Rockstar Developer 2025

အိမောင်
Fairway

FULL STACK JAVASCRIPT

ပုံနှိပ်မှတ်တမ်း

ထုတ်ဝေသူ	-	ဦးအိမောင် Fairway စာပေ (၀၃၃၀၇) အမှတ် 3A308၊ ရတနာနှင်းဆီအိမ်ရာ၊ ရတနာလမ်း၊ ဒဂုံဆိပ်ကမ်းမြို့နယ်၊ ရန်ကုန်တိုင်းဒေသကြီး။
ပုံနှိပ်သူ	-	ဦးအိမောင် Ebook (ဒစ်ဂျစ်တယ်စာအုပ်)
မျက်နှာဖုံးဒီဇိုင်း	-	အိမောင် (Fairway)
ကွန်ပျူတာစာစီ	-	အိမောင် (Fairway)
တန်ဖိုး	-	အခမဲ့
လိုင်စင်	-	CC-BY-NC-SA https://creativecommons.org/licenses/by-nc-sa/4.0
ဖြန့်ချီရေး	-	Fairway စာပေ (၀၉ - ၂၅၂ ၄၂၆ ၃၈၈)

အိမောင် (Fairway)

Rockstar Developer 2025

Fairway စာပေ၊ ပထမအကြိမ်၊ ၂၀၂၄။

စာမျက်နှာ - ၄၃၅ (၃၈ စင်တီ x ၅၃ စင်တီ)

မာတိကာ

4 မိတ်ဆက်

5 အခန်း (၁) – JavaScript – Node, NPM

15 အခန်း (၂) – React – New React Project with Vite

25 အခန်း (၃) – React – Components, Props, States

55 အခန်း (၄) – React – Forms

67 အခန်း (၅) – React – Context Provider

79 အခန်း (၆) – React – UI Framework – MUI

97 အခန်း (၇) – React – More Hooks

116 အခန်း (၈) – React – React Router

144 အခန်း (၉) – JavaScript – Promises, async / await

154 အခန်း (၁၀) – Database – SQLite, Prisma ORM

178 အခန်း (၁၁) – Express – Back-end API

199 အခန်း (၁၂) – React – React Query

222 အခန်း (၁၃) – React Native – Expo, Expo Router

239 အခန်း (၁၄) – React Native – Sample App UI

270 အခန်း (၁၅) – API Authentication – Login, Register

311 အခန်း (၁၆) – Project – Social Media Features

357 အခန်း (၁၇) – JavaScript – WebSocket

369 အခန်း (၁၈) – Project – Notifications

390 အခန်း (၁၉) – Next.js – Server-Side Rendering Project

429 နိဂုံးချုပ်

433 စာရေးသူ၏ကိုယ်ရေးအကျဉ်း

မိတ်ဆက်

များပြားစုံလင်တဲ့ JavaScript နည်းပညာတွေဟာ အမြဲတိုးတက် ပြောင်းလဲနေလို့ မနည်းအမှီလိုက်ကြရလေ့ရှိပါတယ်။ Front-end, Back-end, Hybrid Mobile စသည်ဖြင့် ကဏ္ဍစုံသလို ဆန်းကျယ်ထူးခြားတဲ့ နည်းပညာတွေလည်း ဖြစ်နေကြပါတယ်။ ဒါကြောင့် နည်းပညာအရ အရမ်းဟော့ပြီး အခွင့်အလမ်းကောင်းတွေ ရှိနေပေမယ့် လေ့လာရတာ ခက်ခဲတဲ့ဘာသာရပ်တစ်ခုလို ဖြစ်နေတာပါ။

ဒီစာအုပ်မှာ အဲ့ဒီလို ခက်ခဲတဲ့ အကြောင်းအရာတွေကို နားလည်ရလွယ်ကူအောင် ဖော်ပြပါမယ်။ လက်တွေ့ကျကျလည်း ဖော်ပြမယ်။ Up to Date ဖြစ်အောင်လည်း ဖော်ပြသွားပါမယ်။ အကြောင်းအရာတွေများလို့ မလိုတာတွေ သိပ်အကျယ်ချဲ့မနေဘဲ၊ လိုတာတွေကိုတော့ စုံအောင်ထည့်သွင်း ဖော်ပြပေးသွားမှာပါ။

ဒီစာအုပ်ကိုဖတ်ဖို့အတွက် စာဖတ်သူက အနည်းဆုံးအနေနဲ့ JavaScript အခြေခံ ရှိထားပြီးဖြစ်ဖို့ လိုပါတယ်။ လိုအပ်ရင် **JavaScript လိုတိုရှင်း** စာအုပ်ကို အောက်ပါလိပ်စာမှာရယူပြီး လေ့လာနိုင်ပါတယ်။

https://eimaung.com/jsbook

ဒီစာအုပ်မှာပါဝင်တဲ့ ကုဒ်နမူနာတွေအားလုံးကိုလည်း အောက်ပါလိပ်စာမှာ ရယူနိုင်ပါတယ်။ ကုဒ်တွေများလို့ လိုက်လုပ်ရင်း အမှားရှာရခက်တာမျိုး ကြုံလာတဲ့အခါ ပြန်တိုက်ကြည့်ဖို့ပါ။ နမူနာတွေကို Step-by-Step တစ်ဆင့်ချင်း၊ ကြိုးစားပြီး ရအောင်လိုက်လုပ်ကြည့်ပါ။

https://github.com/eimg/rsd25

အခန်း (၁) - JavaScript - Node, NPM

များပြားကျယ်ပြန့်လှတဲ့ JavaScript နည်းပညာတွေ အားလုံးရဲ့ ဗဟိုချက်ဟာ Node ပါပဲ။ Node ဟာ JavaScript Runtime ဖြစ်ပါတယ်။ တိုတိုပြောရရင် JavaScript ကုဒ်တွေကို Run ပေးနိုင်တဲ့ နည်းပညာ ဆိုပါတော့။

အခုနောက်ပိုင်းမှာ Node နဲ့ အပြိုင် တခြား JavaScript Runtime နည်းပညာသစ်တွေ ထွက်ပေါ်လာကြပါတယ်။ အဓိကအကျဆုံးကတော့ Deno နဲ့ Bun တို့ပါပဲ။ အလားအလာရှိတဲ့ ရွေးချယ်စရာ နည်းပညာတွေ အနေနဲ့ ဗဟုသုတ ထည့်မှတ်ဖို့ပါ။ ဒီစာကိုရေးနေချိန်မှာ သူတို့ကို အဓိကနည်းပညာအနေနဲ့ သုံးနေကြတဲ့ အထိတော့ မရောက်ကြသေးပါဘူး။

Node – https://nodejs.org/

Deno – https://deno.com/

Bun – https://bun.sh/

Deno ဆိုတဲ့ စကားလုံးကို သေချာကြည့်ရင် Node ဆိုတဲ့စကားလုံးကိုပဲ ရှေ့နောက်ပြောင်းထားတာ ဖြစ်ကြောင်း တွေ့နိုင်ပါတယ်။ Node ကို စတင်တီထွင်ခဲ့သူကပဲ ဆက်လက်တီထွင်ထားတဲ့ ပရောဂျက်ပါ။ Node က ပေါ်တာ ကြာပြီဆိုတော့ တချို့အပိုင်းတွေမှာ ဒိတ်အောက်နေပြီလို့ ယူဆကြသူတွေ ရှိပါတယ်။ ဥပမာ – TypeScript တို့ JSX တို့လို နည်းပညာတွေဟာ ကနေ့ခေတ်မှာ အဓိကနည်းပညာတွေ ဖြစ်နေကြပေမယ့်၊ ဒီနည်းပညာတွေကို Node က တိုက်ရိုက် Run မပေးနိုင်ပါဘူး။ Run ပေးနိုင်တဲ့ တစ်ဆင့်ခံ နည်းပညာတွေကို အသုံးပြုကြရပါတယ်။ Deno တို့ Bun တို့မှာတော့ Runtime ကိုယ်တိုင်က ဒီနည်းပညာတွေ

ကို တိုက်ရိုက်နားလည် အလုပ်လုပ်ပေးနိုင်ကြပါတယ်။ တခြား အားသာချက် အားနည်းချက်တွေ ရှိသေးပေမယ့် အကျယ်မချဲ့တော့ပါဘူး။ ဒီနေရာမှာတော့ အဓိကလေ့လာသွားချင်တဲ့ အကြောင်းအရာတွေကိုပဲ ဆက်ကြည့်ကြပါမယ်။

Node Version Manager (NVM)

ဒီစာရေးနေချိန်မှာ နောက်ဆုံးထွက်ထားတာက Node 22.2.0 ဖြစ်ပါတယ်။ နည်းပညာသစ်တွေ စမ်းသပ်လိုရင် နောက်ဆုံး Version ကို သုံးတာကောင်းပေမယ့်၊ လက်တွေ့ပရောဂျက်တွေအတွက် ဆိုရင်တော့ ပိုပြီးတော့ Stable ဖြစ်တဲ့ Long-Term Support (LTS) Version တွေကို သုံးရပါတယ်။ လက်ရှိ LTS Version က Node 20.14.0 ပါ။ ပြီးတော့ ကိုယ်အသုံးပြုလိုတဲ့ တချို့နည်းပညာတွေက Version အတိအကျနဲ့ လိုအပ်တာမျိုးလည်း ရှိတတ်ပါသေးတယ်။ ဒါကြောင့် Node ကို ကိုယ့်စက်ထဲမှာ Install လုပ်တဲ့အခါ Version တစ်ခုကို ရွေးပြီး Install လုပ်တဲ့နည်းလမ်းဟာ သိပ်အဆင်မပြေပါဘူး။ Node Version Manager (NVM) ရဲ့ အကူအညီနဲ့ Install လုပ်ခြင်းကသာ ပိုပြီးတော့ အဆင်ပြေတဲ့နည်းလမ်း ဖြစ်ပါတယ်။

Mac (သို့မဟုတ်) Linux အသုံးပြုနေသူတွေက အောက်ပါလိပ်စာမှာ ကြည့်ပြီး NVM ကို အရင်ဆုံး Install လုပ်ထားသင့်ပါတယ်။

NVM – https://github.com/nvm-sh/nvm

Install လုပ်ရတာ လွယ်ကူပါတယ်။ သူ့ Page မှာ ဖော်ပြထားတဲ့ Installing & Updating Section မှာ ပေးထားတဲ့ ဒီ Command ကို ကူးယူပြီး Terminal မှာ Run ပေးလိုက်ယုံပါပဲ။

```
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh | bash
```

စာဖတ်သူ အဆင်ပြေစေဖို့ Command ကို ထည့်ဖော်ပြထားပေမယ့် Update တွေဘာတွေ ရှိနိုင်လို့ သူ့မူရင်း Repo Page မှာ တိုက်ရိုက်ကြည့်ပြီးလုပ်ဖို့ အကြံပေးပါတယ်။

Windows အသုံးပြုသူတွေကတော့ Installer ဖိုင်ကို အောက်ပါလိပ်စာမှာ ဒေါင်းနိုင်ပါတယ်။

NVM Installer (Windows) – https://github.com/coreybutler/nvm-windows

ပြီးတဲ့အခါ Installer ဖိုင်ကို Run ပြီးထည့်သွင်းရမှာပါ။

NVM ထည့်သွင်းပြီးပြီဆိုရင် လိုချင်တဲ့ Node Version ကို အခုလို Install လုပ်ယူလို့ ရပါတယ်။

```
nvm install 20
nvm install 22
```

Install လုပ်ထားတဲ့ Version တွေထဲက အသုံးပြုလိုတဲ့ Version ကို အခုလို ရွေးချယ်နိုင်ပါတယ်။

```
nvm use 22
```

Install မလုပ်ရသေးတဲ့ Version ကို `use` လုပ်လိုက်ရင်လည်း တစ်ခါထဲ Install လုပ်ပေးသွားပါလိမ့်မယ်။ ဒီနည်းနဲ့ အသုံးပြုလိုတဲ့ Node Version တွေကို အလွယ်တကူ ပြောင်းလဲ အသုံးပြုလို့ ရသွားပါပြီ။

Mac နဲ့ Linux မှာ Install လုပ်ထားတဲ့ Version တွေထဲက တစ်ခုကို Default အနေနဲ့ သတ်မှတ်ချင်ရင် အခုလို သတ်မှတ်နိုင်ပါတယ်။

```
nvm alias default 22
```

သတိပြုရမှာကတော့ Node Version တစ်ခုနဲ့ Install လုပ်ထားတဲ့ Global Package တွေရှိရင် တခြား Node Version ပြောင်းလိုက်တဲ့အခါ ပြန် Install လုပ်ပေးဖို့ လိုပါလိမ့်မယ်။ ဒီတစ်ချက်ကို သတိပြုလိုက်ရင် ကျန်တာအကုန် အဆင်ပြေသွားပါပြီ။

Node နဲ့ အလုပ်တစ်ခုခုမလုပ်ခင် လက်ရှိ အသက်ဝင်နေတဲ့ Node Version ကို စစ်ကြည့်သင့်ပါတယ်။

```
node -v
v22.2.0
```

နမူနာအရ လက်ရှိ အသက်ဝင်နေတာ Node 22.2.0 ဖြစ်ပါတယ်။

Node Package Manager (NPM)

ဒီတစ်ခါဆက်ကြည့်မှာ NVM မဟုတ်တော့ပါဘူး။ NPM ဖြစ်သွားပါပြီ။ Node ရဲ့ Default Package Manager ပါ။ ကိုယ့်ပရောဂျက်မှာ ထည့်သွင်းအသုံးပြုလိုတဲ့ Package တွေနဲ့ Dependencies ခေါ် လိုအပ်တဲ့ ဆက်စပ် Package တွေကို NPM နဲ့ Download ရယူနိုင်ပါတယ်။

NPM အစား အစားထိုးရွေးချယ်စရာ နည်းပညာတွေလည်း ရှိကြပါတယ်။ အဓိကအကျဆုံးကတော့ yarn နဲ့ pnpm တို့ဖြစ်ကြပါတယ်။ yarn ဟာ NPM ရဲ့ အားနည်းချက်တွေကို ဖြေရှင်းဖို့ ထွက်ပေါ်လာတဲ့ နည်းပညာတစ်ခုပါ။ Package တွေကို Parallel Download နဲ့ ပြိုင်တူဒေါင်းပေးလို့ ပိုမြန်ခြင်း အပါအဝင် တခြားအားသာချက်တွေ ရှိတဲ့အတွက်၊ ပေါ်ခါစကဆိုရင် NPM အစား သူ့ကို ပြောင်းသုံးသင့်တယ်လို့ ဆိုနိုင်တဲ့ နည်းပညာတစ်ခုပါ။ ဒါပေမယ့် နောက်ပိုင်းမှာ NPM ကလည်း yarn ကို နမူနာယူပြီး သူ့အားနည်းချက်တွေကို ပြုပြင်ပေးလာတဲ့အတွက် သိပ်မကွာတော့ပါဘူး။ ဆိုလိုတာက yarn ရဲ့ အားသာချက်တွေ ရှိပေမယ့် NPM ကိုပဲ ဆက်သုံးမယ်ဆိုရင် ဘာပြဿနာမှမရှိပါဘူး၊ သုံးလို့ရပါတယ်။

pnpm ကတော့ Package တွေကို လိုတိုင်း ပြန်ဒေါင်းပြီး ထည့်မနေတော့ဘဲ ဒေါင်းပြီးသားကနေ Shortcut လုပ်ပြီး ချိတ်လို့ရရင် ချိတ်ပေးပါတယ်။ ဒါကြောင့် နေရာယူ သိသိသာသာ သက်သာသွားစေတဲ့ နည်းပညာပါ။ Monorepo ခေါ် ပရောဂျက်မှာပါတဲ့ Front-end, Back-end အားလုံးကို တစ်စုထဲ Manage လုပ်တဲ့ ပရောဂျက်အမျိုးအစားတွေမှာ သုံးလို့ပိုကောင်းနိုင်ပါတယ်။

NPM – https://www.npmjs.com/

yarn – https://yarnpkg.com/

pnpm – https://pnpm.io/

ဒီစာအုပ်မှာတော့ NPM ကိုပဲ အသုံးပြုဖော်ပြသွားမှာပါ။ Node ကို Install လုပ်ချိန်မှာ NPM ဟာ တစ်ခါထဲ ပါဝင်သွားပြီးဖြစ်ပါတယ်။ ထပ်တောင်ထည့်စရာ မလိုတော့ပါဘူး။ နောက်သင်ခန်းစာတွေ မဆက်ခင် သူ့ရဲ့ အလုပ်လုပ်ပုံနဲ့ သဘာသဘာဝကို လက်တွေ့ စမ်းကြည့်သင့်ပါတယ်။

ပထမဆုံး ဖိုဒါအသစ်တစ်ခု တည်ဆောက်လိုက်ပါ။ အဲ့ဒီဖိုဒါထဲမှာ Terminal ကို ဖွင့်ပါ။

Tip – Windows အသုံးပြုသူတွေဟာ Windows ရဲ့ Default ဖြစ်တဲ့ Command Prompt ကို အသုံးမပြု သင့်ပါဘူး။ Windows Terminal App ကို အစားထိုး အသုံးပြုသင့်ပါတယ်။ မရှိသေးရင် Windows Store မှာပဲ ရှာပြီးတော့ Install လုပ်နိုင်ပါတယ်။ Microsoft က ဖန်တီးပေးထားတဲ့ ပိုကောင်းတဲ့ Terminal ပါ။

ဖိုဒါအသစ်တစ်ခုမှာ Terminal ဖွင့်ပြီးရင် အခုလို စမ်းကြည့်နိုင်ပါတယ်။

```
npm init -y
```

ဒါ လက်ရှိဖိုဒါကို NPM Package တစ်ခုဖြစ်သွားအောင် လုပ်ပေးလိုက်တာပါ။ နောက်ဆုံးက -y Option မ ထည့်ရင် သူက မေးခွန်းတွေ မေးပါလိမ့်မယ်။ ဖြေပေးဖို့လိုပါတယ်။ စမ်းချင်ရင် မထည့်ဘဲ စမ်းကြည့်လို့ ရပါ တယ်။ လက်ရှိနမူနာမှာတော့ မေးခွန်းတွေ တစ်ခုချင်း ဖြေစရာမလိုအောင် "အားလုံး Yes, Default Value အတိုင်းပဲသုံးမယ်" ဆိုတဲ့ အဓိပ္ပါယ်နဲ့ -y Option ထည့်ပေးလိုက်တာပါ။ ဒီ Command ကို Run လိုက်ရင် `package.json` ဆိုတဲ့ဖိုင်တစ်ခု ရပါလိမ့်မယ်။ ဖွင့်ကြည့်လိုက်ရင် အခုလိုဖြစ်နိုင်ပါတယ်။

JSON - package.json

```json
{
  "name": "chapter-1",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
```

```
  "author": "",
  "license": "ISC",
  "description": ""
}
```

Package Name, Version, License စသည်ဖြင့် အချက်အလက်တွေပါဝင်ပြီး လိုအပ်သလို ပြင်လို့ ရပါတယ်။ Version နံပါတ်တွေ ပေးပုံပေးနည်းနဲ့ပက်သက်ပြီး အသေးစိတ်ကို အောက်ပါလိပ်စာမှာ လေ့လာကြည့်နိုင်ပါတယ်။

Semantic Versioning (semver) – https://semver.org

ဒီနေရာမှာ ထည့်သွင်းသတိပြုသင့်တာကတော့ scripts ဖြစ်ပါတယ်။ အခုလိုပြင်ပြီး စမ်းကြည့်ပါ။

JSON - package.json

```
{
  "name": "chapter-1",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "dev": "echo \"Running dev scripts\"",
    "build": "echo \"Running build scripts\""
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "description": ""
}
```

နမူနာအရ `scripts` တွေထဲမှာ `dev` နဲ့ `build` ပါဝင်သွားခြင်း ဖြစ်ပါတယ်။ `package.json` ထဲမှာ ရေးထားတဲ့ Scripts တွေကို အခုလို Run လို့ရပါတယ်။

```
npm run dev
> chapter-1@1.0.0 dev
> echo "Running dev scripts"

Running dev scripts
```

```
npm run build
> chapter-1@1.0.0 build
> echo "Running build scripts"

Running build scripts
```

ဘာအလုပ်မှတော့ ထူးထူးဆန်းဆန်း မလုပ်သေးပါဘူး။ Output စာလေး ရိုက်ထုတ်ပြယုံလောက်ပါပဲ။ ဒီနေရာမှ လုပ်ဆောင်ချက်ထက် `package.json` ဖိုင်ထဲမှာ Scripts တွေထည့်ရေးလို့ရတယ်၊ လိုအပ်ရင် ပြန် Run လို့ရတယ်ဆိုတဲ့ သဘောသဘာဝကို အဓိကထား သတိပြုရမှာပါ။

လက်ရှိဖိုဒါထဲမှာ ထည့်သွင်းအသုံးပြုလိုတဲ့ JavaScript Package တွေကို အခုလို Download ရယူထည့်သွင်းနိုင်ပါတယ်။

```
npm install react
```

(သို့မဟုတ်)

```
npm i react
```

Command မှာ `install` လို့အပြည့်အစုံရေးရင်ရသလို အတိုကောက် `i` လို့ပဲရေးရင်လည်းရပါတယ်။ ဒါ `react` ကို ကိုယ့်ပရောဂျက်ထဲမှာ ထည့်သွင်းလိုက်တာပါပဲ။ ဒီလိုထည့်သွင်းလိုက်တဲ့အခါ ဖိုဒါထဲမှာ `node_modules` အမည်နဲ့ ဖိုဒါတစ်ခု ဝင်သွားမှာဖြစ်ပြီး `package-lock.json` အမည်နဲ့ ဖိုင်တစ်ခုလည်း ထပ်တိုးလာမှာ ဖြစ်ပါတယ်။ ပြီးတော့ `package.json` မှာ အခုလို `dependencies` Section တစ်ခုပါသွားတာကို တွေ့ရပါလိမ့်မယ်။

JSON - package.json

```json
…
"dependencies": {
  "react": "^18.3.1"
}
…
```

ဒီနည်းနဲ့ NPM က ကိုယ့်ပရောဂျက်ရဲ့ Dependencies တွေကို မှတ်ထားပါတယ်။ တနည်းအားဖြင့် ဒီပရောဂျက်ဟာ React ရှိမှ အလုပ်လုပ်တဲ့ပရောဂျက်ဆိုတာကို မှတ်ထားလိုက်တာပါ။

Install လုပ်လိုက်တဲ့ Package Source Code တွေကို `node_modules` ဖိုဒါထဲမှာ ထည့်သိမ်းသွားပြီး၊ ကိုယ် Install လုပ်လိုက်တဲ့ Package အတွက်လိုအပ်တဲ့ ဆက်စပ် Dependencies Package တွေကိုလည်း တစ်ခါထဲ အလိုအလျှောက် ဒေါင်းပြီးထည့်ပေးသွားတာကို တွေ့ရနိုင်ပါတယ်။ ကိုယ့်ပရောဂျက်အတွက် React က Dependencies ဖြစ်သွားပြီဖြစ်သလို၊ React မှာလည်း သူ့ Dependencies တွေ ရှိနေမှာမို့လို့ပါ။ `package-lock.json` ဖိုင်ထဲမှာတော့ ထည့်သွင်းလိုက်သမျှ Package နဲ့ Dependencies တိုင်းရဲ့ Version အတိအကျကို မှတ်တမ်းတင်ထားပါလိမ့်မယ်။

Dependencies တွေမှာ Development Dependencies ဆိုတာလည်း ရှိပါသေးတယ်။ အခုလို စမ်းကြည့်နိုင်ပါတယ်။

```
npm i jest -D
```

-D Option နဲ့ အခု Install လုပ်တဲ့ `jest` ဟာ Development Dependencies ဖြစ်ကြောင်း ပြောလိုက်တာပါ။ အခုနေ `package.json` ဖိုင်ကို ကြည့်လိုက်ရင် အခုလိုထပ်တိုးလာတာကို တွေ့ရနိုင်ပါတယ်။

JSON - package.json

```
…
"dependencies": {
  "react": "^18.3.1"
},
"devDependencies": {
  "jest": "^29.7.0"
}
…
```

`react` က ရိုးရိုး Dependencies ဖြစ်ပြီး `jest` က devDependencies ဖြစ်နေတာပါ။ Development Dependencies ဆိုတာ Development Time မှာပဲ လိုအပ်တဲ့ Dependencies အမျိုးအစားပါ။ တနည်းအားဖြင့် ဒီပရောဂျက် Run ဖို့ လိုအပ်တဲ့ Dependencies တွေမဟုတ်ကြဘဲ ဒီ ပရောဂျက်ကို ရေးသားနေစဉ်မှာ လိုအပ်တဲ့ Dependencies ဆိုတဲ့အဓိပ္ပါယ်ပါ။ `jest` ဆိုတာ Testing Library ပါ။ ပရောဂျက်ကို Test လုပ်ဖို့ သူ့ကိုလိုပါတယ်။ ဒါပေမယ့် ပရောဂျက်ကို Run ဖို့တော့ သူမလိုပါဘူး။ ဒါကြောင့် `jest` လို Package မျိုးကို ရိုးရိုး Dependencies အနေနဲ့မထည့်ဘဲ Development Dependencies အနေနဲ့ ထည့်ကြရတာပါ။

တချို့ Install လုပ်ထားတဲ့ Package တွေက Run လို့ရပါတယ်။ ဥပမာ –

```
npx jest
No tests found, exiting with code 1
```

နမူနာအရ စောစောက Install လုပ်ထားတဲ့ `jest` ကို Run လိုက်တာပါ။ `npm` မဟုတ်တော့ပါဘူး `npx` ဖြစ်သွားပါပြီ။ ဘာအလုပ်မှတော့ မလုပ်သေးပါဘူး။ ဒီနေရာမှာ အလုပ်လုပ်ပုံထက် Install လုပ်ထားတဲ့ Package တွေကို `npx` နဲ့ Run လို့ရတယ်ဆိုတဲ့ သဘောသဘာဝကို အဓိကမှတ်ရမှာပါ။ Scripts တွေကို `npm run` နဲ့ Run ရပြီး၊ Package တွေကို `npx` နဲ့ Run ရပါတယ်။ အဲ့ဒီနှစ်ခုကို ရောပြီး လည်မသွားအောင်သာ မှတ်ထားပါ။

