้างแบบนี้ โดยที่คุณมี

1. **UI Primitives** (Atoms)

2. **Feature Modules** (Molecules/Organisms ตาม Atomic Design)
3. **Layout Components** (Templates)
4. **Page Containers** (Pages)

โดยรวมแล้วเค้าเรียกว่า **Component-Based Architecture** หรือถ้าเจาะลึกลงไปก็คือ **Atomic Design** ผสมกับ **Feature-First (Domain-Driven) Folder Structure** ครับ

- **Component-Based Architecture**

  คือแนวคิดที่แอปถูกแบ่งออกเป็น “ชิ้นส่วน” (components) ย่อย ๆ แต่ละชิ้นดูแลหน้าที่เดียว (Single Responsibility) ทำให้โค้ดอ่านง่าย ขยายต่อได้

- **Atomic Design**

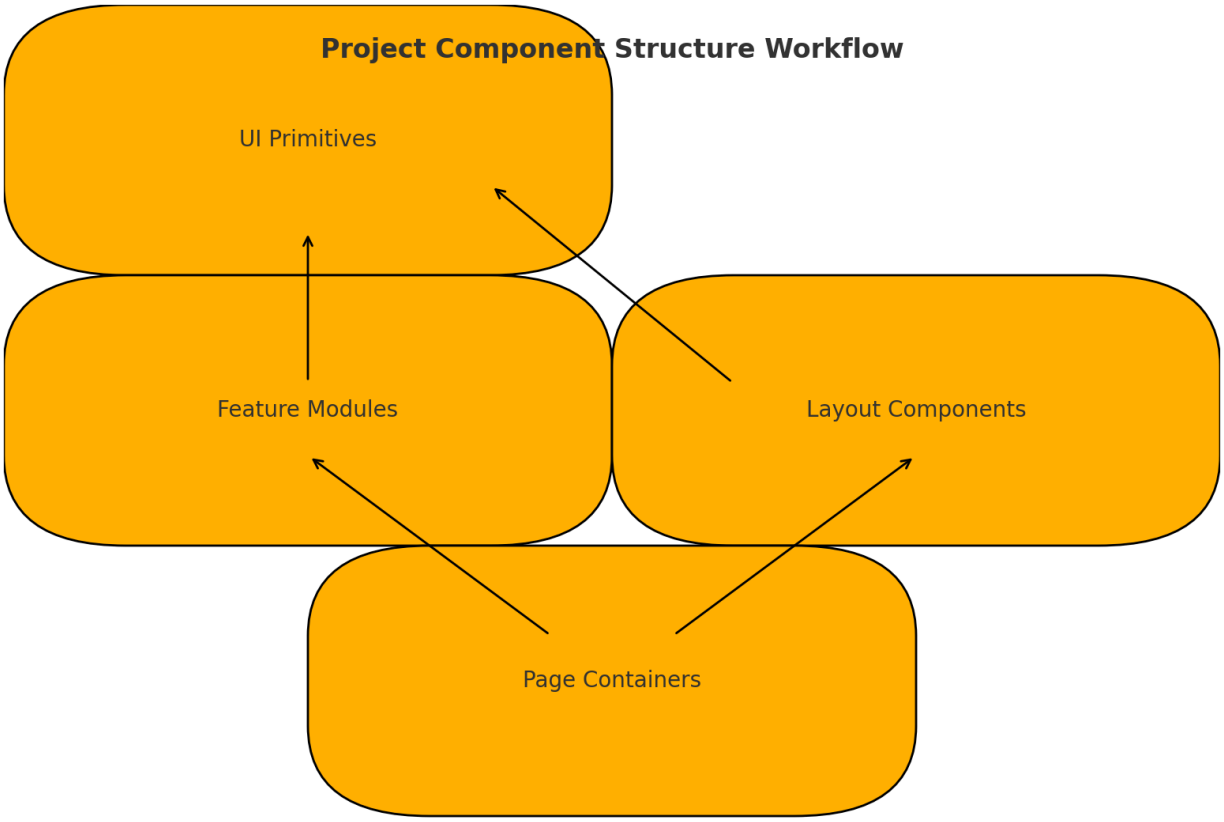
  แบ่งระดับ component เป็น Atoms → Molecules → Organisms → Templates → Pages