Install လုပ်လိုက်တဲ့ Package တွေကို လက်ရှိပရောဂျက်ဖိုဒါထဲမှာပဲ သိမ်းသွားလို့ Local Package လို့ခေါ်နိုင်ပါတယ်။ Global Package အနေနဲ့လည်း Install လုပ်လို့ရပါတယ်။ ဒီလိုပါ –

```
npm i -g nodemon
```

နမူနာအရ nodemon လို့ခေါ်တဲ့ Package ကို -g Option ပေးပြီး Global Package အနေနဲ့ Install လုပ်လိုက်တာပါ။ ဒီလို Global Package အနေနဲ့ Install လုပ်ထားတဲ့ Package တွေကို လက်ရှိကွန်ပျူတာမှာ ကြိုက်တဲ့နေရာက ယူသုံးလို့ရသလို၊ ကြိုက်တဲ့နေရာကနေလည်း Run လို့ရပါတယ်။ ဥပမာ –

```
nodemon
```

ရှေ့က npx တောင် ထည့်စရာမလိုတော့ဘဲ လှမ်း Run လို့ရသွားပါတယ်။ ဘာအလုပ်မှတော့ လုပ်ပေးဦးမှာ မဟုတ်သေးပါဘူး။ ထုံးစံအတိုင်း ဒီလို Install လုပ်ပြီး ဒီလို Run လို့ရတယ်ဆိုတဲ့ သဘောကို ဒီနေရာမှာ အရင်ကမှတ်ရမှာပါ။

လိုတဲ့နေရာတိုင်းကနေ Run လို့ရဖို့လိုတဲ့ Package တချို့ကလွဲရင် အများအားဖြင့် ပရောဂျက်တစ်ခုချင်းစီအလိုက် ကိုယ့်ပရောဂျက်မှာလိုတဲ့ Package တွေကို Local Package အနေနဲ့ သီးခြားစီပဲ ထည့်သုံးဖို့ တိုက်တွန်းပါတယ်။ Version အသေးစိတ် အတိအကျ ပရောဂျက်တစ်ခုနဲ့တစ်ခု လိုအပ်ချက်ကွာနိုင်လို့ ကိုယ့် Package နဲ့ကိုယ် သုံးတာက ပိုကောင်းမှာမို့လို့ပါ။

ဒီလောက်ဆိုရင် NPM ရဲ့ သဘောသဘာဝနဲ့ အသုံးပြုနည်းအကြောင်းကို သိသွားပါပြီ။ ဆက်လက်ပြီး သူ့အကူအညီနဲ့ ပရောဂျက်တွေ ဖန်တီးလေ့လာသွားကြပါမယ်။

အခန်း (၂) - React - New React Project with Vite

React ပရောဂျက်တစ်ခု မစခင် အရင်ဆုံး Client-Side နဲ့ Server-Side အကြောင်းလေး ပြောချင်ပါတယ်။ ဘာဖြစ်လို့လည်းဆိုတော့ React ဟာ မူလနည်းပညာအရ Client-Side သီးသန့် နည်းပညာတစ်ခု ဖြစ်ပေမယ့် အခုနောက်ပိုင်းမှာ Server-Side Rendering (SSR) တို့ Server Components တို့လို ဖြည့်စွက်နည်းပညာတွေကြောင့် Client-Side လား၊ Server-Side လား မသဲကွဲတော့ဘဲ နဂိုကမှ နားလည်ရခက်ရတဲ့ထဲ ပိုပြီးတော့ နားလည်ရခက်သွားသလို ဖြစ်နေပါတယ်။

ဒါကြောင့် အခုကတည်းက အခြေခံလေးတွေကို ရှင်းအောင် ကြိုပြောထားမှ နောက်ပိုင်းမှာ ဆက်ကြည့်ရတာ အဆင်ပြေမှာမို့လို့ပါ။

စမ်းကြည့်ဖို့အတွက် ပရောဂျက်ဖိုဒါအသစ်တစ်ခု တည်ဆောက်ပြီး သူ့ထဲမှာ `npm init -y` ကို Run ပေးလိုက်ပါ။ ပြီးတဲ့အခါ `a.js` ဆိုတဲ့ ဖိုင်တစ်ခုနဲ့ ဒီကုဒ်တွေ ရေးပေးလိုက်ပါ။

JSON - a.js

```
const { createServer } = require("node:http");

const server = createServer((req, res) => {
    res.statusCode = 200;
    res.setHeader("Content-Type", "text/html");
```

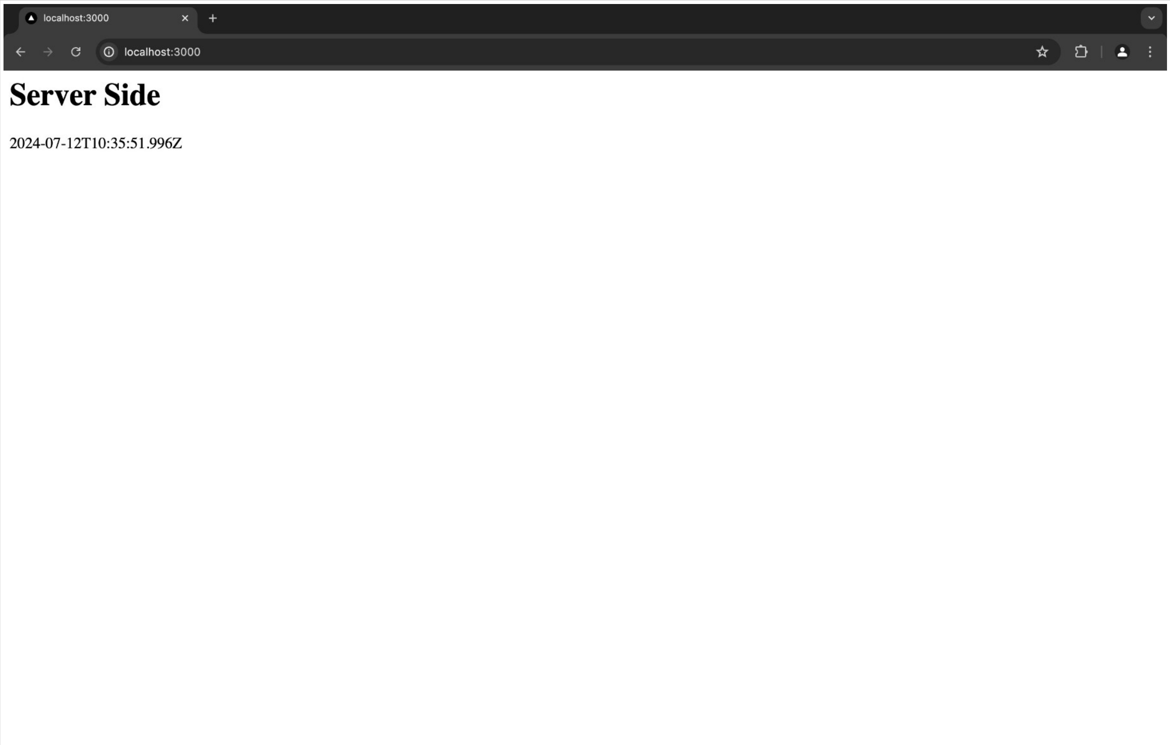
```
    const today = new Date();
    const content = `
      <h1>Server Side</h1>
      <p>${today.toISOString()}</p>
    `;

    res.end(content);
});

server.listen(3000, "localhost", () => {
    console.log("Server running at 3000. Press Ctrl+C to stop");
});
```

ဒါ Node ရဲ့ `http` Module ကို အသုံးပြုပြီး `createServer()` နဲ့ Web Server တစ်ခုကို ဖန်တီးလိုက်တာပါ။ `res.end()` ကိုသုံးပြီး လက်ရှိရက်စွဲနဲ့ အချိန်ကို HTML Elements တချို့နဲ့အတူ ပြန်ပေးထားပါတယ်။

နောက်တစ်ဆင့်မှာ Port နံပါတ် `3000` နဲ့ `"localhost"` တို့ကို ပေးပြီး အထက်မှာ Create လုပ်ထားတဲ့ Server ကို Run ခိုင်းထားပါတယ်။ အခုလို စမ်းကြည့်နိုင်ပါတယ်။

```
node a.js
Server running at 3000. Press Ctrl+C to stop.
```

ပြီးတဲ့အခါ Browser မှာ http://localhost:3000 ကို ရိုက်ထည့်လိုက်မယ်ဆိုရင် ပေးထားတဲ့အတိုင်း လက်ရှိ ရက်စွဲအချိန်ကို HTML Elements တချို့နဲ့အတူ မြင်ရမှာပဲဖြစ်ပါတယ်။



ဒါဟာ လုပ်စရာရှိတဲ့ အလုပ်တွေကို Server ဘက်မှာ အကုန်လုပ်တဲ့ Server-Side ကုဒ်ဖြစ်ပါတယ်။ ဒါကြောင့် Browser မှာ Right Click → View Page Source ကို ခေါ်ကြည့်ရင် အခုလို တွေ့ရပါလိမ့်မယ်။

Output - View Page Source

```
<h1>Server Side</h1>
<p>2024-06-07T08:05:21.028Z</p>
```

JavaScript ကုဒ်တွေမပါဘဲ နောက်ဆုံးရလဒ်ကိုသာ Browser က လက်ခံရရှိတဲ့သဘောပဲ ဖြစ်ပါတယ်။

ဆက်လက်ပြီး နောက်ထပ် `b.js` ဆိုတဲ့အမည်နဲ့ ဖိုင်တစ်ခုထပ်တိုးပြီး ဒီကုဒ်ကို ရေးကြည့်ပါ။

JSON - b.js

```
const { createServer } = require("node:http");

const server = createServer((req, res) => {
    res.statusCode = 200;
    res.setHeader("Content-Type", "text/html");

    const content = `
        <h1>Client Side</h1>
        <p>
          <script>
            const today = new Date();
            document.write(today.toISOString());
          </script>
        </p>
    `;

    res.end(content);
});

server.listen(3000, "localhost", () => {
    console.log("Server running at 3000. Press Ctrl+C to stop.");
});
```

စောစောကကုဒ်နဲ့ အများအားဖြင့် အတူတူပါပဲ။ ထူးခြားချက်အနေနဲ့ ဒီကနေ့ ရက်စွဲနဲ့အချိန်ကို တွက်တဲ့ ကိစ္စတွေ၊ ရိုက်ထုတ်တဲ့ကိစ္စတွေကို တစ်ခါထဲမလုပ်တော့ဘဲ Response ပြန်ပေးတဲ့ Content ထဲမှာ <script> အဖွင့်အပိတ်နဲ့ ထည့်ပေးလိုက်တာပါ။ ဒါကြောင့် Browser က လက်ခံရရှိတဲ့အခါ အဲ့ဒီ JavaScript ကုဒ်တွေကိုပါ လက်ခံရရှိမှာ ဖြစ်ပါတယ်။

Terminal မှာ စောစောက စမ်းလက်စကို Ctrl + C နဲ့ ရပ်လိုက်ပြီး အခုလို ပြန် စမ်းကြည့်လိုက်ပါ။

```
node b.js
```

Browser မှာ http://localhost:3000 နဲ့ နောက်တစ်ကြိမ် ထပ်စမ်းကြည့်လိုက်ရင် စောစောကရတဲ့ ရလဒ်နဲ့ အတူတူပဲ ဆိုတာကို တွေ့ရပါလိမ့်မယ်။ ဒါပေမယ့် Browser မှာ Right Click → View Page Source ကို ကြည့်လိုက်ရင်တော့ တူမှာ မဟုတ်တော့ပါဘူး။ အခုလို ဖြစ်နေပါလိမ့်မယ်။

Output - View Page Source

```
<h1>Client Side</h1>
<p>
  <script>
    const today = new Date();
    document.write(today.toISOString());
  </script>
</p>
```

Browser က လက်ခံရရှိတဲ့ ရလဒ်ထဲမှာ JavaScript ကုဒ်တွေပါနေပြီး အဲ့ဒီ ကုဒ်တွေကို Browser က Run ပေးသွားတာပဲ ဖြစ်ပါတယ်။

ဒီနမူနာတွေကနေ ထုတ်ယူရမယ့် ကောက်ချက်ကတော့၊ JavaScript ကုဒ်တွေမှာ တချို့ကုဒ်တွေက Server ဘက်မှာ အလုပ်လုပ်တဲ့ Server-Side ကုဒ်တွေ ဖြစ်နိုင်ပြီး တချို့ကုဒ်တွေက Client ဘက်မှာ အလုပ်လုပ်တဲ့ Client-Side ကုဒ်တွေ ဖြစ်နိုင်ပါတယ်။ Client-Side ကုဒ်နဲ့ Server-Side ကုဒ်တွေကို လိုအပ်ရင် ပူးတွဲရောနှောပြီးတော့လည်း အသုံးပြုနိုင်တယ် ဆိုတဲ့အချက်ပဲ ဖြစ်ပါတယ်။

React ဟာ မူလက Client-Side နည်းပညာဖြစ်ပါတယ်။ App အလုပ်လုပ်ဖို့ အတွက် လိုအပ်တဲ့ ကုဒ်တွေ အကုန်လုံး Browser ဆီကို ရောက်ပြီးမှ Browser က Run စရာရှိတာ Run ပေးတာပါ။ အခုနောက်ပိုင်းမှာ Server-Side Rendering နည်းပညာနဲ့ တချို့ အလုပ်တွေကို Server ဘက်မှာ ကြိုလုပ်ပေးနိုင်လာပါတယ်။ ဒီနည်းပညာတွေဟာ ရှုပ်ထွေးဆန်းကျယ်ကြပြီး နောက်ပိုင်းအခန်းတွေမှာ ဆက်လေ့လာကြပါမယ်။ အစောပိုင်းမှာတော့ React ရဲ့ မူလနည်းပညာ သဘာဝအတိုင်း Client-Side နည်းပညာတစ်ခုအနေနဲ့ အဓိကထားပြီး လေ့လာကြမှာပါ။

React Project

React ကို တခြား အစဉ်အလာ JavaScript Library များနည်းတူ HTML Document တွေထဲမှာ `<script>` Element နဲ့ တိုက်ရိုက်ချိတ်ပြီး သုံးမယ်ဆိုရင် ရနိုင်ပေမယ့်၊ စမ်းကြည့်ယုံလောက်ပဲ ရမှာဖြစ်ပြီး လက်တွေ့အသုံးပြုဖို့အတွက် အဆင်မပြေနိုင်ပါဘူး။

JSX လို့ခေါ်တဲ့ နည်းပညာကြောင့် ဖြစ်ပါတယ်။ JSX ဟာ ထူးဆန်းပြီး အသုံးဝင်တဲ့ နည်းပညာပါ။ React ကို အခုမှစလေ့လာမယ့်သူတွေအတွက် လက်ခံရခက်တဲ့ နည်းပညာဖြစ်နေနိုင်ပေမယ့်၊ React ရယ်လို့ လွန်ခဲ့တဲ့ဆယ်စုနှစ်အတွင်း အဓိကအကျဆုံး နည်းပညာ ဖြစ်လာခဲ့တာ သူ့ကြောင့်လည်း ပါပါတယ်။

ဥပမာ – React ရိုးရိုးနဲ့ Element တစ်ခုကို အခုလို ဖန်တီးယူလို့ ရနိုင်ပါတယ်။

React - Plain JavaScript

```
function Note() {
  return React.createElement("p", { id: "note" }, "Some Content");
}
```

ဒါ HTML <p> Element တစ်ခုကို လိုအပ်တဲ့ Attribute တွေ Content တွေနဲ့အတူ ဖန်တီးလိုက်တာပါ။ `createElement()` အတွက် Element Name, Attributes နဲ့ Children တို့ကိုပေးပြီး ဖန်တီးယူရပါတယ်။ အကယ်၍ <p> Element ထဲမှာ <b> Element တစ်ခု ထပ်ပါစေချင်ရင် ရေးရမယ့်ကုဒ်က ဒီလိုဖြစ်နိုင်ပါတယ်။

React - Plain JavaScript

```
function Note() {
  let b = React.createElement("b", null, "Some Content");
  return React.createElement("p", { id: "note" }, b);
}
```

ဒါမှမဟုတ် ဒီလိုရေးတာလည်း ဖြစ်နိုင်ပါတယ်။

React - Plain JavaScript

```
function Note() {
  return React.createElement(
    "p",
    { id: "note" },
    React.createElement(
      "b",
      null,
      "Some Content"
    )
  );
}
```

အတော်လေးကြည့်ရခက်တဲ့ကုဒ်လို့ ဆိုနိုင်ပါတယ်။ JSX နဲ့ဆိုရင် အလားတူကုဒ်ကို အခုလိုရေးလိုက်လို့ ရနိုင်ပါတယ်။

React - JSX

```
function Note() {
  return <p id="note">
    <b>Some Content</b>
  </p>
}
```

အများကြီးပိုရှင်းသွားပါပြီ။ ဘာမှမဆိုင်တော့ပါဘူး။ လိုချင်တဲ့ Element Structure အတိုင်း တိုက်ရိုက်ရေးပေးလိုက်တာပါ။ ဒီလို JavaScript ထဲမှာ HTML Element Structure အတိုင်း ကုဒ်တွေတိုက်ရိုက်ရေးလို့ရစေတဲ့ နည်းပညာကို JSX လို့ခေါ်တာပါ။ မူလ JavaScript ကတော့ ဒီကုဒ်ကို လက်ခံမှာ မဟုတ်ပါဘူး။ Syntax မှားနေတယ်လို့ ပြောပါလိမ့်မယ်။ ဒါကြောင့် ဒီ JSX ကုဒ်ကို ရိုးရိုး JavaScript ကုဒ်ဖြစ်အောင် ပြောင်းပေးတဲ့ Babel လို ကြားခံနည်းပညာတွေလိုအပ်ပါတယ်။ ရေးတဲ့အခါ JSX နဲ့ရေးရပေမယ့် Run တဲ့အခါ ရိုးရိုး JavaScript ပြောင်းပြီး Run ရတာပါ။

Babel – https://babeljs.io/

JSX ကုဒ်တွေကို JavaScript ကုဒ်ဖြစ်အောင် Compile လုပ်ယူပုံယူနည်းတွေ၊ ကိုယ့်ဘာသာ Setup လုပ်ပုံလုပ်နည်းတွေကို ဒီနေရာမှာ ထည့်မဖော်ပြတော့ပါဘူး။ အဲ့ဒီအလုပ်တွေကို ကိုယ့်ဘာသာ လုပ်စရာမလိုအပ်လို့ပါ။ အလိုအလျှောက် လုပ်ပေးနိုင်တဲ့ နည်းပညာတွေ ရှိနေပါတယ်။ အလုပ်လုပ်ပုံကို အတွင်းကျကျ ပိုနားလည်ချင်လို့ ဗဟုသုတအနေနဲ့ လေ့လာထားချင်တယ်ဆိုရင်တော့ အောက်ပါလင့်မှာ လေ့လာနိုင်ပါတယ်။

https://gist.github.com/eimg/50832314c7bfbc8d46ed65c44b9d76b5

ဒီအလုပ်တွေကို အလိုအလျှောက်လုပ်ပေးနိုင်တဲ့ နည်းပညာအနေနဲ့ အရင်ကတော့ create-react-app (CRA) လို့ခေါ်တဲ့နည်းပညာကို အသုံးပြုရပါတယ်။ CRA ဟာ နှေးတဲ့အတွက် အခုနောက်ပိုင်းမှာ Recommend မလုပ်ကြတော့ပါဘူး။ သူ့အစား Vite သို့မဟုတ် Next.js တို့ကို သုံးဖို့ Recommend လုပ်ကြပါတယ်။ Next.js အကြောင်းကို နောက်ပိုင်းအခန်းတွေမှာ ဆက်ကြည့်ကြမှာဖြစ်ပြီး ရှေ့ပိုင်းသင်ခန်းစာတွေမှာ Vite ကို အသုံးပြုပြီး ဆက်လက်ဖော်ပြသွားမှာဖြစ်ပါတယ်။

Vite ကိုအသုံးပြီး React ပရောဂျက်တစ်ခု တည်ဆောက်ဖို့အတွက် ပရောဂျက်ကုဒ်တွေ ထားလိုတဲ့ဖိုဒါရွေးပါ။ ပြီးရင် အဲ့ဒီဖိုဒါထဲမှာ Terminal ကိုဖွင့်ပြီး ဒီ Command နဲ့ တည်ဆောက်နိုင်ပါတယ်။

```
npm create vite hello-react

? Select a framework: › - Use arrow-keys. Return to submit.
    Vanilla
    Vue
❯   React
    Preact
    Lit
    Svelte
    Solid
    Qwik
    Others
```

`vite` ကို သုံးပြီး `hello-react` အမည်နဲ့ ပရောဂျက်အသစ် Create လုပ်မယ်လို့ ပြောလိုက်တာပါ။

အသုံးပြုလိုတဲ့ Framework ရွေးခိုင်းပါလိမ့်မယ်။ React ကိုရွေးပြီး Enter နှိပ်လိုက်ပါ။ နောက်တစ်ဆင့်မှာ Language ရွေးခိုင်းပါလိမ့်မယ်။

```
✔ Select a framework: › React
? Select a variant: › - Use arrow-keys. Return to submit.
    TypeScript
    TypeScript + SWC
❯   JavaScript
    JavaScript + SWC
```

TypeScript ဟာ လေ့လာသင့်တဲ့ နည်းပညာပါ။ ဒါပေမယ့် ဒီစာအုပ်မှာတော့ JavaScript နဲ့ပဲ ဆက်လက်လေ့လာသွားမှာပါ။ JavaScript + SWC ဆိုတဲ့ ရွေးချယ်စရာ ပါနေတာကို သတိပြုမိပါလိမ့်မယ်။ ရိုးရိုး JavaScript ကို ရွေးရင် Babel ကို အသုံးပြုသွားမှာဖြစ်ပြီး၊ SWC ကို ရွေးရင် Speedy Web Compiler ခေါ် Babel နဲ့ အလားတူနည်းပညာသစ်ကို အသုံးပြုသွားမှာပါ။ SWC က ပိုမြန်ပေမယ့် ပို Stable ဖြစ်တဲ့ရိုးရိုး JavaScript ကိုပဲရွေးလိုက်ပါ။

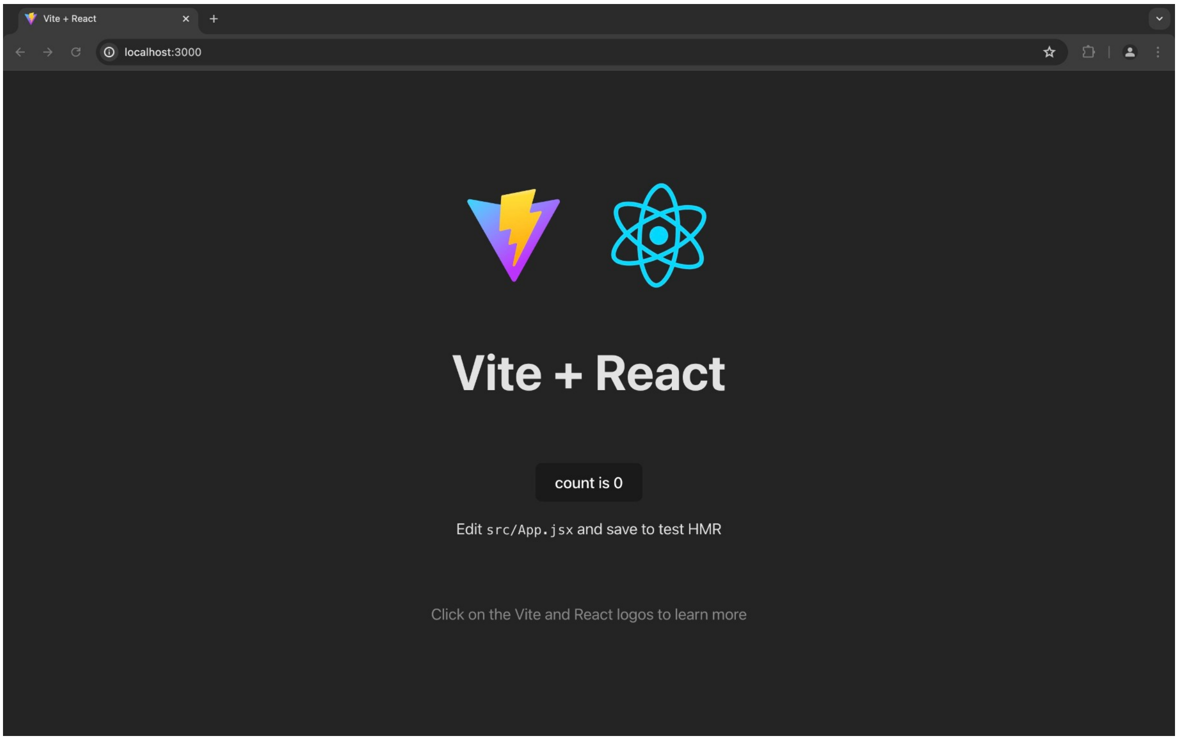
ပြီးတဲ့အခါ အသစ်တည်ဆောက်လိုက်တဲ့ ပရောဂျက်ဖိုဒါထဲကိုဝင်ပြီး လိုအပ်တဲ့ Package တွေကို Install လုပ်ပေးပါ။

```
cd hello-react
npm install
```

Package တွေ Install လုပ်ပြီးရင် React ပရောဂျက်သစ်တစ်ခု အသင့်ဖြစ်သွားပါပြီ။ အခုလို Run ကြည့်နိုင်ပါတယ်။

```
npm run dev

 VITE v5.2.13  ready in 180 ms

  ➜  Local:   http://localhost:5173/
  ➜  Network: use --host to expose
  ➜  press h + enter to show help
```

Browser မှာ http://localhost:5173 ကိုသွားပြီး စမ်းကြည့်လို့ရပါပြီ။



Run လိုက်တဲ့အခါ Browser ကို ကိုယ့်ဘာသာဖွင့်စရာမလိုဘဲ၊ အလိုအလျှောက် ဖွင့်ပေးစေချင်ရင်လည်း လုပ်ထားလို့ရပါတယ်။ လက်ရှိ Run လက်စကို Ctrl + C နဲ့ ရပ်လိုက်ပါ။ ပြီးရင် ပရောဂျက်ဖိုဒါထဲက `vite.config.js` ကိုဖွင့်ပြီး အခုလိုထပ်ဖြည့်ပေးလိုက်ပါ။

JavaScript - vite.config.js

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [react()],
  server: {
    port: 3000,
```

```
    open: true,
  }
})
```

`server` Section တစ်ခုထပ်ထည့်ပေးလိုက်တာပါ။ Port ကို 3000 လို့သတ်မှတ်ထားပြီး `open: true` လို့ ပြောထားပါတယ်။ ဒါကြောင့် အခုနေ `npm run dev` ကို ထပ် Run ကြည့်ရင် Port 3000 မှာ Run နေတဲ့ ပရောဂျက်ကို အလိုအလျှောက် Browser မှာတစ်ခါထဲ ဖွင့်ပေးသွားတာကို တွေ့ရပါလိမ့်မယ်။

အခန်း (၃) - React - Components, Props, States

React ဟာကျယ်ပြန့်တဲ့ နည်းပညာတစ်ခုဖြစ်နေပေမယ့် သူ့ရဲ့သဘောသဘာဝအကျဉ်းချုပ်ကို အခုလိုမှတ်နိုင်ပါတယ်။

- Component Framework – <u>ပြန်လည် အသုံးပြုနိုင်သော၊ ပေါင်းစပ် အသုံးပြုနိုင်သော UI Component တွေကို ဖန်တီးလို့ရတဲ့ နည်းပညာ</u>။
- Reactive – <u>State Data ကိုပြောင်းလိုက်ရင် UI ဖော်ပြပုံကို အလိုအလျှောက် ပြောင်းလဲပြီး ပြသပေးနိုင်တဲ့ နည်းပညာ</u>။

ဒါကြောင့် React အကြောင်းကို လေ့လာတယ်ဆိုတာ Component တွေ တည်ဆောက် အသုံးပြုပုံတွေ၊ State Data စီမံပုံတွေကို လေ့လာကြရမှာပါ။

Components & Props

React နဲ့ Component တစ်ခုတည်ဆောက်ရတာ လွယ်ကူပါတယ်။ Element တစ်ခုကို Return ပြန်ပေးတဲ့ Function အနေနဲ့ ရေးသားတည်ဆောက်ရပါတယ်။

React

```
function Item(props) {
  return <li>{props.name}</li>
}
```

ဒါ Item အမည်နဲ့ Component တစ်ခု တည်ဆောက်လိုက်တာပါပဲ။ Element တစ်ခုကို ပြန်ပေးမှာ ဖြစ်ပါတယ်။ JSX ကို အသုံးပြု ရေးသားထားပါတယ်။ တစ်ခါထဲ ချရေးကြည့်ဖို့ မလိုသေးပါဘူး။ ရှင်းလင်းချက်လေးတွေ နားလည်အောင်သာ လိုက်ဖတ်ထားပါ။ ခဏနေမှာ အားလုံးပေါင်းစပ်ပြီး လက်တွေ့သပ်သပ် လုပ်ကြည့်ကြပါမယ်။

အထက်က ကုဒ်နမူနာမှာ props (Properties) ကို Parameter အနေနဲ့ လက်ခံထားတာကိုလည်း တွေ့ရပါလိမ့်မယ်။ JSX Element တွေထဲမှာ JavaScript Expression တွေထည့်ရေးချင်ရင် တွန့်ကွင်းနဲ့ ရေးရပါတယ်။ ဒါကြောင့် props.name ကို အဖွင့်အပိတ်ကြားထဲမှာ ထည့်ပြချင်တဲ့အတွက် တွန့်ကွင်းနဲ့ ထည့်ပြထားတာကို တွေ့ရနိုင်ပါတယ်။ ဒီ Component ကို လိုအပ်တဲ့နေရာမှာ အခုလို ခေါ်ယူအသုံးပြုနိုင်ပါတယ်။

React

```
<Item name="Apple" />
```

ရေးတုံးက Function အဖြစ် ရေးထားပေမယ့် JSX Element တစ်ခုကဲ့သို့ ပြန်လည်အသုံးပြုနိုင်တာ ဖြစ်ပြီး၊ ခေါ်ယူအသုံးပြုစဉ်မှာ name ကို Props အဖြစ်အသုံးပြုဖို့ ထည့်ပေးထားတာကို တွေ့ရနိုင်ပါတယ်။ Component တည်ဆောက်ပုံရော၊ ပြန်လည်အသုံးပြုပုံကိုပါ နည်းနည်း ပြင်လိုက်ပါမယ်။

React

```
function Item(props) {
  return <li>
    {props.name} <i>(${props.price})</i>
  </li>
}
```

React

```
<Item name="Apple" price="0.99" />
```

Component ကို ရေးသားစဉ်မှာ `name` နဲ့ `price` ဆိုတဲ့ Props Value နှစ်ခုကို ထည့်သုံးပြီး ရေးထား သလိုပဲ၊ ပြန်ခေါ်သုံးတဲ့အခါမှာလည်း `name` နဲ့ `price` နှစ်ခုကို Props အဖြစ်သုံးဖို့ ထည့်ပေးထားတာကို တွေ့ရမှာပါ။ သတိပြုစရာလေးတချို့နဲ့ မှုကွဲလေးတချို့လည်း ထည့်ပြောပြချင်ပါတယ်။

React

```
function Component() {
  return (
    <a href="#">
      <i className="icon"></i>
      <span>Link Button</span>
    </a>
  )
}
```

ဒီနမူနာကနေ သတိပြုစရာက Element ကို Return ပြန်တဲ့အခါ JSX Element တွေကို ပိုပြီး Organize ဖြစ်သွားအောင် ဝိုက်ကွင်းအဖွင့်အပိတ်ထဲမှာ ထည့်ရေးပြီး ပြန်ပေးထားခြင်း ဖြစ်ပါတယ်။ ပထမနမူနာမှာလို ဝိုက်ကွင်းအဖွင့်အပိတ် မထည့်ဘဲရေးရင်လည်း ရသလို၊ လိုအပ်ရင်လည်း ထည့်ရေးနိုင်ပါတယ်။

နောက်ထပ်သတိပြုစရာကတော့ Element မှာ `class` Attribute အသုံးပြုလိုတဲ့အခါ `class` လို့မရေးရ ပါဘူး။ `className` လို့ရေးပေးရပါတယ်။ JSX ဟာ HTML Element ပုံစံဖြစ်နေပေမယ့် JavaScript ကုဒ် ဖြစ်ပါတယ်။ `class` ဆိုတာ JavaScript Keyword ဖြစ်နေတဲ့အတွက် သုံးလို့မရပါဘူး။ ဒါကြောင့် `className` လို့သုံးပြီး ရေးပေးရပါတယ်။ ဒီနမူနာမှာ Props မလိုအပ်လို့ Parameter ထည့်ယူမထားတာ ကိုလည်း သတိပြုရပါမယ်။

နောက်တစ်ခုလောက် ထပ်ကြည့်ပါဦးမယ်။

React

```
function Component() {
  return (
    <>
      <a href="#">
        <i className="save-icon"></i>
        <span>Link Button</span>
      </a>
      <hr />
    </>
  );
}
```

ဒီတစ်ခါလည်း ထူးခြားချက်နှစ်ချက်ပါ။ ပြန်ပေးချင်တဲ့ Element ကတစ်ခုထဲ မဟုတ်တော့ဘဲ `<a>` နဲ့ `<hr>` နှစ်ခုဖြစ်သွားပါပြီ။ Element နှစ်ခု Return ပြန်ပေးလို့မရပါဘူး။ ဒါကြောင့် React Fragment လို့ခေါ်တဲ့ Element အလွတ်တစ်ခုထဲမှာ ထည့်ပြီး ပြန်ပေးထားပါတယ်။ React Fragment ကို အပြည့်အစုံရေးမယ်ဆိုရင် `<React.Fragment>` လို့ရေးရမှာပါ။ နမူနာမှာ ပြထားသလို Element အလွတ်အနေနဲ့ အတိုကောက်ရေးရင်လည်း ရလို့ အတိုကောက်ပဲ ဆက်ရေးကြပါမယ်။

နောက်ထပ်သတိပြုရမှာကတော့ JSX Element တွေကိုရေးတဲ့အခါ အများအားဖြင့် HTML Element တွေရေးတဲ့အတိုင်းပဲ ရေးရပေမယ့် ခြွင်းချက်တချို့ရှိနေပါတယ်။ အထက်မှာပြောခဲ့တဲ့ `className` ဟာ ခြွင်းချက်တစ်ခု အပါအဝင်ဖြစ်သလို၊ နောက်ထပ်သတိပြုရမယ့်ခြွင်းချက်ကတော့ အပိတ်မလိုတဲ့ Empty Tag တွေကို Self Close ပိတ်ပေးဖို့ လိုအပ်ခြင်းဖြစ်ပါတယ်။ ဥပမာ –

- `<input type="text" />`
- `<img src="…" alt="" />`
- `<br />`
- `<hr />`

နောက်ဆုံးမှာ `/>` နဲ့ ပိတ်ပေးရတာပါ။ တခြားခြွင်းချက်တွေ ကျန်သေးပေမယ့် ဒီလောက်သိထားရင် အဆင်ပြေနေပါပြီ။ နောက်ထပ် မှုကွဲလေးတစ်ခုလောက် ထပ်ကြည့်ပါဦးမယ်။

React

```
function Item({ name, price }) {
  return (
    <li>{name} <i>(${price})</i></li>
  );
}
```

Props ကို တစ်ခါထဲ Destructure လုပ်ပြီး လက်ခံထားတာပါ။ ကုဒ်ကို တိုတောင်းကျစ်လစ်သွားစေတဲ့ အတွက် ဒီရေးနည်းကိုပဲ ရှေ့ဆက် အသုံးပြုသွားမှာ ဖြစ်ပါတယ်။ နောက်ထပ် အရေးကြီးတာလေးတစ်ခု ထပ်ကြည့်ရပါမယ်။ `children` Props ဖြစ်ပါတယ်။ Component Composition ခေါ် Component တွေကို ပေါင်းစပ်အသုံးပြုတဲ့အခါ အသုံးဝင်ပါတယ်။ ဒီလိုပါ –

React

```
function List(props) {
  return (
    <ul>
      {props.children}
    </ul>
  )
}

function Item({ name, price }) {
  return (
    <li>{name} <i>(${price})</i></li>
  );
}
```

နမူနာအရ List Component ဟာ Children တွေကို ထည့်သုံးထားပါတယ်။ ဒါကြောင့် ပြန်လည်အသုံးပြုတဲ့အခါ အခုလိုသုံးလို့ရသွားမှာ ဖြစ်ပါတယ်။

React

```
<List>
  <Item name="Apple" price="0.99" />
  <Item name="Orange" price="0.89" />
</List>
```

`<List>` အဖွင့်အပိတ်ထဲမှာ ပေးလိုက်တဲ့ `<Item>` Component တွေက `<List>` ရဲ့ Children တွေဖြစ်သွားတာပါ။

Children ဆိုတာ Props ကလာမှန်း သိအောင် `props` နဲ့ တစ်ဆင့်ခံပြီး ရေးပြခဲ့တာပါ။ အခုလိုရေးရင်လည်း အတူတူပါပဲ။

React

```
function List({ children }) {
  return (
    <ul>
      {children}
    </ul>
  )
}
```

အသုံးဝင်ပြီး မှတ်သားသင့်တဲ့ အခြေခံသဘောသဘာဝတွေ ဖြစ်ပါတယ်။ Style တွေ ရေးပုံရေးနည်းလည်း ဆက်ကြည့်ကြပါမယ်။ React နဲ့ ပူးတွဲအသုံးပြုလို့ရတဲ့ Style Framework တွေ UI Framework တွေ အများကြီး ရှိကြပါတယ်။ နောက်မှကြည့်ကြပါမယ်။ ဒီနေရာမှာတော့ ရိုးရိုး CSS Style တွေ ရေးနည်းပဲ ထည့်ကြည့်ကြပါမယ်။

Styles

React Component တွေကို ဘယ်လိုနည်းလမ်းနဲ့ပဲ ရေးထားထား နောက်ဆုံးမှာ HTML Element တွေ အနေနဲ့ပဲ အလုပ်လုပ်သွားမှာဖြစ်လို့၊ CSS ကုဒ်တွေကို ရေးရိုးရေးစဉ်အတိုင်းပဲ ရေးလို့ရပါတယ်။ ဥပမာ –

CSS - style.css

```css
/* style.css */
.btn {
  display: inline-block;
  width: 150px;
  padding: 8px 0;
  border-radius: 5px;
  text-decoration: none;
  text-align: center;
  background: deeppink;
  color: white;
}
```

ဒီလိုရေးထားတဲ့ CSS ကုဒ်တွေကို React Component တွေမှာ အခုလို ချိတ်သုံးလို့ရပါတယ်။

React

```jsx
import "./style.css";

function Button() {
  return <a className="btn" href="#">Link Button</a>
}
```

`import` နဲ့ CSS ဖိုင်ကို ထည့်ပေးလိုက်ပြီး Element တွေမှာ `className` နဲ့ လိုအပ်သလို ထည့်သုံးလိုက်တာပါ။ နောက်တစ်နည်းအဖြစ် Inline Style အနေနဲ့လည်း ရေးလို့ရပါတယ်။ ဒီလိုပါ –

React

```
const styles = {
  btn: {
    display: "inline-block",
    width: 150,
    padding: "8px 0",
    borderRadius: 5,
    textDecoration: "none",
    textAlign: "center",
    background: "deeppink",
    color: "white",
  }
}

function Button() {
  return <a href="#" style={styles.btn}>
    Link Button
  </a>
}
```

ရေးချင်တဲ့ Style တွေကို Object တစ်ခုအနေနဲ့ ကြိုရေးထားပြီး Element ရဲ့ `style` Attribute မှာ ထည့်ပေးလိုက်တာပါ။ ရိုးရိုး CSS မှာ `border-radius` လို Kebab Case နဲ့ ရေးတဲ့ Property တွေကို JavaScript မှာ `borderRadius` လို့ Camel Case နဲ့ ရေးပေးရတာကို သတိပြုပါ။ ဒါအသစ်အဆန်းတော့ မဟုတ်ပါဘူး။ Style တွေကို JavaScript နဲ့ရေးရင် အမြဲအဲလိုပဲ ရေးရပါတယ်။

အသုံးဝင်တဲ့ ထူးခြားချက်တစ်ချက် ကျန်ပါသေးတယ်။ Pixel Unit တွေမှာ `px` နဲ့ Quote အဖွင့်အပိတ် ထည့်စရာမလိုခြင်း ဖြစ်ပါတယ်။ ဒါကြောင့် `width: w + 100` လို ပေါင်းနှုတ်မြှောက်စားအပါအဝင် Expression တွေ ထည့်ရေးလို့ရသွားတဲ့အတွက် ပိုအသုံးဝင်သွားမှာ ဖြစ်ပါတယ်။

Style Object သပ်သပ်ခွဲမရေးဘဲ တစ်ခါထဲ Inline ထည့်ရေးရင်လည်း ရပါတယ်။ ဒီလိုပါ –

React

```
function Button() {
  return <a href="#" style={{
    display: "inline-block",
    width: 150,
    padding: "8px 0",
    borderRadius: 5,
    textDecoration: "none",
    textAlign: "center",
    background: "deeppink",
    color: "white",
  }}>
    Link Button
  </a>
}
```

ခွဲရေးထားရင်တော့ လိုတဲ့နေရာတွေကနေ ပြန်ယူသုံးလို့ရမှာပါ။ သုံးဖို့မလိုဘူးဆိုရင်လည်း ခွဲရေးမနေဘဲ အခုလို Element ထဲမှာပဲ တစ်ခါထဲ ရေးပေးလိုက်လို့ရမှာ ဖြစ်ပါတယ်။

States

State ဆိုတာ Data တစ်မျိုးပါပဲ။ UI State Data လို့ပြောရင် ပြည့်စုံပါလိမ့်မယ်။ UI ရဲ့ လက်ရှိအခြေအနေကို မှတ်သားထားသော Data ဆိုပါတော့။ ဥပမာ –

React

```
let state = "green";

<div style={{
    width: 300,
    height: 200,
    backgroundColor: state
  }}
>
```

state ရဲ့ Value ဟာ green ဖြစ်ပြီး Component ရဲ့ backgroundColor Style က state နဲ့ချိတ်ထားလို့ green ဖြစ်နေမှာပါ။ အစိမ်းရောင် Component လေးတစ်ခုကို ရမှာပါ။ အခုနေ state ရဲ့ Value ကို red လို့ ပြောင်းပေးလိုက်ရင် အနီရောင် Component အဖြစ်ကို ပြောင်းလဲရရှိမှာပါ။ State Data ဆိုတာ ဒီသဘောပါပဲ။ UI ရဲ့ လက်ရှိအခြေအနေကို မှတ်သားပေးထားတဲ့ Data တစ်ခုပါ။

React ရဲ့ အဓိကအနှစ်သာရ တစ်ခုက State Data ပြောင်းရင် UI ကို ပြောင်းလဲဖော်ပြပေးနိုင်ခြင်းပါပဲ။ State Data ပြောင်းတဲ့အခါ UI ဖော်ပြပုံ လိုက်ပြောင်းအောင် Manual Refresh တွေ Reload တွေ ဘာမှ လုပ်နေစရာမလိုဘဲ Component ကို အလိုအလျောက် ပြောင်းလဲဖော်ပြပေးသွားမှာ ဖြစ်ပါတယ်။ ရေးပုံရေးနည်းက ဒီလိုပါ။

React

```
import { useState } from "react";

export default function Box() {
  const [state, setState] = useState("green");

  return (
    <div
      style={{
        width: 300,
        height: 200,
        display: "flex",
        alignItems: "center",
        justifyContent: "center",
        gap: 10,
        backgroundColor: state,
      }}>
      <button onClick={() => setState("red")}>Red</button>
      <button onClick={() => setState("green")}>Green</button>
    </div>
  );
}
```

အရင်ဆုံး `useState` ကို `react` ကနေ Import လုပ်ယူထားပါတယ်။ Hook Function လို့ခေါ်ပါတယ်။ React မှာ အလားတူ Hook Function တွေအများကြီး ရှိကြပါတယ်။ <u>Hook ဆိုတာ လက်ရှိ Function ရဲ့ လုပ်ဆောင်ချက်တွေကို လာချိတ် ဖြည့်စွက်ပေးတဲ့ လုပ်ဆောင်ချက်တစ်မျိုးပါ</u>။

လောလောဆယ် အဲ့ဒါတွေ နားရှုပ်တယ်ထင်ရင် Component State တွေကြေညာတဲ့အခါ `useState()` ကို အသုံးပြုကြေညာရတယ်လို့ အလွယ်မှတ်ထားနိုင်ပါတယ်။ နမူနာအရ `Box()` Function ထဲမှာ `state` နဲ့ `setState` တို့ကို `useState()` အသုံးပြုပြီး ကြေညာထားပါတယ်။ `state` က လက်ရှိ Component က State Data ဖြစ်သွားပြီး `setState` က အဲ့ဒီ State Data တန်ဖိုးကို သတ်မှတ်ပေးနိုင်တဲ့ Function ဖြစ်ပါတယ်။ <u>State Data ကို တိုက်ရိုက် Value တွေ Assign လုပ်ပြီး ပြောင်းလဲသတ်မှတ်လို့ မရပါဘူး။ `setState()` ကနေတစ်ဆင့် ပြောင်းလဲသတ်မှတ်ပေးရမှာ ဖြစ်ပါတယ်</u>။

`useState()` ကြေညာစဉ်မှာ Default Value ပေးလို့ရပါတယ်။ နမူနာမှာ `"green"` ဆိုတဲ့ Value ကို Default Value အနေနဲ့ ထည့်ပေးထားပါတယ်။ ပြီးတဲ့အခါ Return ပြန်ပေးတဲ့ `<div>` Element ရဲ့ `backgroundColor` Style ကို `state` Value နဲ့ ချိတ်ပြီး သတ်မှတ်ပေးထားလို့ `state` Value ပြောင်းရင် `backgroundColor` လိုက်ပြောင်းသွားမှာပါ။

စမ်းကြည့်လို့ရအောင် `<button>` နှစ်ခုလည်း ထည့်ပေးထားပါတယ်။ `onClick` တွေ ကိုယ်စီပါကြပါတယ်။ `onClick` တွေမှာ Arrow Function တစ်ခုစီ Assign လုပ်ပေးထားပါတယ်။ Red Button ကိုနှိပ်ရင် `setState("red")` လို့ပြောထားတဲ့အတွက် `state` ရဲ့ Value ကို `"red"` လို့သတ်မှတ်လိုက်တာပါ။ ဒါကြောင့် ဒီခလုပ်ကိုနှိပ်လိုက်ရင် Component ရဲ့ `backgroundColor` ကိုအနီရောင်ပြောင်းပြီး ပြပေးသွားမှာ ဖြစ်ပါတယ်။ Green Button ကို နှိပ်ရင်တော့ `setState("green")` လို့ပြောထားတဲ့အတွက် Component ကလည်း အစိမ်းရောင် ပြန်ဖြစ်သွားမှာပဲ ဖြစ်ပါတယ်။

သတိပြုစရာ နှစ်ချက်ရှိပါတယ်။

ပထမတစ်ချက်က `onClick` လို Even Attribute တွေကို အကြီးအသေး ရောရေးရတဲ့ `camelCase` နဲ့ ရေးပေးရပါတယ်။ ရိုးရိုး HTML မှာဆိုရင် အကြီး၊ အသေး၊ အရော ကြိုက်သလို ရေးလို့ရပါတယ်။ JSX ဟာ အများအားဖြင့် HTML ရေးသလိုပဲ ရေးရပါပေမယ့် ဒီလိုခြွင်းချက်တချို့ ရှိနေတာပါ။ နောက်တစ်ချက်ကတော့ `onClick` အတွက် Function တစ်ခုကို Assign လုပ်ပေးရမှာပါ။ ဥပမာ –

React

```
function hello() {
  // do someting
}

<button onClick={hello()}>Button</button>     //အမှား
<button onClick={hello}>Button</button>       //အမှန်
```

နမူနာမှာ `hello` Function ကို `onClick` အတွက် သတ်မှတ်ပေးတဲ့အခါ နောက်က ဝိုက်ကွင်း အဖွင့်အပိတ်ကို ထည့်လို့မရပါဘူး။ ဝိုက်ကွင်းအဖွင့်အပိတ် ထည့်လိုက်ရင် Assign လုပ်ပေးတာ မဟုတ်ဘဲ Function ကို Run လိုက်တာ ဖြစ်နေလို့ပါ။ ဝိုက်ကွင်းအဖွင့်အပိတ် ထည့်ရေးလို့ရချင်ရင် Higher Order Function ကို သုံးရပါတယ်။ ရိုးရိုး Function ကို သုံးလို့မရပါဘူး။ ဒီနေရာမှာတော့ Higher Order Function တွေအကြောင်း ထည့်မပြောနိုင်သေးပါဘူး။ ဒါကြောင့် အခြေခံအားဖြင့် <u>`onClick` လို Event Attribute တွေအတွက် Function တွေ Assign လုပ်ပေးတဲ့အခါ ဝိုက်ကွင်းအဖွင့်အပိတ် ထည့်မရေးရဘူး</u> လို့ပဲမှတ်ထားပေးပါ။ ဒီလိုရေးရင်တော့ ရနိုင်ပါတယ် –

React
```
<button onClick={() => hello()}>Button</button>      //အမှန်
```

ဒါက `onClick` အတွက် Array Function တစ်ခုကို Assign လုပ်ပေးလိုက်တာပါ။ `hello` Function ကို တိုက်ရိုက် Assign လုပ်တာ မဟုတ်ပါဘူး။ ဒါကို မျက်စိလည်မသွားအောင် ဂရုစိုက်ကြည့်ပါ။ Assign လုပ်ပေးတာ Arrow Function ဖြစ်ပြီး၊ ခေါ်ချင်တဲ့ Function ကို Arrow Function ထဲမှာ ခေါ်သုံးထားတာပါ။

ဒါကြောင့် စောစောကနမူနာတွေမှာ onClick တွေအတွက် Arrow Function တွေကို အသုံးပြု ရေးသား ပေးခဲ့ခြင်း ဖြစ်ပါတယ်။

Exercise

လက်ရှိလေ့လာခဲ့တဲ့အကြောင်းအရာတွေကို ပေါင်းစပ်ပြီး လက်တွေ့ချရေးကြည့်လို့ရမယ့် နမူနာလေးတစ်ခုလောက် လုပ်ကြည့်ကြပါမယ်။ React ပရောဂျက်သစ်တစ်ခု တည်ဆောက်လိုက်ပါ။ <u>အခန်းစဉ်အလိုက် လိုအပ်တာတွေ လေ့လာရင်း ပြည့်စုံအောင် ဖြည့်စွက်သွားမယ့် အဓိကပရောဂျက်ဖြစ်ပါတယ်</u>။ ဒါကြောင့် စကတည်းက သေသေချာချာ ဂရုစိုက်ပြီး မှီအောင်လိုက်လုပ်ကြည့်ပါ။