  (ในโปรเจกต์เล็กคุณอาจใช้แค่ Atoms + Molecules + Pages)

- **Feature-First (Domain-Driven) Structure**

  คือการจัดโฟลเดอร์ตามฟีเจอร์ (เช่น `/todo`, `/user`) ทำให้แต่ละฟีเจอร์มีทั้ง Container, Components, Hook ของมันเอง

สรุป Structure



- **UI Primitives** (Button, Card, Icon) อยู่บนสุด
- รากฐานให้ **Feature Modules** (เช่น Counter, User) และ **Layout Components** (Header, Footer) ดึงไปใช้
- สุดท้าย **Page Containers** จึงประกอบทั้งสองกลุ่มมาแสดงในหน้า

Project Structure.

```
/components
  /ui             # “Primitives” / Atoms ที่ใช้ซ้ำได้ทั่วแอป
    Button.tsx
    Input.tsx
    Icon.tsx
    Card.tsx
    index.ts      # re-export ทุกตัวจากโฟลเดอร์นี้
```

```
  /features        # จะแยกตามฟีเจอร์หรือ domain ของแอป
    /counter
      CounterContainer.tsx
      CounterButtons.tsx
      PointDisplay.tsx
      index.ts      # re-export ทั้งหมดในโฟลเดอร์ counter
    /user
      LoginForm.tsx
      UserBar.tsx
      index.ts      # re-export ทั้งหมดในโฟลเดอร์ user
  /layout          # ธรณีประตูของแต่ละหน้า (Header, Footer, Layout wrapper)
    Header.tsx
    Footer.tsx
    MainLayout.tsx
/pages
  index.tsx
  about.tsx
  ...              # Next.js routes
/styles
  theme.ts         # สี, spacing, typography constants
  globals.css
/utils
  cookie.ts
  ...              # helper functions / hooks
```

ทำไมโครงสร้างนี้เหมาะกับ solo dev?

1. **UI Primitives (“Atoms”)**
   - เก็บโค้ดขององค์ประกอบพื้นฐาน (Button, Card, Icon) ไว้ที่เดียว
   - ถ้าอยากเปลี่ยน style หรือ behavior ของปุ่มทั้งหมด ก็แก้แค่ใน `components/ui/Button.tsx`
   - Reuse ง่าย ไม่ต้องสร้างปุ่มซ้ำแต่ละฟีเจอร์
2. **Feature Folders (“Domain-Driven”)**
   - แบ่งงานตามฟีเจอร์จริง ๆ ของแอป (เช่น counter, user)

- ทุก component ที่เกี่ยวกับ counter อยู่ในโฟลเดอร์เดียวกัน (Container, Buttons, Display)
- ถ้าฟีเจอร์ไหนโตขึ้น ก็เพิ่มไฟล์ในโฟลเดอร์นั้น ไม่กระทบส่วนอื่น

3. **Index Files**
   - ในแต่ละโฟลเดอร์มี `index.ts` re-export ให้ import สั้น ๆ
   - ลดความซ้ำซ้อนของ path
4. **Layout Folder**
   - จัดโครงสร้าง global layout แยกออกจาก feature logic
   - หน้า page แค่ import `MainLayout` แล้วใส่ children เข้าไป
5. **Performance & Scaling**
   - Next.js จะทำ tree-shaking และ code-splitting ให้อัตโนมัติ
   - เมื่อคุณ import จาก `components/ui` หรือ `components/features/...` ระบบ build จะรวมแค่ไฟล์ที่ใช้งานจริง
   - สำหรับ UX ก็จะได้ bundle ขนาดเหมาะสม โหลดเร็ว
6. **Maintainability**
   - อ่านโค้ดง่าย — รู้เลยว่าถ้าอยากแก้ปุ่ม ไปที่ `ui/Button.tsx`
   - ต่อเติมฟีเจอร์ใหม่ไม่ต้องเลื่อนโค้ดไปมา
   - ทดสอบแยก unit test ได้สะดวก

สรุป:

แนวทาง hybrid นี้ “เรียบง่าย” แต่ยัง “มีระเบียบ” และ “พร้อมต่อยอด” เหมาะทั้งโปรเจ็คเล็กที่เริ่มต้นและยังรองรับการเติบโตของแอปในระยะยาว