```
npm create vite yaycha

[choose React & JavaScript]
```

```
cd yaycha
npm install
npm run dev

VITE v5.2.13  ready in 233 ms

  ➜  Local:   http://localhost:5173/
  ➜  Network: use --host to expose
  ➜  press h + enter to show help
```

ပရောဂျက်အမည်ကို **yaycha** လို့ပေးထားပါ။ ဒီနာမည်နဲ့ Social Media ပရောဂျက်လေးတစ်ခုလုပ်သွားကြမှာပါ။ စမ်းကြည့်လို့ရအောင် `npm run dev` နဲ့ Run ထားပါတယ်။ Browser မှာ သူ့ Default Setting အတိုင်း http://localhost:5173 ကို ဖွင့်ကြည့်လိုက်ရင် Vite ရဲ့ Default Page ကို တွေ့မြင်ရပါလိမ့်မယ်။

ပရောဂျက်ဖိုဒါကို Code Editor နဲ့လည်း ဖွင့်လိုက်ပါ။ ပါဝင်တဲ့ ဖိုင်တွေ ဖိုဒါတွေရဲ့ ဖွဲ့စည်းပုံက အခုလို ဖြစ်နိုင်ပါတယ်။

```
yaycha
├── README.md
├── index.html
├── node_modules/
├── package-lock.json
├── package.json
├── public/
├── src/
│   ├── App.css
│   ├── App.jsx
│   ├── assets
│   ├── index.css
│   └── main.jsx
└── vite.config.js
```

`vite.config.js` ဖိုင်မှာ Port နဲ့ Server Setting ပြင်လို့ရတာ ပြီးခဲ့တဲ့အခန်းမှာ ကြည့်ခဲ့ပြီးဖြစ်ပါတယ်။ ထပ်မံသတိပြုစရာအချက်တစ်ခု အနေနဲ့ React/JSX Component တွေရေးမယ့် ကုဒ်ဖိုင်ရဲ့

Extension ကို `.jsx` လို့ ပေးရပါတယ်။ `.js` မဟုတ်ပါဘူး။ Vite ကသတ်မှတ်ထားတဲ့ သတ်မှတ်ချက်ဖြစ်ပါတယ်။ React က အဲ့ဒီလိုပေးရမယ်လို့ သတ်မှတ်ထားတာမျိုး မဟုတ်ပါဘူး။

အဓိကပင်မဖိုင်က `src/main.jsx` ဖြစ်ပါတယ်။ ဖွင့်ပြီး ~~`index.css`~~ ဖိုင်ကို Import လုပ်ထားတဲ့လိုင်းကို ဖျက်လိုက်ပါ။ သူနမူနာကြိုရေးထားပေးတဲ့ Style တွေကို မလိုချင်လို့ပါ။

React - src/main.jsx

```
import React from "react";
import ReactDOM from "react-dom/client";
import App from "./App.jsx";
import "./index.css";

ReactDOM.createRoot(document.getElementById("root")).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

လေ့လာကြည့်လိုက်ရင် `App.jsx` ကို Import လုပ်ပြီး အသုံးပြုထားတာကို သတိပြုပါ။

ဒီနေရာမှာ ဖြည့်စွက် မှတ်သားသင့်တာက၊ React Component တွေရေးတဲ့အခါ စာလုံးအကြီးနဲ့စပြီး ရေးရတဲ့ `CapitalCase` ကို သုံးသင့်ပြီး၊ Component ဖိုင်ရဲ့ Name ကို လည်း `CapitalCase` နဲ့ပဲ ပေးသင့်တယ်ဆိုတဲ့အချက်ပဲ ဖြစ်ပါတယ်။ App Component ရှိတဲ့ဖိုင်ဖြစ်လို့ ဖိုင်ရဲ့အမည်ကို Component အမည်နဲ့တူအောင် `App.jsx` လို့ပေးထားတာပါ။ ဒါလည်း လိုက်နာသင့်တဲ့ Convention တစ်ခုပါပဲ။

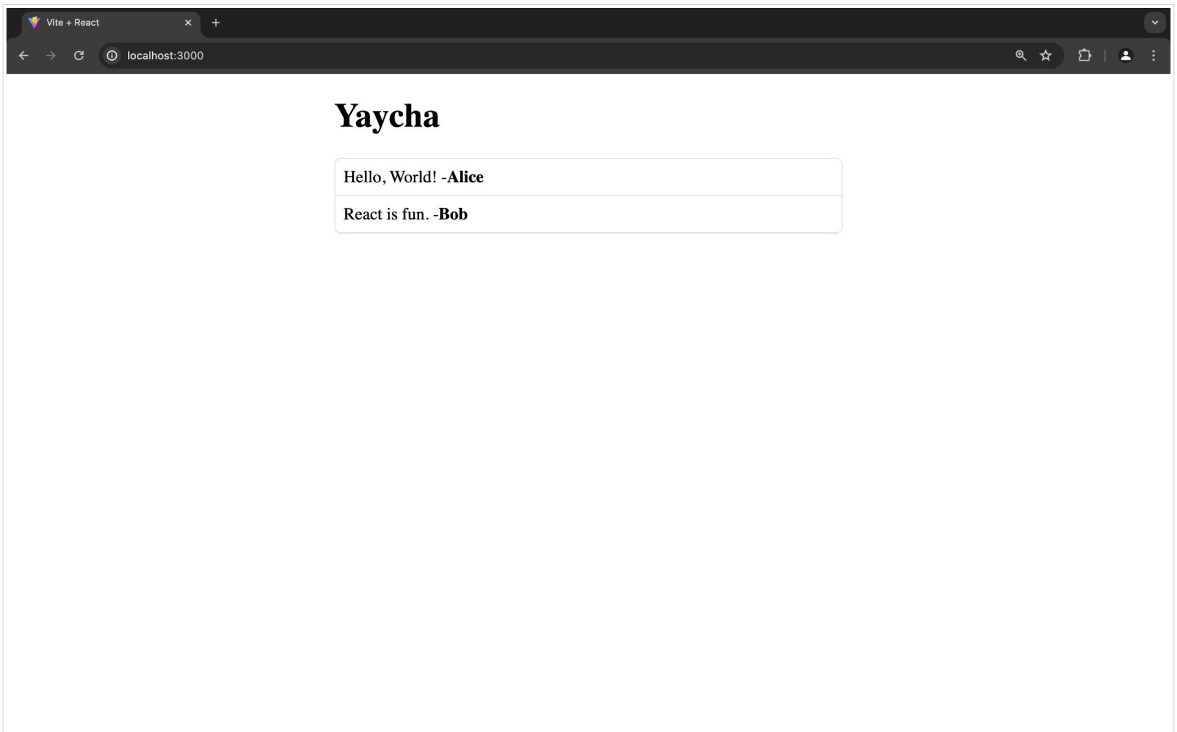
`src/App.jsx` ကို ဖွင့်ပြီး ~~အထဲကနမူနာကုဒ်တွေ~~ အကုန်လုံးကို ဖျက်လိုက်ပါ။ ပြီးရင် ဒီကုဒ်တွေရေးပေးလိုက်ပါ။

React - src/App.jsx

```
function Item({ content, name }) {
  return (
    <li style={{ padding: 10, borderBottom: "1px solid #ddd" }}>
      {content} -<b>{name}</b>
    </li>
  );
}

function List({ children }) {
  return <ul
    style={{
      listStyle: "none",
      padding: 0,
      margin: 0,
      border: "1px solid #ddd",
      borderRadius: 8,
      overflow: "hidden",
    }}>
    {children}
  </ul>;
}

export default function App() {
  return (
    <div style={{ maxWidth: 600, margin: "20px auto" }}>
      <h1>Yaycha</h1>
      <List>
        <Item content="Hello, World!" name="Alice" />
        <Item content="React is fun." name="Bob" />
      </List>
    </div>
  );
}
```

Component (၃) ခု ရေးထားပါတယ်။ `Item`, `List` နဲ့ `App` တို့ပါ။ `App` ကိုတော့ ဒီဖိုင်ရဲ့ အဓိက Component အနေနဲ့ `default export` ပေးထားပါတယ်။ ဒါမှ သူ့ကို `src/main.jsx` ကနေ ယူသုံးလို့ရမှာပါ။

ရှေ့မှာရှင်းပြခဲ့တဲ့ ကုဒ်တွေကိုပဲ လက်တွေ့စမ်းလို့ရအောင် ပေါင်းစပ်ပေးလိုက်တာပါ။ ဒါကြောင့် နောက်တစ်ကြိမ်တော့ ထပ်ရှင်းမနေတော့ပါဘူး။ ရေးထားတဲ့ကုဒ်တွေကိုပဲ ဖတ်ကြည့်လိုက်ပါ။ Browser မှာ ကြည့်လိုက်ရင် ရလဒ်က အခုလို ဖြစ်ပါလိမ့်မယ်။



ဒါ State တွေ Data တွေ မပါသေးဘဲ၊ ဒီအတိုင်းချရေးလိုက်တဲ့ UI တစ်ခုပါ။ `<List>` တစ်ခုထဲမှာ `<Item>` နှစ်ခုထည့်ရေးထားလို့ ရေးထားတဲ့အတိုင်းပဲ ရလဒ်ကို ပြန်မြင်ရတာပါ။

Separated Components

Component (၃) ခုကို ဖိုင်တစ်ခုထဲမှာ စုရေးလိုက်တာပါ။ နောင်ပြင်ချင်တာရှိရင် သီးခြားစီ ပြင်လို့ရအောင် ဖိုင် (၃) ခုနဲ့ ခွဲပြီး ပြောင်းရေးကြပါတယ်။ ဒါကြောင့် `src/` ထဲမှာ `Item.jsx` ဆိုတဲ့အမည်နဲ့ ဖိုင်တစ်ခု

ဆောက်ပြီး ဒီကုဒ်ကို ရေးပေးလိုက်ပါ။

React - src/Item.jsx

```
export default function Item({ content, name }) {
  return (
    <li style={{ padding: 10, borderBottom: "1px solid #ddd" }}>
      {content} -<b>{name}</b>
    </li>
  );
}
```

စောစောက ရေးရေးထားတဲ့ Item Component ကုဒ်နဲ့ အတူတူပါပဲ။ တခြားလိုအပ်တဲ့နေရာကနေ ပြန်ချိတ်သုံးလို့ရအောင် export default လုပ်ပေးထားတာပဲ ထူးသွားပါတယ်။ ပြီးတဲ့ အခါ src/ ထဲမှာပဲ List.jsx ဆိုတဲ့ အမည်နဲ့ နောက်ထပ်ဖိုင်တစ်ခု ထပ်ဆောက်ပြီး ဒီကုဒ်ကို ရေးပေးလိုက်ပါ။

React - src/List.jsx

```
export default function List({ children }) {
  return (
    <ul
      style={{
        listStyle: "none",
        padding: 0,
        margin: 0,
        border: "1px solid #ddd",
        borderRadius: 8,
        overflow: "hidden",
      }}>
      {children}
    </ul>
  );
}
```

စောစောက `List` ကိုပဲ Default Export အနေနဲ့ ထုတ်ပေးပြီးရေးလိုက်တာပါ။ ဒီလိုခွဲရေးလိုက်ပြီဖြစ်လို့ `src/App.jsx` ရဲ့ ကုဒ်က အခုလိုဖြစ်သွားရပါမယ်။

React - src/App.jsx

```
import List from "./List.jsx";
import Item from "./Item.jsx";

export default function App() {
  return (
    <div style={{ maxWidth: 600, margin: "20px auto" }}>
      <h1>Yaycha</h1>
      <List>
        <Item content="Hello, World!" name="Alice" />
        <Item content="React is fun." name="Bob"/>
      </List>
    </div>
  );
}
```

`Item` တွေ `List` တွေကို ရောမရေးတော့ဘဲ Import လုပ်ယူပြီး သုံးလိုက်တာပါ။ Import လုပ်တဲ့အခါ ဖိုင် Name တွေမှာ `.jsx` Extension ကို ထည့်မရေးလည်း ရပါတယ်။ အခုဆိုရင် Component (၃) ခုကို ဖိုင် (၃) ခုနဲ့ခွဲရေးလိုက်ပါပြီ။ ဒါကြောင့် သီးခြားစီ ပြင်လို့ရသွားပါပြီ။ ရလဒ်ကတော့ အပြောင်းအလဲရှိမှာ မဟုတ်သေးပါဘူး။

Component States

ဆက်လက်ပြီး ဒီနမူနာမှာ State Data တချို့ ထည့်ကြည့်ကြပါမယ်။ `App.jsx` ကို အခုလို ပြင်လိုက်ပါ။

React - src/App.jsx

```
import { useState } from "react";

import List from "./List";
import Item from "./Item";

export default function App() {
  const [data, setData] = useState([
    { id: 1, content: "Hello, World!", name: "Alice" },
    { id: 2, content: "React is fun.", name: "Bob" },
    { id: 3, content: "Yay, interesting.", name: "Chris" },
  ]);

  return (
    <div style={{ maxWidth: 600, margin: "20px auto" }}>
      <h1>Yaycha</h1>
      <List>
        {data.map(item => {
          return <Item content={item.content} name={item.name} />
        })}
      </List>
    </div>
  );
}
```

အရင်ဆုံး `useState` ကို Import လုပ်ပါတယ်။ ပြီးတဲ့အခါ App Component Function ထဲမှာ `useState` ကိုသုံးပြီး `data` နဲ့ `setData` တို့ကိုကြေညာပါတယ်။ Default Value အနေနဲ့ Object Array တစ်ခုကိုပေးထားလို့ လက်ရှိ data ရဲ့ Value ဟာ အဲ့ဒီ Object Array ဖြစ်နေမှာပါ။ ဆက်လက်ပြီး မူလက ဒီအတိုင်းနမူနာရေးထည့်ထားတဲ့ `<Item>` Component တွေကို ဖယ်ထုတ်လိုက်ပြီး၊ သူ့အစား `data` ကို `map()` နဲ့ Loop လုပ်ပြီး `<Item>` တွေကို ပြလိုက်ပါတယ်။ `map()` နဲ့ `<Item>` တွေကို ပြ

တဲ့ကုဒ်ကိုရေးနည်းမူကွဲတချို့ ရှိပါတယ်။ ဒီလိုပါ။

React

```
<List>
  { data.map(item => <Item content={item.content} name={item.name} />) }
</List>
```

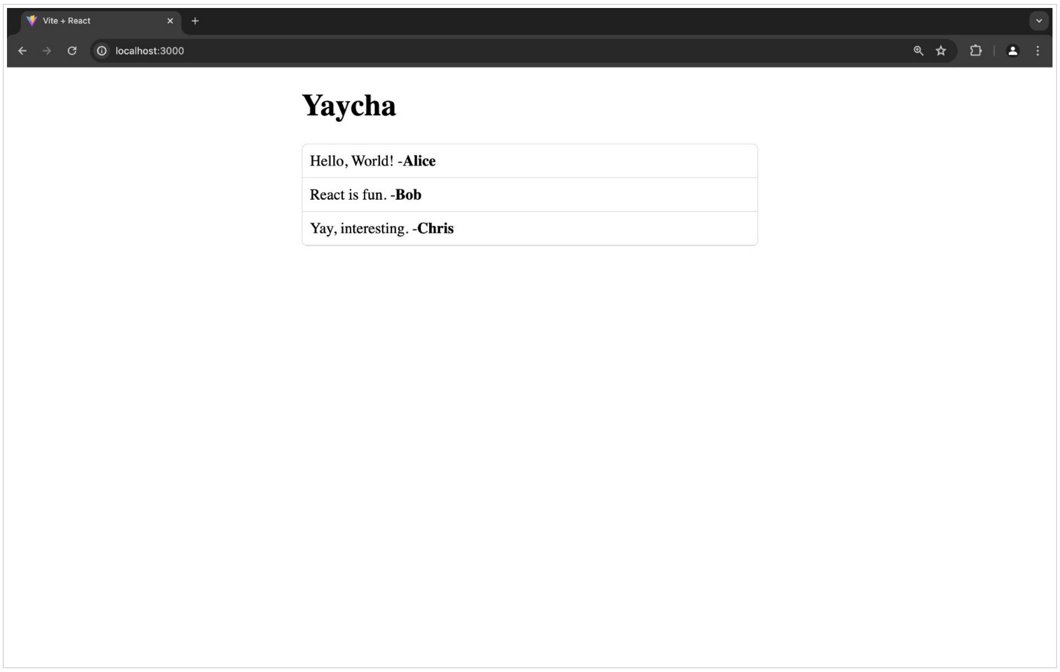
React

```
<List>
  {data.map(item => (
    <Item content={item.content} name={item.name} />
  ))}
</List>
```

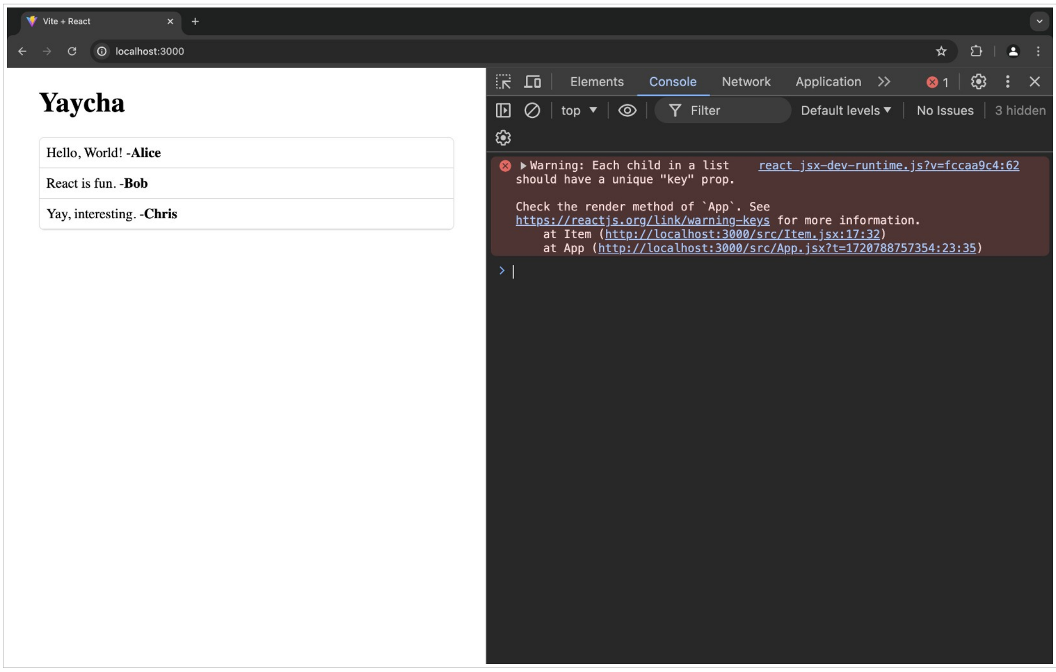
Arrow Function တွေမှာ Statement ကတစ်ကြောင်းထဲဆိုရင် တွန့်ကွင်းတွေ Return တွေ ထည့်စရာမလိုပါဘူး။ ဒါကြောင့် မူကွဲ (၁) မှာ တွန့်ကွင်းတွေ Return တွေ ဖြုတ်လိုက်တာပါ။ ဒါပေမယ့် တစ်ကြောင်းထဲ အမြဲတမ်းဖြစ်နိုင်မှာ မဟုတ်ပါဘူး။ နှစ်ကြောင်းသုံးကြောင်း ခွဲရေးဖို့ လိုတတ်ပါတယ်။ ဒါကြောင့် မူကွဲ (၂) မှာ ဝိုက်ကွင်းအဖွင့်အပိတ်နဲ့ ရေးထားပါတယ်။ <u>တွန့်ကွင်းမဟုတ်ပါဘူး။ ဝိုက်ကွင်းအဖွင့်အပိတ်ထဲမှာ ရေးလိုက်တဲ့အတွက်၊ နှစ်ကြောင်းသုံးကြောင်း ရေးထားရင်တောင် Statement တစ်ခုထဲလို ဖြစ်သွားတဲ့အတွက် Return ထည့်စရာမလိုတော့ပါဘူး။</u>

မူလနမူနာနဲ့ ပြန်တိုက်ကြည့်ပါ။ မူလနမူနာမှာတော့ ရေးရိုးရေးစဉ်အတိုင်း တွန့်ကွင်းတွေ Return တွေနဲ့ အပြည့်အစုံရေးပေးထားပါတယ်။

State `data` ကို သုံးပြီး `Item` တွေကို ပြလိုက်တာဖြစ်လို့ ရလဒ်ကိုကြည့်ရင် အခုလိုဖြစ်ပါလိမ့်မယ်။



နောက်ထပ်သတိပြုစရာတစ်ခု ရှိလာပါတယ်။ အခုနေ Browser Console ကို ဖွင့်ကြည့်ရင် Warning တစ်ခု ရှိနေပါလိမ့်မယ်။ ဒီလိုပါ –



Each child in a list should have a unique "key" prop လို့ပြောနေပါတယ်။ ဆိုလိုတာက Loop လုပ်ပြီးဖော်ပြတဲ့ List Item တွေမှာ key လို့ခေါ်တဲ့ prop တစ်ခုမဖြစ်မနေ ပါသင့်တယ်လို့ ပြောနေတာပါ။ ဒါကြောင့် ဒီ Warning ပျောက်သွားအောင် အခုလိုပြင်ပေးဖို့လိုပါတယ်။

React - src/List.jsx

```
<List>
  {data.map(item => {
    return (
      <Item
        key={item.id}
        content={item.content}
        name={item.name}
      />
    )
  })}
</List>
```

`<Item>` မှာ key ပါသွားပြီး Unique Value ဖြစ်ဖို့လိုတဲ့အတွက် Default State Data မှာထည့်ပေးခဲ့တဲ့ id ကို အသုံးပြုထားပါတယ်။ ပြင်ပြီး ပြန်စမ်းကြည့်လိုက်ရင် စောစောက key ထည့်ဖို့ပြောနေတဲ့ Warning ပျောက်သွားပါလိမ့်မယ်။

Virtual DOM

React ရဲ့ ထူးခြားချက်တွေထဲမှာ Virtual DOM ဟာလည်း အဓိကကျတဲ့အချက်ဖြစ်ပါတယ်။ သူ့ရဲ့အလုပ်လုပ်ပုံ လိုရင်းအကျဉ်းချုပ်က၊ State အပြောင်းအလဲရှိလို့ Component တွေကို Browser မှာ Render ပြန်လုပ်ရတဲ့အခါ အားလုံးကို Render ပြန်မလုပ်ဘဲ လိုတဲ့အပိုင်းကိုပဲ ရွေးပြီး Render ပြန်လုပ်ပေးတဲ့အတွက် အလုပ်လုပ်ပုံမြန်ဆန်ခြင်းဖြစ်ပါတယ်။

ဥပမာ –

```
state = ["A", "B", "C"]
⬇️
Render
⬇️
<ul>
  <li>A</li>
  <li>B</li>
  <li>C</li>
</ul>
```

UI ကို State Data ရှိတဲ့အတိုင်း ပြထားပေးတာပါ။ ဒါကြောင့် State Data ပြောင်းသွားချိန်မှာ UI ရဲ့ဖော်ပြပုံကလည်း အခုလို လိုက်ပြီး ပြောင်းရမှာပါ။

```
setState(["A", "D", "C"])
⬇️
/* Render Start */
<ul>
  <li>A</li>
  <li>D</li>
  <li>C</li>
</ul>
/* Render End */
```

ဒီနေရာမှာ လုပ်ရိုးလုပ်စဉ်အတိုင်းဆိုရင် UI ကြီးတစ်ခုလုံးကို အစအဆုံး Render ပြန်လုပ်ရမှာပါ။ ဒါကြောင့် React က အလားတူအခြေအနေမျိုးမှာ အခုလို အလုပ်လုပ်ပါတယ်။

```
state = ["A", "B", "C"]
⬇️
VirtualDOM = { "0": "A", "1": "B", "2": "C" }
⬇️
/* Render Start */
<ul>
  <li key="0">A</li>
  <li key="1">B</li>
  <li key="2">C</li>
</ul>
/* Render End */
```

UI ကို Browser မှာ ချက်ခြင်းမပြဘဲ Virtual DOM Object တစ်ခုအရင် တည်ဆောက်လိုက်တာပါ။ State Data ကိုပြောင်းလိုက်တဲ့အခါ၊ အခုလို အလုပ်လုပ်သွားပါလိမ့်မယ်။

```
state = ["A", "D", "C"]
⬇️
OldVirtualDOM = { "0": "A", "1": "B", "2": "C" }
NewVirtualDOM = { "0": "A", "1": "D", "2": "C" }
⬇️
Compare OldVirtualDOM to NewVirtualDOM
⬇️
<ul>
  <li key="0">A</li>

  /* Render Start */
  <li key="1">D</li>
  /* Render End */

  <li key="2">C</li>
</ul>
```

UI တစ်ခုလုံးကို Render ပြန်မလုပ်တော့ဘဲ လိုတဲ့အပိုင်းကိုပဲ ရွေးလုပ်သွားမှာဖြစ်ပါတယ်။ အတိအကျ မဟုတ်ပေမယ့် ဒါဟာ Virtual DOM ရဲ့ အခြေခံအလုပ်လုပ်ပုံပါပဲ။

တစ်ခုချင်းအသုံးပြုတဲ့ Component မှာ ပြဿနာမရှိပါဘူး။ Loop လုပ်ပြီး ထပ်ခါထပ်ခါ ပြန်ပြတဲ့ တူညီတဲ့ Component တွေမှာတော့ `key` ပါမှသာ React က ဒီလိုအလုပ်လုပ်ပေးနိုင်မှာဖြစ်ပါတယ်။ ဒါကြောင့် စောစောကနမူနာမှာ `key` မပါတဲ့အခါ ထည့်ပေးဖို့ Warning လာပေးခြင်းပဲဖြစ်ပါတယ်။

State Action

ဆက်လက်ပြီး လုပ်လက်စနမူနာမှာ Delete Feature လေးထည့်စမ်းကြည့်ပါမယ်။ `Item.jsx` မှာ ရေးထားတဲ့ `Item` Component ကို အခုလိုပြင်ပေးပါ။

React - src/Item.jsx

```
export default function Item({ item, remove }) {
  return (
    <li
      style={{
        padding: 10,
        borderBottom: "1px solid #ddd",
        display: "flex",
        flexDirection: "row",
        justifyContent: "space-between",
      }}>
      <span>
        {item.content} -<b>{item.name}</b>
      </span>
      <button onClick={() => remove(item.id)}>Delete</button>
    </li>
  );
}
```

အပြောင်းအလဲ နည်းနည်းများပါတယ်။ ပထမဆုံး `props` ကနေ လက်ခံယူတာ `name` တွေ `price` တွေ မဟုတ်တော့ပါဘူး။ `item` နဲ့ `remove` ကို လက်ခံယူထားပါတယ်။ ဒါကြောင့် `name` တွေ `price` တွေကို တိုက်ရိုက်မပြတော့ဘဲ `item.name` နဲ့ `item.price` အဖြစ် ပြထားပါတယ်။ အဓိကဖြည့်စွက်ချက် အနေနဲ့ `<button>` တစ်ခုပါသွားပြီး `onClick` မှာ Arrow Function တစ်ခုကို Assign လုပ်ပေးထားပါတယ်။ Arrow Function ထဲမှာ `item.id` ကိုပေးပြီး `remove()` ကို Run ထားပါတယ်။ ဒါကြောင့် `App.jsx` ကိုလည်း အခုလို ပြင်ပေးရမှာ ဖြစ်ပါတယ်။

React - src/App.jsx

```
import { useState } from "react";

import List from "./List";
import Item from "./Item";

export default function App() {
  const [data, setData] = useState([
    { id: 1, content: "Hello, World!", name: "Alice" },
    { id: 2, content: "React is fun.", name: "Bob" },
    { id: 3, content: "Yay, interesting.", name: "Chris" },
  ]);

  const remove = id => {
    setData(data.filter(item => item.id !== id));
  };

  return (
    <div style={{ maxWidth: 600, margin: "20px auto" }}>
      <h1>Yaycha</h1>
      <List>
        {data.map(item => {
          return (
            <Item
              key={item.id}
              item={item}
```

```
                remove={remove}
              />
            );
          })}
        </List>
      </div>
    );
}
```

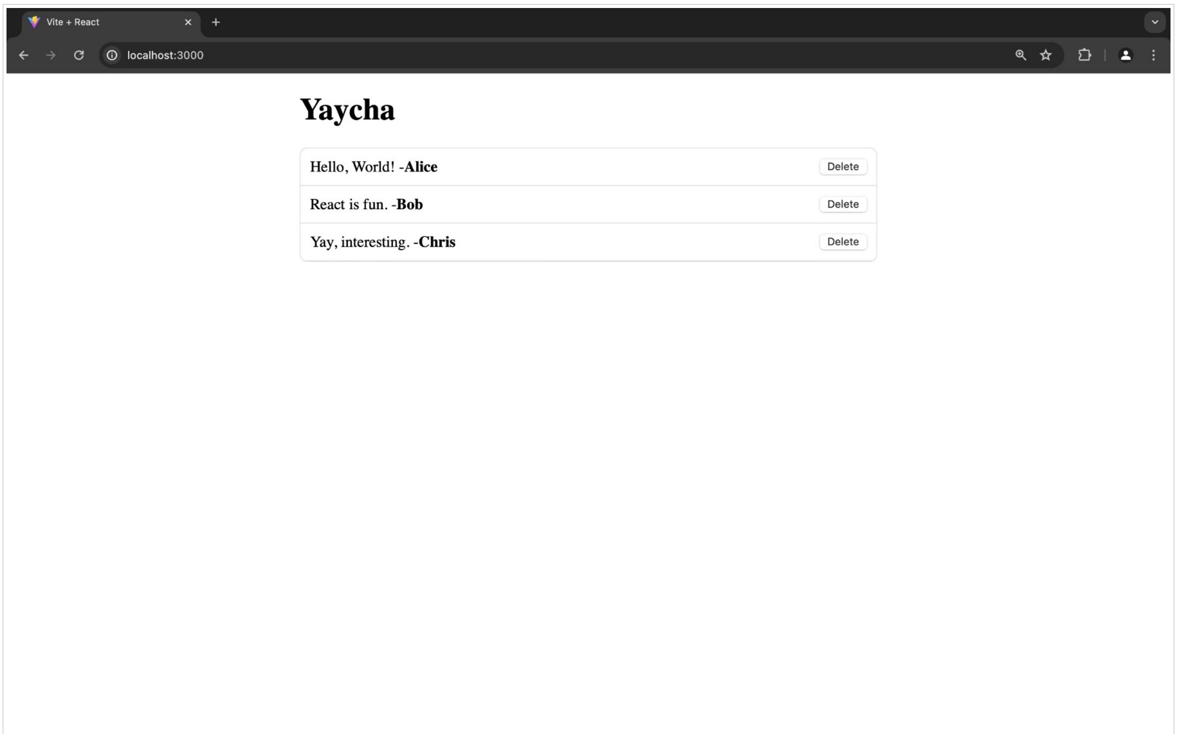
ကုဒ်အစအဆုံး အကုန်ပြန်ပြထားပါတယ်။ အဓိကထူးခြားချက်အနေနဲ့ App Component ထဲမှာ `remove()` ပါသွားပါတယ်။ `remove()` က `id` ကို လက်ခံပြီး `data` ကို `filter()` လုပ်ထားပါတယ်။ ပေးလာတဲ့ `id` နဲ့ `item.id` မတူတာတွေပဲ ပြန်ပေးတာဖြစ်လို့၊ တနည်းအားဖြင့် `id` တူတာကို ချန်ထားလိုက်တဲ့သဘော၊ ပယ်ဖျက်လိုက်တဲ့သဘော ဖြစ်သွားပါတယ်။

`map()` တွေ `filter()` အကြောင်းကို သိပ်မရှင်းရင် **JavaScript လိုတိုရှင်း** စာအုပ်မှာ ပြန်လေ့လာနိုင်ပါတယ်။ ဒီနေရာမှာတော့ အကျယ်ပြန်မရှင်းတော့ပါဘူး။

https://eimaung.com/jsbook

ဆက်လက်ပြီး `<Item>` ကိုခေါ်တဲ့အခါ `item` နဲ့ `remove` ကို ထည့်ပေးထားပါတယ်။ ဒါအရေးကြီးပါတယ်။ `<button>` ကိုနှိပ်လိုက်ရင် ပျက်ချင်တာက State Data ပါ။ State Data က App မှာ ရှိနေပြီး `<button>` က `Item` မှာ ရှိနေပါတယ်။ နေရာမတူကြပါဘူး။ `Item` က App ရဲ့ State Data ကို ဖျက်လို့မရပါဘူး။ သူ့ Data မဟုတ်လို့ပါ။ ဖျက်တဲ့အလုပ်ကို App မှာပဲ လုပ်ရပါတယ်။ ဒါကြောင့် App က ဖျက်တဲ့အလုပ်ကို လုပ်ပေးနိုင်တဲ့ `remove` ကို `props` အနေနဲ့ `Item` ကို ပေးထားတာဖြစ်ပါတယ်။ App ကပေးလာတဲ့ `remove` ကိုသုံးပြီး `Item` က App ရဲ့ State Data ကို ဖျက်လို့ရသွားမှာ ဖြစ်ပါတယ်။

စမ်းကြည့်လိုက်ရင် Delete လုပ်ဆောင်ချက်ပါဝင်သွားတဲ့ အခုလိုရလဒ်ကို ရရှိပါလိမ့်မယ်။



ထုံးစံအတိုင်း State Data ပြောင်းသွားတဲ့အခါ UI က အလိုအလျှောက် လိုက်လံပြောင်းလဲ အလုပ်လုပ်သွားတဲ့သဘောကို တွေ့မြင်ရမှာပဲ ဖြစ်ပါတယ်။

အခန်း (၄) - React - Forms

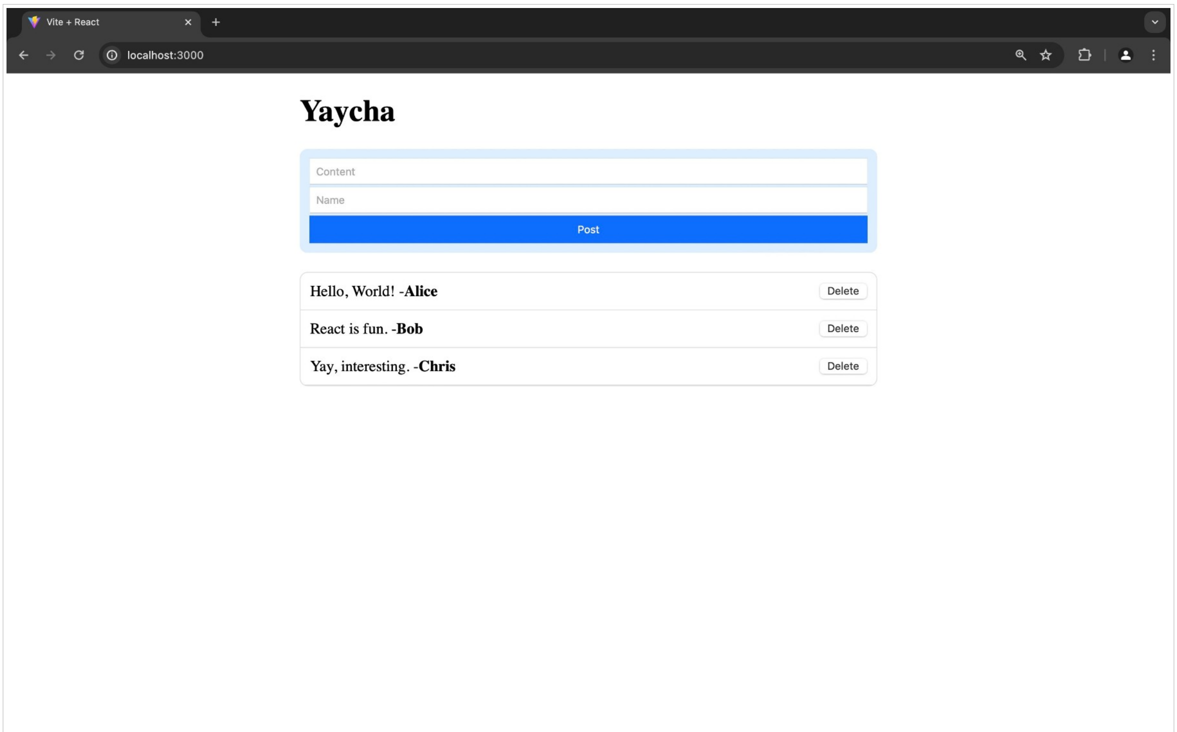
ဆက်လက်ပြီး Form Input တွေနဲ့ အလုပ်လုပ်ပုံအကြောင်း ဆက်လေ့လာကြပါမယ်။ တစ်လက်စထဲ Conditional Rendering လို အခြေခံသဘောလေးတွေပါ ထည့်ကြည့်ကြပါမယ်။ ပထမဆုံးအနေနဲ့ `src` ဖိုဒါထဲမှာ `Form.jsx` ဖိုင်နဲ့ ဒီကုဒ်တွေ ရေးပေးပါ။

React - src/Form.jsx

```
export default function Form() {
    return (
    <form
      style={{
        display: "flex",
        flexDirection: "column",
        gap: 3,
        padding: 10,
        borderRadius: 8,
        marginBottom: 20,
        background: "#def",
      }}>
      <input
        type="text"
        placeholder="Content"
        style={{ padding: 5 }}
      />
```

```
      <input
        type="text"
        placeholder="Name"
        style={{ padding: 5 }}
      />
      <button
        type="submit"
        style={{
          padding: 8,
          background: "#0d6efd",
          color: "white",
          border: "0 none",
        }}>
        Post
      </button>
    </form>
  );
}
```

HTML `<form>` တစ်ခုကို Return ပြန်ပေးတဲ့ Component တစ်ခုပါ။ ထူးခြားချက်ရယ်လို့ ထွေထွေထူးထူး သိပ်မပါသေးဘဲ၊ လိုအပ်တဲ့ Style တွေ ရေးပေးထားတဲ့ Input နှစ်ခုနဲ့ Button တစ်ခုပါတဲ့ Form တစ်ခုသာဖြစ်ပါတယ်။ ဒီ Form Component ကို ချိတ်မသုံးရသေးလို့ စမ်းလို့မရသေးပေမယ့် နောင် စမ်းကြည့်လို့ ရတဲ့အခါ ရလဒ်က ဒီလိုပုံစံဖြစ်မှာပါ။ ကြိုမြင်ကြည့်လို့ရအောင် ကြိုပြလိုက်တာပါ။



နောက်တစ်ဆင့်အနေနဲ့ အခုလို ပြင်ပေးလိုက်ပါ။

React - src/Form.jsx

```
import { useRef } from "react";

export default function Form({ add }) {
  const contentRef = useRef();
  const nameRef = useRef();
  ...
  ...
}
```

`useRef` ကို `react` ကနေ Import လုပ်ယူပါတယ်။ Form Component Function က `add` ကို `props` အနေနဲ့ လက်ခံယူပါတယ်။ ပြီးတဲ့အခါ `useRef` ကိုသုံးပြီး `contentRef` နဲ့ `nameRef` တို့ကိုကြေညာပါတယ်။

အများအားဖြင့် React နဲ့ ကုဒ်တွေရေးတဲ့အခါ Element ရဲ့ အသွင်အပြင်၊ Content နဲ့ ဖွဲ့စည်းပုံ အပြောင်းအလဲတွေကို ကိုယ်တိုင် Manage လုပ်ဖို့ မလိုပါဘူး။ State ပေါ်မူတည်ပြီး တစ်ကြိမ် Declare လုပ်ထားပေးဖို့ပဲ လိုပါတယ်။ ဒါကြောင့် React ကို Declarative Programming လို့လည်း ခေါ်ကြပါတယ်။ လိုအပ်တဲ့ အပြောင်းအလဲတွေကို React က State Data တွေကိုကြည့်ပြီး အလိုအလျှောက် လုပ်ပေးသွားမှာပါ။ ဒါကြောင့် အပြောင်းအလဲတွေကို ကိုယ်တိုင်တိုက်ရိုက် Manage လုပ်ဖို့ မလိုဘူးလို့ ပြောတာပါ။

ဒါပေမယ့် ရံဖန်ရံခါ Element ရဲ့ Input Value ကို လိုချင်တာတို့၊ Width + Height ကို တွက်ချင်တာ၊ Element ကို Focus လုပ်ပေးချင်တာ စသည်ဖြင့် Element ကိုတိုက်ရိုက် Manage လုပ်ဖို့လိုတဲ့ကိစ္စလေး တချို့ ရှိနေပါတယ်။ အဲ့ဒီလို Element ကို ကိုယ်တိုင် တိုက်ရိုက် Manage လုပ်ဖို့လိုရင် React Reference တွေကို သုံးရပါတယ်။ နမူနာမှာ ပြထားသလို `useRef` Hook နဲ့ Reference တွေ ချိတ်ဆက်အသုံးပြုလို့ရပါတယ်။ ကြေညာထားတဲ့ Reference တွေကို Input တွေမှာ အခုလို ချိတ်ပေးလိုက်ပါ။

React - src/Form.jsx

```
<input
  ref={contentRef}
  type="text"
  placeholder="Content"
  style={{ padding: 5 }}
/>
<input
  ref={nameRef}
  type="text"
  placeholder="Name"
  style={{ padding: 5 }}
/>
```

`ref` Attribute ကိုသုံးပြီး Input နဲ့ Reference ကိုချိတ်ပေးလိုက်တာပါ။ ဒါကြောင့် Input ကို Manage လုပ်ဖို့လိုရင် သက်ဆိုင်ရာ Reference ကနေတဆင့် လုပ်လို့ရသွားပါပြီ။

ဆက်လက်ပြီး Form ရဲ့ `onSubmit` မှာ အခုလိုရေးပေးပါ။

React - src/Form.jsx

```
<form
  onSubmit={e => {
    e.preventDefault();
    const content = contentRef.current.value;
    const name = nameRef.current.value;

    add(content, name);

    e.currentTarget.reset();
  }}

  ...
>
  ...
</form>
```

`onSubmit` အတွက် Arrow Function တစ်ခု Assign လုပ်ပေးထားပါတယ်။ HTML Form ဆိုတာ မူလ သဘာဝအရ Submit ကို နှိပ်လိုက်ရင် Server ကို Request လုပ်ပေးမှာပါ။ ဒါကြောင့် နမူနာမှာ Event Variable e ကနေတဆင့် `preventDefault()` နဲ့ အဲ့ဒီလို Server Submit မလုပ်ဖို့ ပြောထားပါတယ်။

ပြီးတဲ့အခါ သက်ဆိုင်ရာ Reference တွေကနေတဆင့် Input တွေရဲ့ လက်ရှိ Value ကနေ `content` နဲ့ `name` တို့ကိုရယူပါတယ်။ အဲ့ဒီ Value တွေကို ထည့်ပေးပြီး `props` အနေနဲ့လက်ခံရရှိမယ့် `add()` ကို ခေါ်ထားပါတယ်။ နောက်ဆုံးက `reset()` ကတော့ လုပ်စရာရှိတာလုပ်ပြီးတဲ့အခါ Form Input တွေကို Clear လုပ်ပေးခိုင်းလိုက်တာပါပဲ။ ဒီအထိ ရပြီဆိုရင် `Form.jsx` မှာ ရေးရမယ့် ကုဒ်ပြည့်စုံပါပြီ။ ကုဒ် အပြည့်အစုံကို နောက်တစ်ခါ ပြန်ထည့်ပေးလိုက်ပါတယ်။

React - src/Form.jsx

```
import { useRef } from "react";
```

```
export default function Form({ add }) {
  const contentRef = useRef();
  const nameRef = useRef();

  return (
    <form onSubmit={e=> {
      e.preventDefault();
      const content = contentRef.current.value;
      const name = nameRef.current.value;

      add(content, name);

      e.currentTarget.reset();
    }}

    style={{
      display: "flex",
      flexDirection: "column",
      gap: 3,
      padding: 10,
      borderRadius: 8,
      marginBottom: 20,
      background: "#def",
    }}>
      <input
        ref={contentRef}
        type="text"
        placeholder="Content"
        style={{ padding: 5 }}
      />
      <input
        ref={nameRef}
        type="text"
        placeholder="Name"
        style={{ padding: 5 }}
```

```
      />
      <button
        type="submit"
        style={{
          padding: 8,
          background: "#0d6efd",
          color: "white",
          border: "0 none",
        }}>
        Post
      </button>
    </form>
  );
}
```

ဒီ Form Component ကို `App.jsx` မှာ ထည့်သုံးပါမယ်။ ဒီလိုပါ –

React - src/App.jsx

```
import { useState } from "react";
import List from "./List";
import Item from "./Item";
import Form from "./Form";

export default function App() {
  const [data, setData] = useState([
    { id: 1, content: "Hello, World!", name: "Alice" },
    { id: 2, content: "React is fun.", name: "Bob" },
    { id: 3, content: "Yay, interesting.", name: "Chris" },
  ]);

  const remove = id => {
    setData(data.filter(item => item.id !== id));
  };
```

```
  const add = (content, name) => {
    const id = data[data.length - 1].id + 1;
    setData([...data, { id, content, name }]);
  };

  return (
    <div style={{ maxWidth: 600, margin: "20px auto" }}>
      <h1>Yaycha</h1>

      <Form add={add} />

      <List>
        {data.map(item => {
          return (
            <Item
              key={item.id}
              item={item}
              remove={remove}
            />
          );
        })}
      </List>
    </div>
  );
}
```

လိုက်ရေးရတာ အဆင်ပြေအောင်လို့ အပြည့်အစုံပဲ ပြန်ထည့်ပေးလိုက်ပါတယ်။ ထူးခြားချက်တွေကို လိုက်ကြည့်ပါ။ အရင်ဆုံး Form ကို Import လုပ်ပါတယ်။ ပြီးတဲ့အခါ State `data` မှာ အသစ်ထပ်တိုးပေးတဲ့ လုပ်ဆောင်ချက်ကို လုပ်ပေးတဲ့ `add()` ကို ရေးပေးပါတယ်။ `content` နဲ့ `name` လက်ခံပြီး `id` ကိုတော့ `data` ရဲ့နောက်ဆုံး Item `id` မှာ (၁) တိုးပြီးအသစ်ဖန်တီးယူပါတယ်။ ပြီးတဲ့အခါ `setData()` နဲ့ State Data ကို Update လုပ်ပေးလိုက်ပါတယ်။ နောက်ဆုံးမှာ မူလလက်ရှိ `data` တွေအပြင် `id`, `content`, `name` အားလုံးပါဝင်တဲ့ Item အသစ်တစ်ခု ထပ်တိုးသွားမှာဖြစ်ပါတယ်။

React ရဲ့ ထုံးစံအတိုင်း State Data မှာ ထပ်တိုးတဲ့အတွက် UI မှာလည်း အလိုအလျှောက် ထပ်တိုးပေး သွားတာကို တွေ့ရမှာပဲဖြစ်ပါတယ်။

Conditional Rendering

Form တွေနဲ့ တိုက်ရိုက်ဆိုင်တာ မဟုတ်ပေမယ့် သိရှိထားဖို့လိုတဲ့ Conditional Rendering အကြောင်းကို တစ်ခါထဲ ထည့်ကြည့်ကြပါမယ်။ အရင်ဆုံး App Component မှာ နောက်ထပ် State Data တစ်ခု ထပ် ကြေညာပေးလိုက်ပါ။

React - src/App.jsx

```
export default function App() {
  const [showForm, setShowForm] = useState(false);
  ...
}
```

showForm နဲ့ setShowForm လို့ အမည်ကိုယ်စီပေးထားပြီး Default Value က false ဖြစ်ပါတယ်။ ပြီးတဲ့အခါ <Form> Component ကို ပြထားတဲ့ကုဒ်ကို အခုလို ပြင်ပေးလိုက်ပါ။

React - src/App.jsx

```
{showForm && <Form add={add} />}
```

JavaScript Logical And Operator ဖြစ်တဲ့ && ကိုအသုံးပြုလိုက်တာပါ။ && Operator ဟာ ရှေ့က Value true ဖြစ်မှ နောက်ကအလုပ်ကို ဆက်လုပ်တဲ့ Operator ဖြစ်ပါတယ်။ ရှေ့က Value false ဖြစ် နေရင် နောက်က အလုပ်ကို ဆက်မလုပ်ပါဘူး။ ဒါကြောင့် showForm ရဲ့ Value က true ဖြစ်မှပဲ <Form> ကို ပြပေးတော့မှာ ဖြစ်ပါတယ်။ React ကုဒ်တွေထဲမှာ ဒီနည်းနဲ့ တန်ဖိုးတစ်ခုရှိမှ အလုပ်လုပ်စေ ချင်တဲ့ အရာတွေရှိရင် ရေးကြပါတယ်။ ဒါကြောင့် အခုနေစမ်းကြည့်ရင် စောစောကထည့်ထားတဲ့ <Form> ကို တွေ့မြင်ရမှာ မဟုတ်တော့ပါဘူး။ showForm က Default false ဖြစ်နေလို့ပါ။

ဆက်လက်ပြီး <h1> Element ကို အခုလို ပြင်ရေးပေးပါ။

React - src/App.jsx

```
<h1
  style={{
    display: "flex",
    justifyContent: "space-between",
    alignItems: "center",
  }}>
  Foodie
  <button
    onClick={() => setShowForm(!showForm)}
    style={{
      width: 32,
      height: 32,
      borderRadius: 50,
      border: "0 none",
      background: showForm ? "#dc3545" : "#0d6efd",
      color: "white",
    }}>
    {showForm ? "×" : "+"}
  </button>
</h1>
```

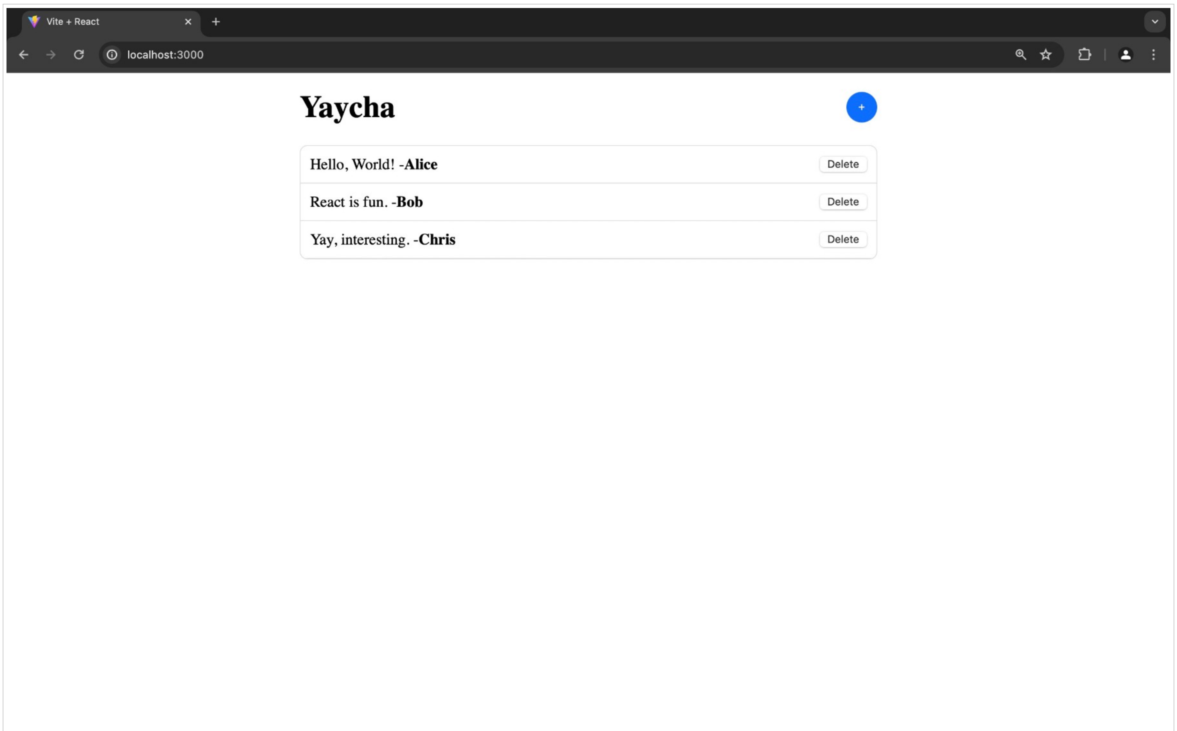
Style တွေကြောင့် ကုဒ်နည်းနည်းများသွားပေမယ့် လိုရင်းကတော့ <h1> ထဲမှာ <button> တစ်ခု ထပ်ထည့်လိုက်တာပါ။ ထူးခြားချက်တွေ ရှိပါတယ်။ Button ရဲ့ onClick မှာ `setShowForm()` နဲ့ လက်ရှိ Value ကို ! ထည့်ပြီး ပြောင်းပြန်လှန်ထားပါတယ်။ ဒါကြောင့် Button ကို နှိပ်လိုက်တဲ့အခါ `false` ဆိုရင် `true` ဖြစ်သွားမှာဖြစ်ပြီး `true` ဆိုရင် `false` ဖြစ်သွားမှာပါ။

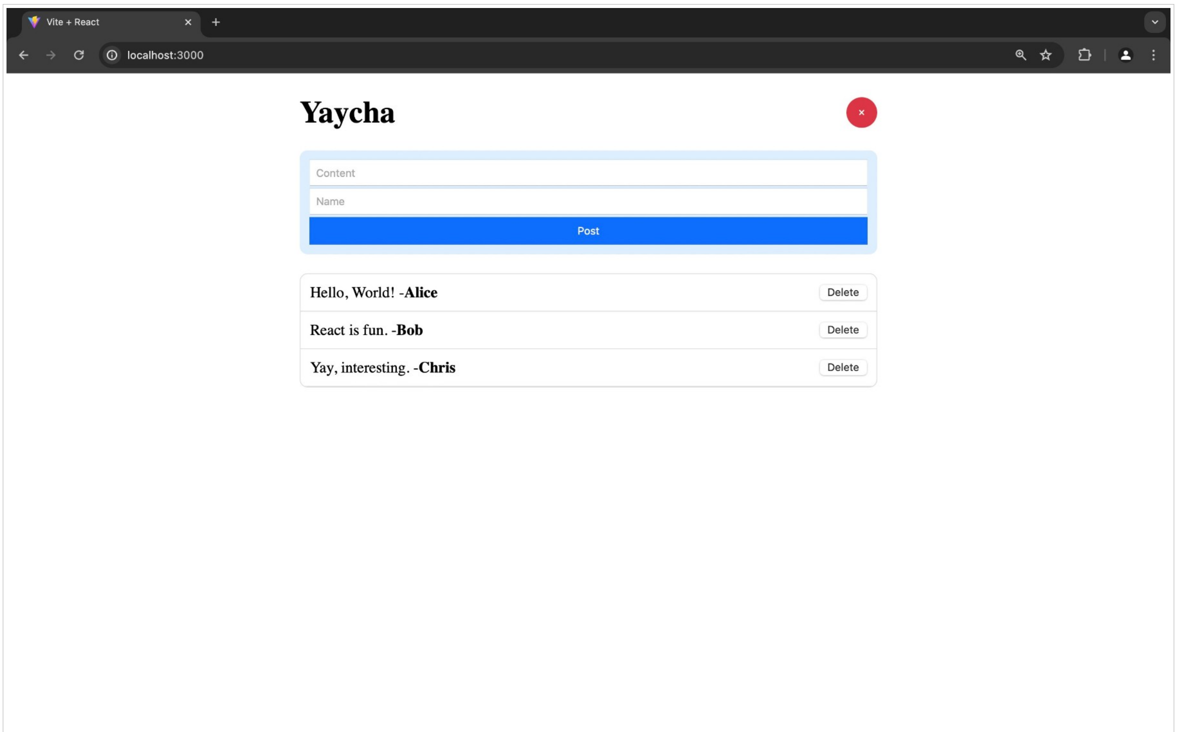
Button Style ကိုလည်း သတိပြုကြည့်ပါ။ `background` Property အတွက် Ternary Operator ကိုသုံးပြီး Condition စစ်ထားပါတယ်။ Ternary Operator ဟာအခုလို အလုပ်လုပ်ပါတယ်။

```
condition ? work-if-true : work-if-false
```

ဒါကြောင့် ရှေ့ဆုံးက Condition ဖြစ်တဲ့ `showForm` `true` ဖြစ်ရင် `?` သင်္ကေတနောက်က အနီရောင် အလုပ်လုပ်ပြီး `false` ဖြစ်ခဲ့ရင် `:` သင်္ကေတနောက်က အပြာရောင် အလုပ်လုပ်ပေးမှာ ဖြစ်ပါတယ်။ ရိုးရိုး If-Else လို Statement တွေကို JSX တွေထဲမှာ ထည့်ရေးလို့ မရပါဘူး။ Expression တွေပဲ ထည့်ရေးလို့ရ တာမို့လို့ပါ။ ဒါကြောင့် အခြေအနေပေါ်မူတည်ပြီး ပြစေချင်တဲ့အခါ Ternary Operator နဲ့ပဲရေးရပါတယ်။

ဆက်လက်ပြီး `<button>` အဖွင့်အပိတ်ထဲမှာလည်း Ternary Operator နဲ့ပဲ `showForm` `true` ဖြစ်ရင် အမြှောက်သင်္ကေတ (Close) ကို ပြခိုင်းထားပြီး `false` ဖြစ်ရင် အပေါင်းသင်္ကေတ (Add) ကို ပြခိုင်းထား ပါတယ်။ ဒါကြောင့် စမ်းကြည့်လိုက်ရင် အခုလို ရမှာဖြစ်ပါတယ်။





ဒီနည်းနဲ့ ခလုပ်နှိပ်ပြီး အဖွင့်အပိတ် Toggle လုပ်လို့ရတဲ့ Form Component တစ်ခုကို ထည့်သွင်း လိုက်ခြင်းပဲဖြစ်ပါတယ်။

အခန်း (၅) - React - Context Provider

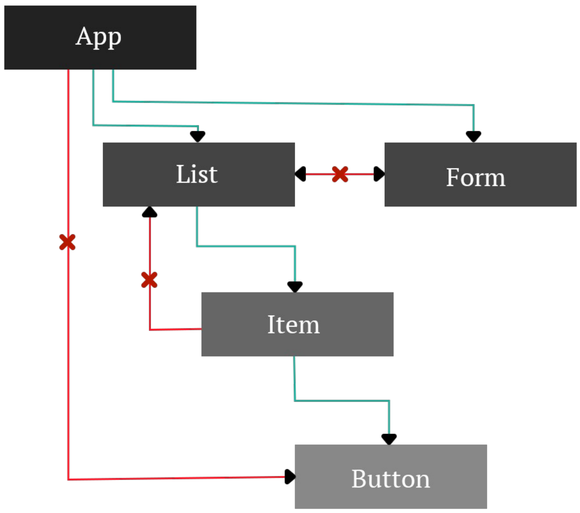
React Component တွေရဲ့ Props တွေမှာ ထူးခြားချက်တွေ ရှိကြပါတယ်။

Props တွေဟာ Read Only ဖြစ်ကြပါတယ်။ ဆိုလိုတာက Component A က Component B ကို Props တွေကို Pass လုပ်ပေးလို့ရတယ်။ ဒီသဘောကို ရှေ့ပိုင်းမှာ တွေ့ခဲ့ကြပြီးသားပါ။ အဲ့ဒီ Props တွေကို Component B က လက်ခံအသုံးပြုလို့ရတယ်။ ဒါပေမယ့် ပြင်လို့မရပါဘူး။ ပြင်လိုက်လို့လည်း Component A မှာ ရှိနေတဲ့ မူလတန်ဖိုးတွေ ပြောင်းလဲသွားမှာ မဟုတ်ပါဘူး။

Props တွေဟာ Parent Component ကနေ Child Component ကိုသာ စီးဆင်းပါတယ်။ တစ်မျိုးထဲ၊ တစ်နည်းထဲပဲ သုံးလို့ရပါတယ်။

1. Level တူ Component ချင်း Props တွေ ပေးလို့ယူလို့မရပါဘူး။
2. Child ကနေ Parent ကို ပြန်ပေးလို့လည်းမရပါဘူး။
3. Parent ကလည်း အဆင့်ကျော်ပြီး ပေးလို့မရပါဘူး။

ဒီသဘောကို နည်းနည်းပိုရှင်းသွားအောင်လို့ ပုံလေးနဲ့ပြပေးပါမယ်။



ပုံမှာလေ့လာကြည့်ပါ။ App က List နဲ့ Form ကို သုံးထားပါတယ်။ ဒါကြောင့် App ကနေ List နဲ့ Form ကို Props တွေပေးလို့ရပါတယ်။ အလားတူပဲ List က Item ကို ပေးလို့ရပါတယ်။ Item က Button ကို ပေးလို့ရပါတယ်။ ဒါပေမယ့် App က အဆင့်ကျော်ပြီး Button ကို Props တွေပေးလို့မရပါဘူး။ Item က နောက်ပြန်ပြီး Parent ဖြစ်တဲ့ List ကိုလည်း ပေးလို့မရပါဘူး။ Level တူ List နဲ့ Form နဲ့လည်း အပေးအယူလုပ်လို့ မရနိုင်ပါဘူး။

Parent ကနေ Child ကို ပေးလို့ရတယ်ဆိုတဲ့ တစ်မျိုးထဲသာ အသုံးပြုနိုင်မှာ ဖြစ်ပါတယ်။

State Data တွေမှာလည်း သတိပြုရပါမယ်။ လက်ရှိ Component ရဲ့ State Data ဟာ လက်ရှိ Component နဲ့သာ သက်ဆိုင်ပါတယ်။ Parent တွေ Child တွေ တခြား Component တွေနဲ့ သက်ဆိုင်ခြင်း မရှိပါဘူး။ State Data ကို Props အဖြစ်နဲ့ Child Component တွေကို လက်ဆင့်ကမ်းပေးလို့တော့ရပါတယ်။ State Data ကို Manage လုပ်တာကိုတော့ သက်ဆိုင်ရာ Component မှာပဲ လုပ်လို့ရပါတယ်။ ဒါပေမယ့် Manage လုပ်ပေးနိုင်တဲ့ Action တွေကို Props အနေနဲ့ Child Component တွေကို လက်ဆင့်ကမ်းပေးပြီး ပေးသုံးလို့တော့ ရပါတယ်။ ဒီသဘောတွေကိုလည်း ရှေ့ပိုင်းနမူနာတွေမှာ တွေ့ခဲ့ကြပြီးဖြစ်ပါတယ်။

ဒီအကန့်အသတ်တွေကိုဖြေရှင်းဖို့ React မှာ Context လို့ခေါ်တဲ့ လုပ်ဆောင်ချက်တစ်ခုပါဝင်ပါတယ်။ အသုံးပြုပုံက သိပ်မခက်လှပါဘူး။ ဒီလိုပါ –

React

```
import { createContext, useContext } from "react";

const AppContext = createContext();

function App() {
  return <AppContext.Provider value="Yaycha">
    <Home />
  </AppContext.Provider>
}

function Home() {
  return <div>
    <Header />
    <Footer />
  </div>
}

function Header() {
  const title = useContext(AppContext);
  return <h1>{title}</h1>
}

function Footer() {
  const title = useContext(AppContext);
  return <footer>
    Copyright {title}
  </footer>
}
```

လက်တွေ့ချရေးဖို့က ခဏနေမှ ကြည့်ကြပါမယ်။ ကုဒ်ရဲ့ဖွဲ့စည်းပုံကိုနားလည်အောင် အရင်လိုက်ဖတ်

ကြည့်ပါ။ အရင်ဆုံး `createContext` နဲ့ `useContext` တို့ကို `react` ကနေ Import လုပ်ယူပါတယ်။ ပြီးတဲ့အခါ `createContext` ကို သုံးပြီး `AppContext` ကို ဖန်တီးယူလိုက်ပါတယ်။ App Component မှာ `AppContext.Provider` ကိုသုံးထားပါတယ်။ `value` အဖြစ် String တစ်ခုကို ပေးထားတာ တွေ့ရပါလိမ့်မယ်။ <u>Context Provider အတွင်းမှာရှိတဲ့ Component တိုင်းက ဒီ Value ကို ယူသုံးလို့ရသွားပါပြီ</u>။

ဒါကြောင့် `Header` Component နဲ့ `Footer` Component တို့က `useContext` ကိုသုံးပြီး Context Value ကို ယူသုံးထားတာကို တွေ့ရမှာပဲ ဖြစ်ပါတယ်။ Context Value က `App` မှာ ရှိနေတာဆိုပေမယ့် `Header` တို့ `Footer` တို့က နောက်ပြန်အဆင့်ကျော်ပြီး လှမ်းယူသုံးလိုက်တာပါ။ ဒီနည်းနဲ့ စီးဆင်းမှု အကန့်အသတ်ရှိတဲ့ State တွေ Props တွေကို Context နဲ့ ပေါင်းစပ်လိုက်တဲ့အခါ အသုံးပြုရတာ ပိုမိုကျယ်ပြန့် ပြည့်စုံသွားတဲ့သဘောပဲ ဖြစ်ပါတယ်။

လုပ်လက်စ ပရောဂျက်မှာ နမူနာရေးပြီး စမ်းကြည့်ကြပါမယ်။ `ThemedApp.jsx` အမည်နဲ့ ဖိုင်အသစ် တစ်ခု တည်ဆောက်ပြီး ဒီကုဒ်ကို ရေးပေးပါ။

React - src/ThemedApp.jsx

```
import { createContext } from "react";

import App from "./App";

export const AppContext = createContext();

export default function ThemedApp() {
  return (
    <AppContext.Provider value={{ mode: "dark" }}>
      <App />
    </AppContext.Provider>
  );
}
```

`createContext` နဲ့ `AppContext` တစ်ခု ဖန်တီးထားပါတယ်။ ထူးခြားချက်အနေနဲ့ <u>`AppContext`</u>

ကို Export ပေးထားတာကိုလည်း သတိပြုပါ။ ဒါမှတခြား ကုဒ်ဖိုင်တွေက လိုအပ်ရင် သူ့ကို Import ယူသုံးလို့ ရမှာပါ။ `App` ကို `AppContext.Provider` ထဲထည့်ပြီး ပြန်ပို့ထားပါတယ်။ ဒါကြောင့် `AppContext` ရဲ့ Value ဖြစ်တဲ့ `{ mode }` ကို App နဲ့ တခြား Component တွေက ယူသုံးလို့ ရနိုင်သွားပါပြီ။

ဆက်လက်ပြီး `index.css` ထဲမှာ ဒီကုဒ်ကိုရေးပေးပါ။

React - src/index.css

```css
html, body {
  margin: 0;
  padding: 0;
}
```

ဘာမှထူးထူးဆန်းဆန်းမပါပါဘူး။ မလိုလားအပ်တဲ့ Spacing တွေပါနေမှာစိုးလို့ ဖြုတ်လိုက်တာပါ။ ဆက်လက်ပြီး `main.jsx` ကို အခုလိုပြင်ပေးပါ။

React - src/main.jsx

```jsx
import React from "react";
import ReactDOM from "react-dom/client";
import "./index.css";

import ThemedApp from "./ThemedApp";

ReactDOM.createRoot(document.getElementById("root")).render(
  <React.StrictMode>
    <ThemedApp />
  </React.StrictMode>
);
```

`index.css` ကို Import လုပ်ထားသလို `main.jsx` က ယူသုံးထားတာလည်း `App` မဟုတ်တော့ပါဘူး။ `ThemedApp` ဖြစ်သွားပါပြီ။

ဆက်လက်ပြီး `App.jsx` မှာ အခုလို ပြင်ပေးပါ။

React - src/App.jsx

```
import { useState, useContext } from "react";

import { AppContext } from "./ThemedApp";

// other imports

export default function App() {
  const { mode } = useContext(AppContext);

  const [showForm, setShowForm] = useState(false);
  const [data, setData] = useState([ ... ]);
  const remove = id => { ... };
  const add = (content, name) => { ... };

  return (
    <div
      style={{
        minHeight: 1500,
        background: mode === "dark" ? "black" : "white",
        color: mode === "dark" ? "white" : "black",
        paddingTop: 20,
      }}>
      <div style={{ maxWidth: 600, margin: "0 auto" }}>
        <h1
          style={{
            display: "flex",
            justifyContent: "space-between",
            alignItems: "center",
            margin: "0 0 20px 0",
          }}>
```

```
          Yaycha
          <button ...> ... </button>
        </h1>

        {showForm && <Form add={add} />}

        <List> ... </List>
      </div>
    </div>
  );
}
```

ပြင်စရာမလိုတဲ့ကုဒ်တွေကို ချုံ့ပြီး ပြင်ရမယ့်အပိုင်းတွေကိုပဲ အထူးပြုပြီး ပြထားပါတယ်။ အရင်ဆုံး `useContext` ကို Import လုပ်ထားပါတယ်။ `AppContext` ကိုလည်း `ThemedApp` ကနေ Import လုပ်ယူထားပါတယ်။ အဲ့ဒီနှစ်ခုကို ပေါင်းစပ်ပြီး `useContext(AppContext)` နဲ့ Context Value ကို ရယူထားပါတယ်။ လက်ရှိ Value က `{ mode: "dark" }` ဖြစ်မှာပါ။

ဆက်လက်ပြီး မူလရေးလက်စ UI ကုဒ်တွေကို `<div>` Element တစ်ခုထဲမှာ စုထည့်လိုက်ပါတယ်။ `<div>` Element ရဲ့ `background` နဲ့ `color` Style တွေကို `mode` ပေါ်မူတည်ပြီး ပြအောင်ရေးပေး ထားပါတယ်။ ဒါကြောင့် လက်ရှိ `{ mode: "dark" }` ဖြစ်နေချိန်မှာ Background အမဲရောင်နဲ့ စာ အဖြူရောင်တို့ကိုသုံးပြီး ပြပေးမှာ ဖြစ်ပါတယ်။

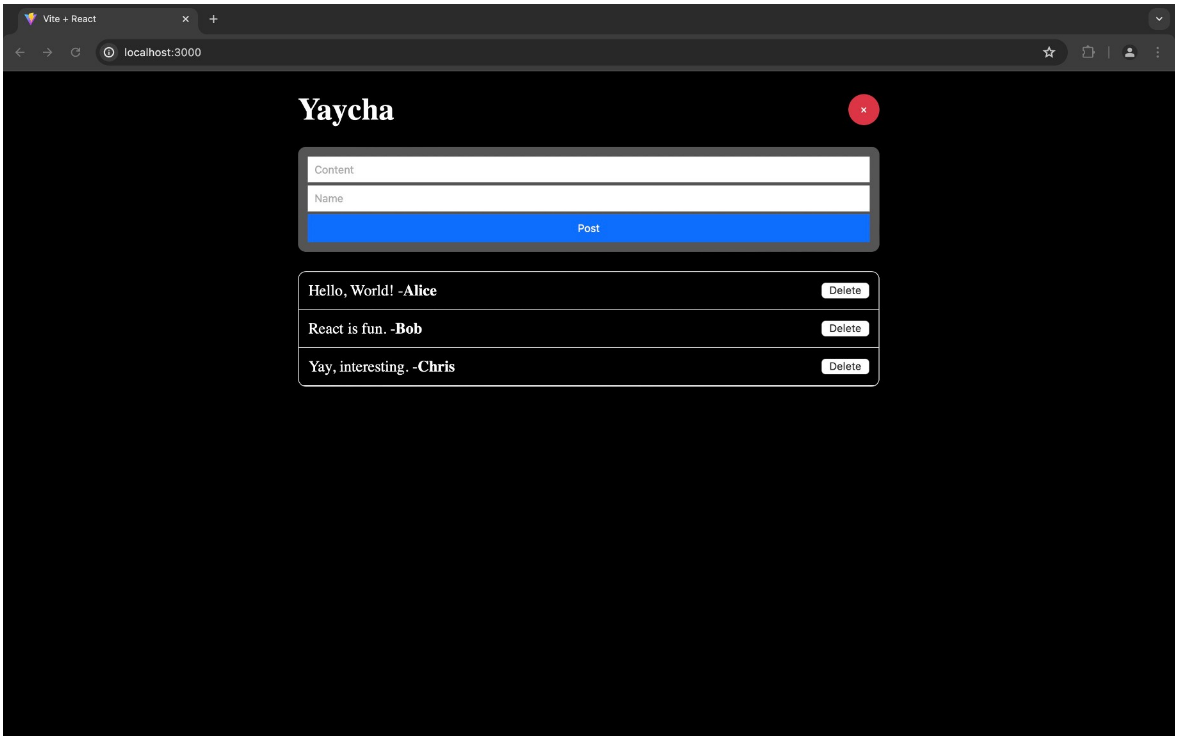
နောက်တစ်ဆင့်အနေနဲ့ `<Form>` Component ကို အခုလိုပြင်ပေးလိုက်ပါ။

React - src/Form.jsx

```
import { useContext, useRef } from "react";

import { AppContext } from "./ThemedApp";

export default function Form({ add }) {
  const { mode } = useContext(AppContext);
```

```
  const contentRef = useRef();
  const nameRef = useRef();

  return (
    <form
      onSubmit={ ... }
      style={{
        display: "flex",
        flexDirection: "column",
        gap: 3,
        padding: 10,
        borderRadius: 8,
        marginBottom: 20,
        background: mode === "dark" ? "#555" : "#def",
      }}>
      <input ... />
      <input ... />
      <button ...> ... </button>
    </form>
  );
}
```

အလားတူပါပဲ။ `useContext` နဲ့ `mode` ကိုယူပြီး Form ရဲ့ Background ကို `mode` ပေါ်မူတည်ပြီး ပြ အောင်ရေးပေးလိုက်တာပါ။ `{ mode: "dark" }` ဆိုရင် မှောင်တဲ့ Background ကိုသုံးမှာဖြစ်ပြီး မဟုတ်ရင်တော့ ပိုလင်းတဲ့ Background ကိုသုံးသွားမှာ ဖြစ်ပါတယ်။ ဒါကြောင့် အခုနေစမ်းကြည့်ရင် ရလဒ်က အခုလို ဖြစ်ပါလိမ့်မယ်။



ပိုကြည့်ကောင်းချင်ရင် အသေးစိတ်ညှိရမှာတွေ ကျန်သေးပေမယ့်၊ အသေးစိတ်တော့ လိုက်မညှိတော့ပါဘူး။ နောက်တစ်ခန်းမှာ UI Framework တစ်ခုနဲ့ ဆက်လေ့လာမှာဖြစ်လို့ မလိုအပ်ဘဲ အသေးစိတ်တွေ မလုပ်ချင်သေးလို့ပါ။ စမ်းကြည့်ချင်ရင် `ThemedApp` ရဲ့ Context Provider Value မှာ `mode` ကို `"light"` လို့ပြောင်းပြီး စမ်းကြည့်လို့ရပါတယ်။ ဒီ Value ပေါ်မူတည်ပြီး App ရဲ့ Color Mode ကို ပြပေးနိုင်သွားပြီဆိုတာကို တွေ့ရပါလိမ့်မယ်။

ဒီနည်းနဲ့ Context Value တွေကို လိုတဲ့နေရာကနေ ချိတ်ယူနိုင်ခြင်းအားဖြင့် အသုံးဝင်တဲ့ လုပ်ဆောင်ချက်တွေကို ရရှိနိုင်ပါတယ်။ လက်ရှိနမူနာမှာ Context Value က အသေဖြစ်နေလို့ ဒီ Value ကို ပြောင်းလို့ရအောင် ဆက်လုပ်ကြပါဦးမယ်။

React - src/ThemedApp.jsx

```
import { useState, createContext } from "react";

import App from "./App";

export const AppContext = createContext();

export default function ThemedApp() {
    const [mode, setMode] = useState("dark");

  return (
    <AppContext.Provider value={{ mode, setMode }}>
      <App />
    </AppContext.Provider>
  );
}
```

mode Value ကို အသေရေးပေးထားတာ မဟုတ်တော့ဘဲ State Data အဖြစ် ပေးထားတာ ဖြစ်သွားပါပြီ။ Context Provider ရဲ့ Value ကိုလည်း သတိပြုပါ။ mode အပြင် setMode ကိုပါ Context အနေနဲ့ ယူသုံးလို့ရအောင် ပေးထားပါတယ်။

ဆက်လက်ပြီး App.jsx မှာ mode အပြင် setMode ကိုပါ အခုလို Context ကနေ တွဲယူလိုက်ပါ။

React - src/App.jsx

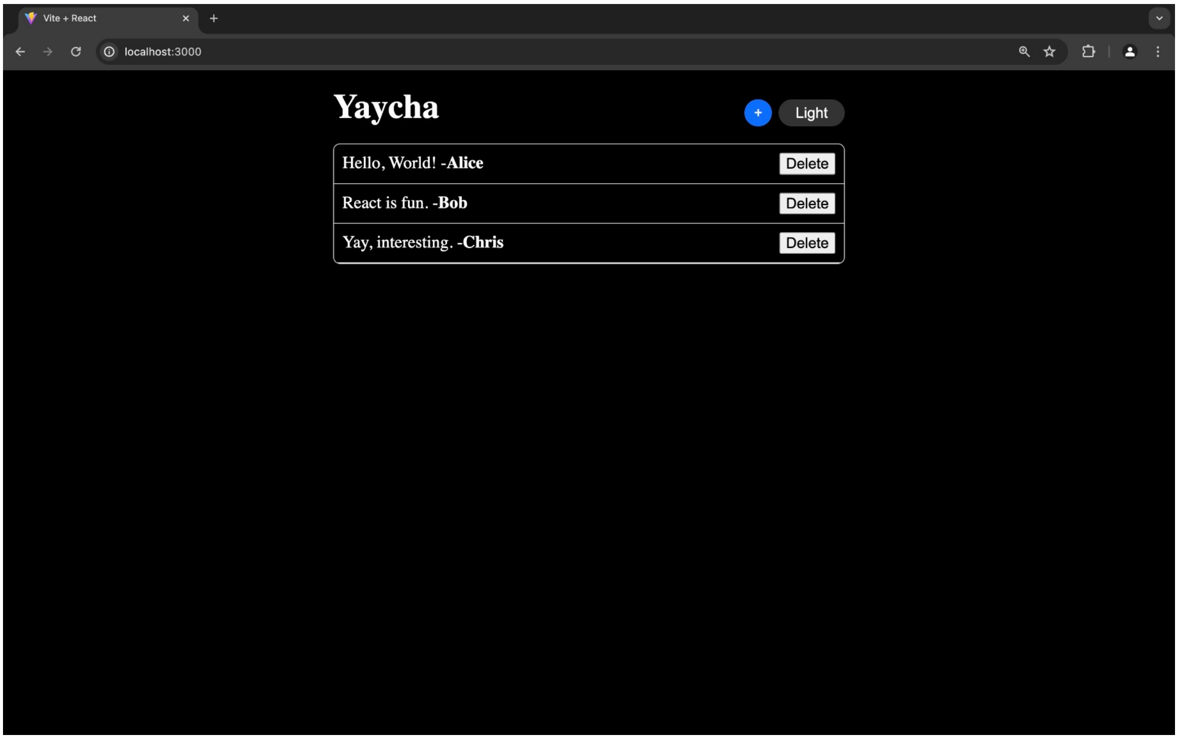
```
const { mode, setMode } = useContext(AppContext);
```

ပြီးတဲ့အခါ <h1> Element ကို အခုလို ဖြည့်စွက်ပေးလိုက်ပါ။

React - src/App.jsx

```
<h1 ...>
  Yaycha
  <div>
    <button ...> ... </button>

    <button
      onClick={() =>
        setMode(mode === "dark" ? "light" : "dark")
      }
      style={{
        marginLeft: 8,
        padding: "0 20px",
        height: 32,
        borderRadius: 32,
        border: "0 none",
        background: mode === "dark" ? "#333" : "#ddd",
        color: mode === "dark" ? "white" : "black",
      }}>
      {mode === "dark" ? "Light" : "Dark"}
    </button>
  </div>
</h1>
```

Add Button ရဲ့နောက်မှာ နောက်ထပ် `<button>` တစ်ခုထည့်လိုက်တာပါ။ သူ့ရဲ့ `onClick` မှာ `setMode` ကို သုံးပြီး `mode` Value တွေပြောင်းလို့ရအောင် ရေးပေးထားပါတယ်။ ဒါကြောင့် ဒီခလုပ်ကို နှိပ်ပြီး `mode` Value ကို ပြောင်းလို့ရသွားပါပြီ။



အရမ်းအသုံးဝင်ပါတယ်။ Light Mode, Dark Mode လုပ်ဆောင်ချက်ရအောင် ကိုယ့်ဘာသာ ဖန်တီးယူလိုက်တာပါ။ လက်တွေ့မှာ အကုန်လုံးကို ကိုယ့်ဘာသာလုပ်စရာတော့ မလိုပါဘူး။ လုပ်ပေးနိုင်တဲ့ UI Framework ကောင်းတွေ ရှိနေပါတယ်။ နောက်အခန်းမှာ ဆက်လက်လေ့လာကြပါမယ်။

အခန်း (၆) - React - UI Framework - MUI

ဒီအခန်းမှာ လုပ်လက်စ **yaycha** ပရောဂျက်ကို UI Framework တစ်ခုနဲ့ ပိုပြီးတော့ သပ်သပ်ရပ်ရပ်ဖြစ်သွားအောင် လုပ်ကြမှာပါ။ React အတွက် အဆင်သင့်သုံးလို့ရတဲ့ UI Framework တွေအများကြီး ရှိပါတယ်။ ဒီစာရေးနေချိန်မှာ လူကြိုက်များနေကြတာက ဒီ Framework တွေပါ။

- MUI
- Radix UI
- shadcn/ui
- React Aria Components

အားလုံးက သူ့နည်းသူ့ဟန်နဲ့ ကောင်းမွန်အသုံးဝင်ကြပါတယ်။ ဒီထဲမှာ MUI က အစောဆုံးပေါ်တာဖြစ်သလို သူ့ Component Style တွေက အနည်းငယ် ဒိတ်အောက်သလိုတော့ ဖြစ်နေပါတယ်။ ဒါပေမယ့် Components, Layouts, Icons, Utilities, Responsive, Themes, Customization စသည်ဖြင့် ဘက်စုံပါဝင်ပြီး လက်တွေ့အသုံးများတဲ့ နည်းပညာတစ်ခုပါ။ ပရောဂျက်တစ်ခုမှာ လိုအပ်နိုင်သမျှ အကုန်အသင့်ရနိုင်သလို Documentation ကလည်း ပြည့်စုံထောင့်စိပါတယ်။

MUI – https://mui.com

MUI လို အားလုံးအသင့်ရနိုင်တဲ့ UI Framework တွေရဲ့ပြဿနာကတော့ သူ့ကိုအသုံးပြုတည်ဆောက်တဲ့ ပရောဂျက်တွေဟာ သူ့စတိုင်ဘောင်ထဲမှာပဲ ရှိနေတတ်ခြင်းဖြစ်ပါတယ်။ တစ်ခုနဲ့တစ်ခု ခပ်ဆင်ဆင်တွေ

ဖြစ်နေတတ်တာပါ။ မတူကွဲပြားပြီး Unique ဖြစ်တဲ့ စတိုင်အသွင်အပြင်လိုချင်တဲ့ ပြင်ဆင်မယ်ဆိုရင် ပြင်ဆင်အသုံးပြုရခက်တတ်ပါတယ်။

Radix UI ကတော့ ပိုခေတ်မီတဲ့ စတိုင်အသွင်အပြင်တွေ ပါဝင်ပါတယ်။ သူလည်း အသင့်သုံးလို့ရတာတွေ ပြည့်စုံပါတယ်။ အထက်မှာပြောခဲ့သလို သူ့ကိုသုံးထားတဲ့ UI တိုင်းခပ်ဆင်ဆင်ဖြစ်နေတယ်ဆိုတဲ့ ပြဿနာကို ဖြေရှင်းဖို့ Primitive Components ခေါ် ကျန်လိုအပ်ချက်တွေ အကုန်လုပ်ပေးထားပေမယ့် စတိုင်တွေ ကြိုလုပ်ပေးမထားဘဲ၊ မိမိစိတ်ကြိုက် Style တွေရေးသားအသုံးပြုနိုင်တဲ့ နည်းပညာလည်း ပူးတွဲပြီး ထည့်ပေးထားပါတယ်။ ကိုယ့်ဘာသာ Style တွေရေးရမှာဖြစ်လို့ အလုပ်တော့ နည်းနည်းပိုသွားမယ်၊ ဒါပေမယ့် သီးခြား Unique ဖြစ်တဲ့ အသွင်အပြင်တွေကို ရနိုင်မှာဖြစ်ပါတယ်။

Radix – https://www.radix-ui.com

shadcn/ui ကတော့ Radix Primitive ပေါ်မှာ အခြေခံပြီး TailwindCSS နဲ့ Customize လုပ်ရလွယ်အောင် လုပ်ပေးထားတဲ့ UI Framework ပါ။ သူ့နည်းကလည်း ထူးခြားပါတယ်။ ပုံမှန်အားဖြင့် UI Framework တွေကို ကိုယ့်ပရောဂျက်မှာ ထည့်သွင်းအသုံးပြုတဲ့အခါ မူရင်း Package ကနေ လိုအပ်တဲ့ Component ကို Import လုပ်ယူပြီး သုံးကြရလေ့ရှိပါတယ်။ shadcn/ui က UI Component တွေကို ကိုယ့်ပရောဂျက်ထဲ တိုက်ရိုက်ထည့်သိမ်းပေးသွားမှာ ဖြစ်ပါတယ်။ ဒါကြောင့် ပြင်ချင်ရင် ကိုယ့်ပရောဂျက်ထဲမှာပဲ မူရင်း Component ကို စိတ်ကြိုက်ပြင်သုံးသွားလို့ရပါတယ်။ ဒီလိုထူးခြားတဲ့ နည်းစနစ်သစ်ကို လူတွေအတော် သဘောကျနေကြပါတယ်။

shadcn/ui – https://ui.shadcn.com

React Aria Components တွေကတော့ Radix Primitive လိုပဲ Headless UI (သို့) Unstyled Components ခေါ် စတိုင်တွေ ကြိုလုပ်ပေးမထားဘဲ ကိုယ့်စိတ်တိုင်းကျ လုပ်ယူလို့ရအောင် ပြင်ဆင်ပေးထားတဲ့ နည်းပညာပါ။ ခပ်ဆင်ဆင် UI တွေကို တွေ့ရတာများပြီး ရိုးအီလာကြတဲ့အချိန်ဖြစ်လို့ ဒီလို Unique စတိုင်တွေ ကိုယ့်ဘာသာလုပ်ဖို့ ပံ့ပိုးပေးတဲ့ နည်းပညာမျိုးကို နှစ်သက်နေကြချိန်ဖြစ်ပါတယ်။ စတိုင်တွေ ကြိုလုပ်ပေးမထားပေမယ့် လိုအပ်မယ့် UI Interaction တွေ Accessibility Feature တွေ အားလုံးအသင့် ပါဝင်ပါတယ်။ TailwindCSS ကိုသုံးပြီး စတိုင်တွေကို Customize လုပ်ရမှာပါ။

React Aria Components – https://react-spectrum.adobe.com/react-aria/index.html

ဒီအခန်းမှာတော့ MUI ကို အသုံးပြုသွားမှာဖြစ်ပြီး၊ နောက်ပိုင်းအခန်းတွေမှာ shadcn/ui ကိုလည်း ထည့်သုံးပြပေးမှာပါ။

MUI

MUI ကိုအသုံးပြုဖို့အတွက် ပထမဆုံးအနေနဲ့ လိုအပ်တဲ့ Package တွေကို Install လုပ်ပေးရပါမယ်။ လုပ်လက်စ **yaycha** ပရောဂျက်ထဲမှာ အခုလို Install လုပ်ပေးပါ။

```
npm install @mui/material @emotion/react @emotion/styled
```

Install လုပ်တဲ့ထဲမှာပါတဲ့ Emotion ဆိုတာ CSS ကုဒ်တွေကို JavaScript ထဲမှာတွဲရေးလို့ရအောင် လုပ်ပေးတဲ့ နည်းပညာတစ်မျိုးပါ။ MUI က Emotion ကိုအသုံးပြုပြီး အလုပ်လုပ်တာဖြစ်လို့ တစ်ခါထဲ ထည့် Install လုပ်ပေးရတာပါ။ ဆက်လက်ပြီး Icon တွေ Font တွေ Install လုပ်ပေးပါမယ်။

```
npm install @mui/icons-material @fontsource/roboto
```

MUI နဲ့အတူတွဲပါတဲ့ Material Icons တွေကို Install လုပ်ပြီး Roboto ဖွန့်ကိုလည်း Install လုပ်လိုက်ပါတယ်။ Material Design တို့ Roboto ဖွန့်တို့ဆိုတာ Google ရဲ့ နည်းပညာတွေဖြစ်ပြီး Android Device တွေမှာသုံးတဲ့ နည်းပညာတွေဖြစ်လို့ MUI ကို အသုံးပြုဖန်တီးတဲ့ UI တွေကလည်း Google/Android ဆန်နေတာကို တွေ့ရပါလိမ့်မယ်။

ဆက်လက်ပြီး ဖွန့်ဖိုင်တွေကို Import လုပ်ပေးရပါမယ်။ `main.jsx` မှာအခုလို ထည့်ပေးလိုက်ပါ။

React - src/main.jsx

```
...
import '@fontsource/roboto/300.css';
import '@fontsource/roboto/400.css';
import '@fontsource/roboto/500.css';
import '@fontsource/roboto/700.css';
...
```

လိုအပ်တာတွေ ထည့်သွင်းချိတ်ဆက်ပြီးပြီဆိုရင် စတင်အသုံးပြုလို့ရပါပြီ။ `ThemedApp.jsx` ကို အခုလို ပြင်ရေးပေးရပါမယ်။

React - src/ThemedApp.jsx

```
import { useState, createContext, useContext } from "react";

import {
  CssBaseline,
  ThemeProvider,
  createTheme,
} from "@mui/material";

import App from "./App";

const theme = createTheme({
  palette: {
    mode: "dark",
  },
});

const AppContext = createContext();

export function useApp () {
    return useContext(AppContext);
}
```

```
export default function ThemedApp() {
    const [showForm, setShowForm] = useState(false);

  return (
    <ThemeProvider theme={theme}>
      <AppContext.Provider value={{ showForm, setShowForm }}>
        <App />
        <CssBaseline />
      </AppContext.Provider>
    </ThemeProvider>
  );
}
```

CssBaseline, ThemeProvider နဲ့ createTheme တို့ကို MUI ကနေ Import လုပ်ယူထားပါတယ်။ Light Mode, Dark Mode လုပ်ဆောင်ချက်ကို ကိုယ့်ဘာသာ လုပ်ထားတာတွေ မသုံးတော့ပါဘူး။ MUI ကို အသုံးပြုပြီး ဆက်လုပ်မှာပါ။ ဒါကြောင့် createTheme နဲ့ MUI Theme တစ်ခုဖန်တီးပါတယ်။ လောလောဆယ် mode ကို dark လို့ပဲ ခဏအသေပေးထားပါတယ်။

useApp လို့ခေါ်တဲ့ Function တစ်ခုထည့်ရေးပေးထားတာကိုလည်း သတိပြုပါ။ အရင်က AppProvider ရဲ့လုပ်ဆောင်ချက်တွေကို ယူသုံးချင်ရင် အခုလိုသုံးရမှာပါ။

React

```
import { useContext } from "react";
import { AppContext } form "./ThemedApp";

const { mode } = useContext(AppContext);
```

useContext ကနေတစ်ဆင့်သုံးရခြင်းဖြစ်ပါတယ်။ အဲ့ဒီလို useContext ကိုခဏခဏတစ်ဆင့်ခံ သုံးနေစရာမလိုအောင် useApp ကိုထည့်ရေးပေးထားတာပါ။ ဒါကြောင့် အခုနေ AppProvider ကိုလိုအပ်ရင် အခုလို သုံးပြုနိုင်သွားမှာ ဖြစ်ပါတယ်။

React

```
import { useApp } from "./ThemedApp";

const { mode } = useApp();
```

ကြီးမားတဲ့ အပြောင်းအလဲကြီးတော့ မဟုတ်ပါဘူး။ ယူသုံးချိန်မှာ ရေးရတာနည်းနည်းလျော့သွားတာပါ။ မူလက App Context Provider နဲ့ ပေးထားတဲ့ `mode` နဲ့ `setMode` တို့ကို ဆက်မသုံးတော့ပါဘူး။ ဖယ်ထုတ်လိုက်ပါပြီ။ သူတို့အစား နောက်ပိုင်းလိုအပ်မယ့် `showForm` နဲ့ `setShowForm` တို့ကို အစားထိုး အသုံးပြုထားပါတယ်။ `return` ပြန်ပေးချိန်မှာ Provider နှစ်ထပ်ဖြစ်သွားတာကိုလည်း သတိပြုပါ။ MUI ရဲ့ Theme Provider နဲ့ ကိုယ့်ဘာသာရေးထားတဲ့ AppContext Provider တို့ဖြစ်ကြပါတယ်။

`<CssBaseline />` လို့ခေါ်တဲ့ Component တစ်ခုလည်း ထည့်သုံးထားတာကို တွေ့ရပါလိမ့်မယ်။ MUI Theme အလုပ်လုပ်ပုံ မှန်ဖို့အတွက် မဖြစ်မနေ ထည့်သုံးပေးဖို့လိုအပ်ပါတယ်။ သူ့ရဲ့ အလုပ်လုပ်ပုံက CSS Reset နဲ့ အလုပ်လုပ်ပုံ ဆင်တူပါတယ်။ ဒါကြောင့် အစောပိုင်းက ကိုယ့်ဘာသာရေးထည့်ထားတဲ့ `index.css` ကို ပြန်ဖယ်ထုတ်ချင်ရင် ထုတ်လိုက်လို့ ရပါပြီ။ `margin: 0, padding: 0` အပါအဝင် လိုအပ်တဲ့ Style Reset တွေကို `CssBaseline` က လုပ်ပေးသွားပါလိမ့်မယ်။ ကိုယ့်ဘာသာ ထပ်လုပ်ပေးဖို့ မလိုအပ်တော့ပါဘူး။

ဆက်လက်ပြီး နောက်ပိုင်းမှာ ပါမယ့်ကုဒ်တွေ များလာမှာဖြစ်လို့ Organize ဖြစ်သွားအောင် ပရောဂျက် ဖိုဒါ Structure ကို အခုလို ပြင်ပေးလိုက်သင့်ပါတယ်။

```
yaycha
├── README.md
├── index.html
├── node_modules/
├── package-lock.json
├── package.json
├── public/
```

```
├── src/
│   ├── App.jsx
│   ├── ThemedApp.jsx
│   ├── assets/
│   ├── components/
│   │   ├── Header.jsx
│   │   ├── Form.jsx
│   │   └── Item.jsx
│   └── main.jsx
└── vite.config.js
```

`components` ဖိုဒါထဲမှာ ရေးလက်စ `Form.jsx` နဲ့ `Item.jsx` တို့ကို ခွဲထားလိုက်တာဖြစ်ပြီး `Header.jsx` အမည်နဲ့ ဖိုင်အသစ်တစ်ခု ထပ်ထည့်ထားပါတယ်။ ရေးလက်စ `List.jsx` ကိုတော့ မလိုအပ်တော့လို့ ဖယ်ထုတ်လိုက်ပါပြီ။ `Header.js` ထဲမှာ အခုလို ရေးပေးပါ။

React - src/components/Header.jsx

```
import { useApp } from "../ThemedApp";

import {
    Box,
    AppBar,
    Toolbar,
    Typography,
    IconButton,
} from "@mui/material";

import {
  Menu as MenuIcon,
  Add as AddIcon,
  LightMode as LightModeIcon,
} from "@mui/icons-material";

export default function Header() {
  const { showForm, setShowForm } = useApp();
```

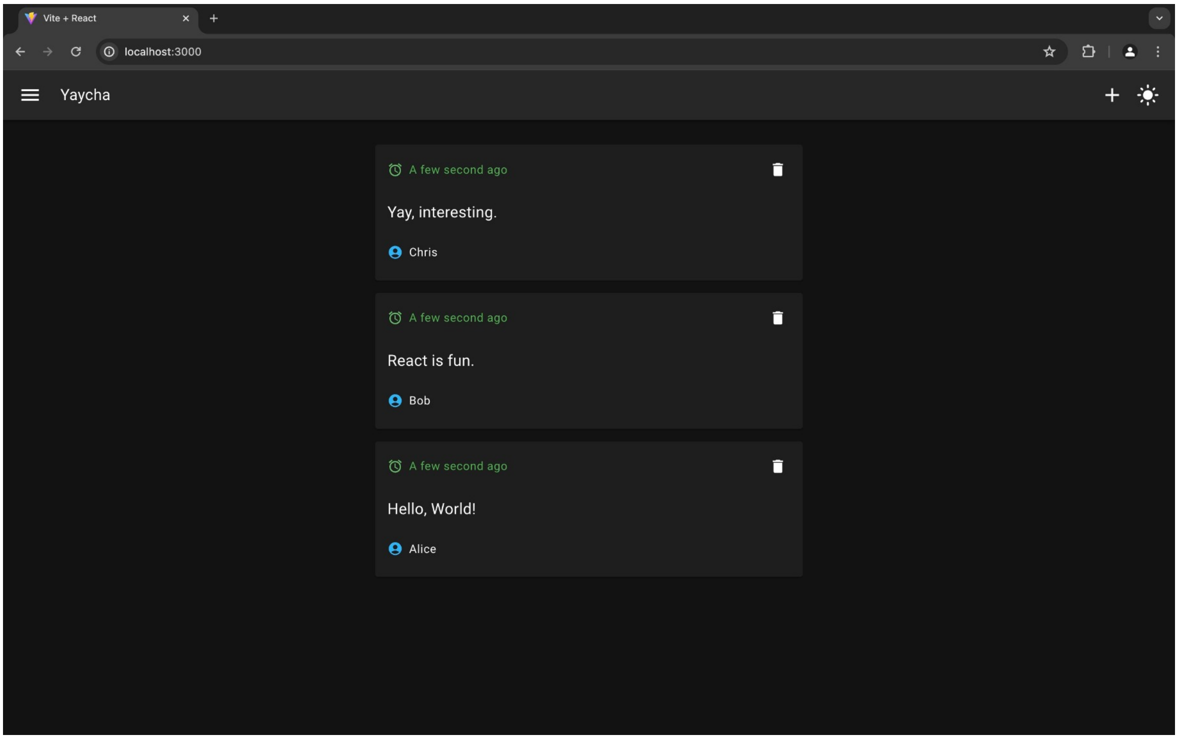
```
  return (
    <AppBar position="static">
      <Toolbar>
        <IconButton
          color="inherit"
          edge="start">
          <MenuIcon />
        </IconButton>

        <Typography sx={{ flexGrow: 1, ml: 2 }}>Yaycha</Typography>

        <Box>
          <IconButton
            color="inherit"
            onClick={() => setShowForm(!showForm)}>
            <AddIcon />
          </IconButton>
          <IconButton
            color="inherit"
            edge="end">
            <LightModeIcon />
          </IconButton>
        </Box>
      </Toolbar>
    </AppBar>
  );
}
```

ကုဒ်တွေများပေမယ့် အများအားဖြင့် MUI Component တွေကို ယူသုံးထားတာပါ။ `<Box>` Component ကို `<div>` တွေအစား လိုအပ်တဲ့နေရာတိုင်းမှာ သုံးနိုင်ပါတယ်။ ပုံမှန်အားဖြင့် React Component တွေမှာ `style` Props နဲ့ လိုအပ်တဲ့ စတိုင်တွေ ထည့်ရေးနည်းကို ရှေ့ပိုင်းမှာ ကြည့်ခဲ့ပြီးသားပါ။ MUI မှာ `sx` ခေါ် ပိုပြည့်စုံတဲ့နည်းကို သုံးရပါတယ်။ ရိုးရိုး CSS Style တွေအပြင် Bootstrap တို့ TailwindCSS တို့လို

Utilities တွေကိုလည်း `sx` နဲ့ တွဲသုံးလို့ရနိုင်ပါတယ်။ သုံးနည်းတွေ သူ့နေရာနဲ့သူ ပါလာပါလိမ့်မယ်။ `sx` ကို ရိုးရိုး HTML Elements တွေမှာ သုံးလို့မရပါဘူး။ MUI Component တွေမှာပဲ သုံးလို့ရပါတယ်။ ဒါကြောင့် `<div>` လိုတဲ့နေရာတိုင်းမှာ `<Box>` ကို သုံးသင့်ပြီး စာတွေပြဖို့အတွက် `<h1>…</h6>`, `<p>`, `<span>` စတဲ့ Elements တွေအစား `<Typography>` ကို သုံးသင့်ပါတယ်။

နမူနာမှာ `AppBar` Component ကိုသုံးထားပါတယ်။ `AppBar` ထဲမှာ `Toolbar` ရှိပြီး `Toolbar` ထဲမှာ `IconButton`, `Typography` နဲ့ `Box` တို့ရှိကြတာကို တွေ့ရပါလိမ့်မယ်။ `IconButton` ထဲမှာ `MenuIcon` ကိုထည့်ပြထားပါတယ်။ ဘာအလုပ်မှတော့မလုပ်သေးပါဘူး။ ပြယုံသက်သက်ပါပဲ။ Typography ကို `sx` နဲ့ `flexGrow: 1` လို့ သတ်မှတ်ပေးထားတဲ့အတွက် အပြည့်နေရာယူပါလိမ့်မယ်။ Margin Left အတွက် CSS ရဲ့ `marginLeft` ကို မသုံးတော့ဘဲ `ml` ကိုသုံးထားပါတယ်။ Bootstrap တို့ ဘာတို့မှာ သုံးနေကြပုံစံပဲ ဆက်သုံးရတာကို တွေ့ရမှာပါ။ `ml` ရှိသလို၊ `mr`, `mt`, `mb`, `my`, `mx` တွေလည်း လိုအပ်ရင် သုံးလို့ရမှာဖြစ်ပါတယ်။ တခြား `py`, `px`, `pl`, `pt`, `pb`, `pr` တွေအပါအဝင် သုံးနေကြ Bootstrap Utility Class တွေနဲ့ ပုံစံတူ လိုအပ်ရင် သုံးလို့ရမှာဖြစ်ပါတယ်။

နောက်ဆုံးက `Box` ထဲမှာ `IconButton` နှစ်ခုဆက်ရှိပါတယ်။ `Form` ကို အဖွင့်အပိတ်လုပ်လို့ရတဲ့ ခလုပ်နဲ့ Mode ပြောင်းလို့ရတဲ့ ခလုပ်နှစ်ခုပါ။ Mode ပြောင်းတဲ့အလုပ်ကို ထည့်မလုပ်သေးပါဘူး။ နောက်တစ်ခန်းကျမှ ဆက်လုပ်ကြပါမယ်။ Form အဖွင့်အပိတ်ကိုတော့ `useApp` ကနေရတဲ့ `setShowForm` ကိုသုံးပြီး လုပ်ပေးထားတာကို သတိပြုရမှာ ဖြစ်ပါတယ်။ စမ်းကြည့်ဖို့ မပြည့်စုံသေးပေမယ့် ဒီ Header ကနေရမှာက အခုလိုပုံစံ ဖြစ်ပါတယ်။



Icon တွေကို Import တဲ့အခါ သူ့မူရင်းနာမည်အတိုင်း `Menu`, `Add` စသည်ဖြင့် မလုပ်ဘဲ `MenuIcon`, `AddIcon` စသည်ဖြင့် နောက်ကနေ Icon လေးတွေထည့်ပြီး Import လုပ်တာကိုလည်း သတိပြုပါ။ ရိုးရိုး Component တွေနဲ့ ကွဲပြားသွားအောင်လို့ပါ။

ဆက်လက်ပြီး `Form.jsx` ကို အခုလိုပြင်ပေးပါ။

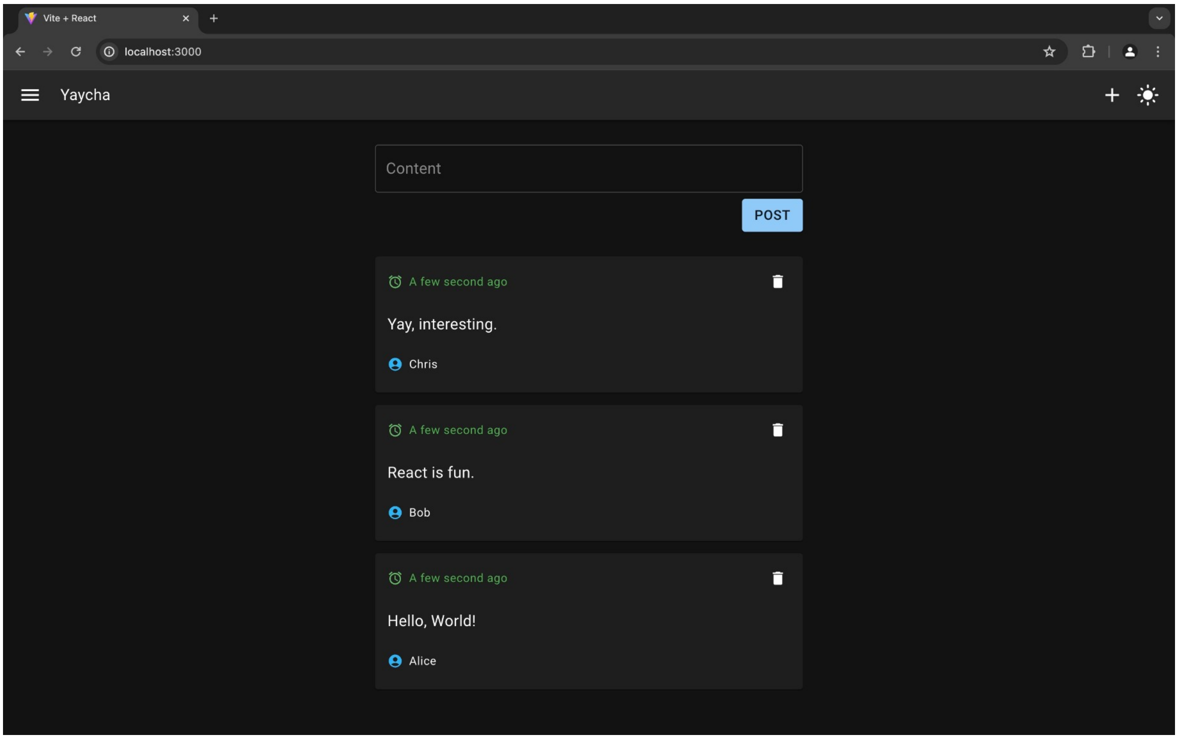
React - src/components/Form.jsx

```
import { useRef } from "react";

import { Box, TextField, Button } from "@mui/material";

export default function Form({ add }) {
  const contentRef = useRef();

  return (
    <form
```

```
      onSubmit={e => {
        e.preventDefault();
        const content = contentRef.current.value;

        add(content, "Alice");

        e.currentTarget.reset();
      }}>
      <Box sx={{ mb: 4, textAlign: "right" }}>
        <TextField
          inputRef={contentRef}
          type="text"
          placeholder="Content"
          fullWidth
          multiline
          sx={{ mb: 1 }}
        />
        <Button
          variant="contained"
          type="submit">
          Post
        </Button>
      </Box>
    </form>
  );
}
```

ထုံးစံအတိုင်း UI Component တွေပါလို့ ကုဒ်နည်းနည်းများပေမယ့် နားလည်ရအခက်ကြီး မဟုတ်ပါဘူး။ မူလက Input အနေနဲ့ content နဲ့ name နှစ်ခုပါပေမယ့် name ကို ဖယ်ထုတ်လိုက်ပါပြီ။ content တစ်ခုပဲ ပါတော့မှာပါ။ ဒါကြောင့် Ref လည်းနှစ်ခုမလိုတော့ပါဘူး။ တစ်ခုပဲလိုပါတော့တယ်။ <form> Submit မှာ add() ကိုခေါ်တဲ့အခါ name ကို အသေပေးပြီးခေါ်ထားတာကို သတိပြုပါ။ အရင်လို Input ရဲ့ Reference Value မဟုတ်တော့ပါဘူး။

ပြီးတဲ့အခါ HTML <input> နဲ့ <button> အစား MUI ရဲ့ <TextField> နဲ့ <Button> တို့ကို အစားထိုး အသုံးပြုထားပါတယ်။ <TextField> မှာ ref အစား inputRef လို့ရေးပေးရတာကို သတိပြုပါ။ <Button> မှာလည်း type="submit" ပါဖို့ မဖြစ်မနေ လိုအပ်သွားပါတယ်။ ဒီ Form ရဲ့ ဖော်ပြပုံက အခုလိုဖြစ်မှာပါ။



နောက်တစ်ဆင့်အနေနဲ့ Item.jsx ကို အခုလိုပြင်ပေးပါ။

React - src/components/Item.jsx

```
import {
  Box,
  Card,
  CardContent,
  Typography,
  IconButton,
} from "@mui/material";
```

```
import {
  Alarm as TimeIcon,
  AccountCircle as UserIcon,
  Delete as DeleteIcon,
} from "@mui/icons-material";

import { green } from "@mui/material/colors";

export default function Item({ item, remove }) {
  return (
    <Card sx={{ mb: 2 }}>
      <CardContent>
        <Box
          sx={{
            display: "flex",
            flexDirection: "row",
            justifyContent: "space-between",
          }}>
          <Box
            sx={{
              display: "flex",
              flexDirection: "row",
              alignItems: "center",
              gap: 1,
            }}>
            <TimeIcon
              fontSize="10"
              color="success"
            />
            <Typography
              variant="caption"
              sx={{ color: green[500] }}>
              A few second ago
            </Typography>
          </Box>
```

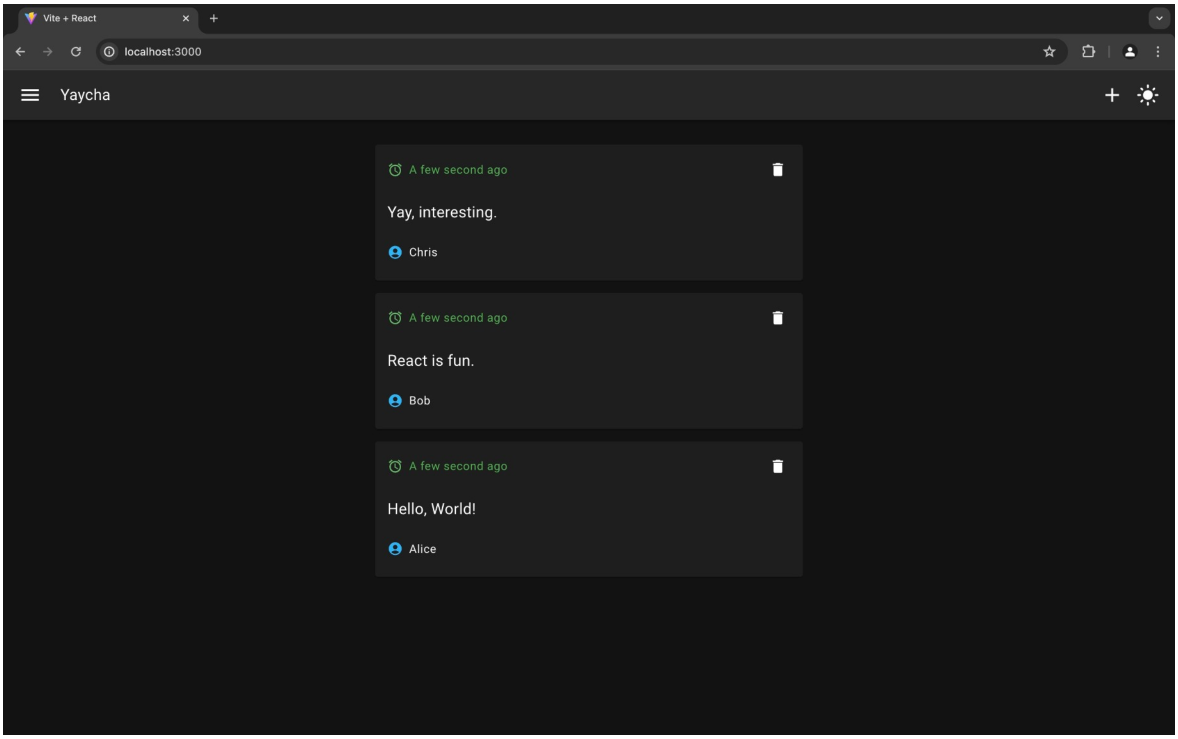
```
          <IconButton
            size="small"
            onClick={() => remove(item.id)}>
            <DeleteIcon fontSize="inherit" />
          </IconButton>
        </Box>

        <Typography sx={{ my: 3 }}>{item.content}</Typography>

        <Box
          sx={{
            display: "flex",
            flexDirection: "row",
            alignItems: "center",
            gap: 1,
          }}>
          <UserIcon
            fontSize="12"
            color="info"
          />
          <Typography variant="caption">{item.name}</Typography>
        </Box>
      </CardContent>
    </Card>
  );
}
```

`Item` တွေကို မူလက `<li>` နဲ့ ပြရာကနေ၊ အခု MUI ရဲ့ `<Card>` ကို အသုံးပြုပြီး ပြပေးတော့မှာ ဖြစ်ပါတယ်။ ထုံးစံအတိုင်း ကုဒ်တွေများပေမယ့် နားလည်ရမခက်လှပါဘူး။ ဒါကြောင့် ကုဒ်ကိုပဲ တိုက်ရိုက်ဖတ်ကြည့်လိုက်ပါ။ ထူးခြားချက်တစ်ခုကတော့ MUI မှာ `color` တွေ Import လုပ်ပြီး သုံးလို့ရခြင်းပဲ ဖြစ်ပါတယ်။ နမူနာမှာ `green` ကို Import လုပ်ပြီးသုံးထားပါတယ်။ တခြားသုံးလို့ရတဲ့ `color` တွေ Icons တွေကို ဒီလင့်တွေမှာ လေ့လာကြည့်နိုင်ပါတယ်။

Colors – https://mui.com/material-ui/customization/color/

Icons – https://mui.com/material-ui/material-icons/

ဒီ `Item` ရဲ့ဖော်ပြပုံက အခုလိုဖြစ်မှာပါ။



Component တွေစုံပြီဖြစ်လို့ နောက်ဆုံးအနေနဲ့ `App.jsx` ကို အခုလိုပြင်ပေးလိုက်ပါ။

React - src/App.jsx

```
import { useState } from "react";

import { Box, Container } from "@mui/material";

import Header from "./components/Header";
import Form from "./components/Form";
import Item from "./components/Item";
```

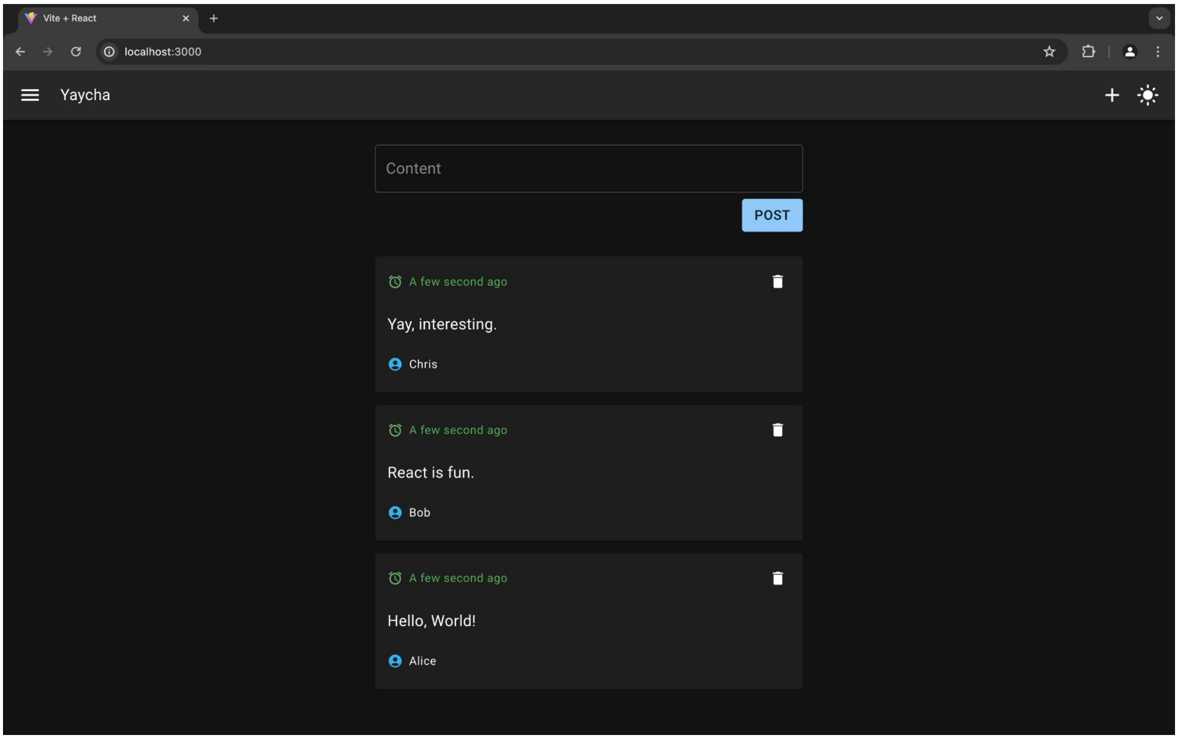
```
import { useApp } from "./ThemedApp";

export default function App() {
  const { showForm } = useApp();

  const [data, setData] = useState([
    { id: 3, content: "Yay, interesting.", name: "Chris" },
    { id: 2, content: "React is fun.", name: "Bob" },
    { id: 1, content: "Hello, World!", name: "Alice" },
  ]);

  const remove = id => {
    setData(data.filter(item => item.id !== id));
  };

  const add = (content, name) => {
    const id = data[0].id + 1;
    setData([{ id, content, name }, ...data]);
  };

  return (
    <Box>
      <Header />

      <Container
        maxWidth="sm"
        sx={{ mt: 4 }}>
        {showForm && <Form add={add} />}

        {data.map(item => {
          return (
            <Item
              key={item.id}
              item={item}
              remove={remove}
```

```
            />
          );
        })}
      </Container>
    </Box>
  );
}
```

ထူးခြားချက်လေးတွေပြောပြပါမယ်။ `Header, Form, Item` စတဲ့ Component တွေ Import လုပ်ပုံကို သတိပြုပါ။ ဖိုဒါခွဲရေးထားလို့ မှန်အောင် Import လုပ်ပေးရပါမယ်။ `useApp` နဲ့ `showForm` ကို ယူထားတာသတိပြုပါ။ State Data မှာ `name` တွေပါသွားပြီး `id` တွေကို `3, 2, 1` ပြောင်းပြန်စီတာကိုလည်း သတိပြုပါ။ အသစ်ထပ်ထည့်တဲ့ `Item` ကို အရင်ပြစေချင်လို့ပါ။ ပြီးတော့ အရင်ထည့်ထားတဲ့ `<h1>` Element ကိုလည်း မလိုအပ်တော့လို့ ဖယ်ထုတ်လိုက်ပါပြီ။ သူ့အစား `<Header>` ကို အသုံးပြုလိုက်တာပါ။

ကျန်ကုဒ်တွေကတော့ အများကြီးထူးခြားတာမရှိတော့ဘဲ `<Container>` Component ထဲမှာ `<Item>` တွေကို ပြလိုက်တာပဲဖြစ်ပါတယ်။ `<Container>` ရဲ့ `maxWidth` မှာ `sm` လို့ သတ်မှတ်ပေးထားတာကို သတိပြုပါ။ ဒါလည်းပဲ Bootstrap မှာလို `md` တွေ `lg` တွေ အစားထိုးသုံးချင်ရင် သုံးလို့ရနိုင်ပါတယ်။

လိုအပ်တာတွေ စုံပြီဖြစ်လို့ `npm run dev` နဲ့ စမ်းကြည့်လိုက်ရင်ရမယ့် ရလဒ်က အခုလိုဖြစ်မှာပါ။



အများကြီး သပ်သပ်ရပ်ရပ်ဖြစ်သွားပါပြီ။ လက်ရှိ Default အနေနဲ့ `dark` Mode ကိုသုံးပြီးပြထားပါတယ်။ စမ်းချင်ရင် `ThemedApp` မှာ `dark` အစား `light` လို့ပြောင်းပေးပြီး စမ်းလို့ရပါတယ်။ နောက်တစ်ခန်းမှာ လေ့လာသင့်တာလေးတွေ ဖြည့်စွက်လေ့လာပြီး Mode ပြောင်းတဲ့ခလုပ်ကို ပြန်အသက်သွင်းကြပါမယ်။

အခန်း (၇) - React - More Hooks

React Component တွေ ဖန်တီးဖို့အတွက် ရိုးရိုး Function တွေကိုပဲ အသုံးပြုရတာ တွေ့ရှိခဲ့ကြပြီး ဖြစ်ပါတယ်။ အဲ့ဒီလို Function Component တွေမတိုင်ခင်က Class တွေကို သုံးခဲ့ကြရပါတယ်။ Class ကို အသုံးပြုဖန်တီးတဲ့ Component တွေမှာ React ရဲ့လုပ်ဆောင်ချက်တွေရရှိဖို့အတွက် Inheritance လို လုပ်ဆောင်ချက်မျိုးကို အသုံးပြုနိုင်ပါတယ်။ ဥပမာ –

React - Class Component

```
class List extends React.Component {
  ...
}
```

ဒီနည်းနဲ့ `List` Component က `React.Component` ရဲ့ လုပ်ဆောင်ချက်တွေကို ဆက်ခံရရှိသွားပါတယ်။ Function Component တွေမှာ ဒီလို Inheritance မပါဘဲ React ရဲ့လုပ်ဆောင်ချက်တွေကို ရရှိဖို့ လိုအပ်လာပါတယ်။ ဒါကြောင့် React က `useState`, `useRef`, `useContext` တို့လို အသုံးဝင်တဲ့ Hook Functions တွေကို ထည့်သွင်းပေးခဲ့ပါတယ်။ သူတို့ရဲ့ အလုပ်လုပ်ပုံကို တွေ့ရှိခဲ့ကြပြီး ဖြစ်ပါတယ်။

ဒီ Hook Functions တွေဟာ အခြေခံအားဖြင့် သူတို့ကို ခေါ်လိုက်တဲ့ Function ရဲ့လုပ်ဆောင်ချက်ကို ဖြည့်စွက်ပေးလိုက်တာ ဖြစ်ပါတယ်။ ဥပမာ – `useState` ကို ခေါ်လိုက်တဲ့အခါ React က State Queue ထဲမှာ State Date နဲ့ Component ကို တွဲမှတ်ပေးလိုက်ပါတယ်။ ဒါကြောင့် State Data ပြောင်းတဲ့အခါ Component ကို အလိုအလျောက် Re-Render လုပ်ပေးနိုင်သွားပါတယ်။

နောက်ကွယ်မှာ အလုပ်လုပ်သွားတဲ့ ကိစ္စတွေဖြစ်လို့ မြင်ဖို့ခက်နေနိုင်ပါတယ်။ ဒါတွေကို ကောင်းကောင်း နားလည်ဖို့ဆိုရင် Higher Order Function တို့ Function Composition တို့အကြောင်းကို ကြားဖြတ်ပြော ဖို့လိုပါလိမ့်မယ်။ ထည့်မပြောနိုင်တဲ့အတွက် ဒီလိုအလွယ်မှတ်ထားပေးပါ။ Hook Functions တွေကို ခေါ် သုံးလိုက်တဲ့အခါ ခေါ်သုံးလိုက်တဲ့ Function ရဲ့ အလုပ်လုပ်ပုံမှာ React ရဲ့ လုပ်ဆောင်ချက်တွေ ပေါင်းစပ် ပါဝင်သွားတဲ့ သဘောပဲဖြစ်ပါတယ်။

ဒီအခန်းမှာ `useMemo` နဲ့ `useEffect` ခေါ် နောက်ထပ် အသုံးဝင်တဲ့ Hook Function (၂) ခုအကြောင်း ကို လေ့လာသွားပါမယ်။ လေ့လာသူအများစု ဘယ်လိုပဲလေ့လာလေ့လာ ကောင်းကောင်းနားမလည်ဘဲ စိတ်ညစ်ကြရတဲ့ Function နှစ်ခုပါ။ တကယ်တော့ သိပ်အခက်ကြီးမဟုတ်ပါဘူး။ ဥပမာ – ဒီကုဒ်ကို လေ့လာကြည့်ပါ။

React

```
import { useState } from "react";

function expensive() {
  console.log("Some expensive tasks");
  return "data";
}

function App() {
  const [count, setCount] = useState(0);

  const data = expensive();

  return <div>
    <h1>Count: {count}</h1>
    <button onClick={() => setCount(count + 1)}>
      Button
    </button>
  </div>;
}
```

Data ကို Return ပြန်ပေးတဲ့ expensive Function တစ်ခုရှိနေပြီး App Component ကအဲ့ဒီ Function ကို ခေါ်ထားပါတယ်။ ဒါကြောင့် Component Re-Render လုပ်တိုင်း expensive Function အလုပ် လုပ်မှာ ဖြစ်ပါတယ်။ အဲ့ဒီလို Component Re-Render လုပ်တိုင်း အလုပ်မလုပ်စေချင်ဘူး၊ လိုအပ်မှပဲ လုပ် စေချင်တယ်ဆိုရင် useMemo ကို အသုံးပြုနိုင်ပါတယ်။ ဒီလိုပါ –

React

```
import { useState, useMemo } from "react";

function expensive() {
  console.log("Some expensive tasks");
  return "data";
}

function App() {
  const [count, setCount] = useState(0);

  const data = useMemo(() => {
    return expensive();
  }, []);

  return <div>
    <h1>Count: {count}</h1>
    <button onClick={() => setCount(count + 1)}>
      Button
    </button>
  </div>;
}
```

useMemo က Function တစ်ခုနဲ့ Dependencies Array တစ်ခုတို့ကို လက်ခံပါတယ်။ နမူနာမှာ expensive Function ကို useMemo Function ထဲမှာ ခေါ်ထားပါတယ်။ ဒါကြောင့် <u>expensive Function ဟာ တစ်ကြိမ်ပဲအလုပ်လုပ်မှာဖြစ်ပြီး Component ကို Re-Render လုပ်တဲ့အခါ ထပ်ခါထပ်ခါ အလုပ်မလုပ်တော့ပါဘူး။ ဒါပေမယ့် ရလဒ် data ကိုတော့ မှတ်ထားပေးမှာပါ</u>။ Re-Render ပြန်လုပ်တဲ့

အချိန်မှာ မှတ်ထားတဲ့ `data` ကို ဆက်အသုံးပြုလို့ရနေမှာ ဖြစ်ပါတယ်။

အလုပ်တွေအများကြီးလုပ်ရတဲ့ Function တွေကို ဒီအတိုင်းခေါ်သုံးရင် မလိုအပ်ဘဲ ထပ်ခါထပ်ခါ Run နေတာမျိုးဖြစ်ပြီး နှေးတတ်ပါတယ်။ `useMemo` က တစ်ကြိမ်ပဲ Run ပြီး ရလဒ်ကိုမှတ်ထားပေးလို့ ပိုမြန်သွားမှာပဲဖြစ်ပါတယ်။

လက်ရှိနမူနာမှာ `useMemo` ရဲ့ ဒုတိယ Parameter အဖြစ် Array အလွတ်တစ်ခုကို ပေးထားပါတယ်။ Dependencies Array လို့ ခေါ်ပါတယ်။ Dependencies Array ထဲက တန်ဖိုးတွေပြောင်းရင် `useMemo` က `expensive` Function ကို ပြန်အလုပ်လုပ်ပေးမှာပါ။ Dependencies Array မပါရင် `useMemo` က ဒီ `expensive` Function ကို ထပ်ခါထပ်ခါ ပြန် Run နေမှာပါ။ ဒါကြောင့် အနည်းဆုံး Dependencies Array အလွတ်တော့ ပါဖို့ လိုပါတယ်။

`useEffect` ရဲ့အလုပ်လုပ်ပုံက နည်းနည်းဆင်တူပေမယ့် ကွဲပြားသွားပါတယ်။ `useEffect` ကလည်း အလုပ်လုပ်ရမယ့် Function တစ်ခုနဲ့ Dependencies Array တစ်ခုကို လက်ခံပါတယ်။ ပေးလိုက်တဲ့ Function ကို Component ရဲ့ ပထမဆုံးအကြိမ် Render လုပ်ချိန်မှာ တစ်ကြိမ်ပဲ Run ပေးပါလိမ့်မယ်။ Component ကို Re-Render လုပ်တိုင်း ထပ်ခါထပ်ခါ မ Run တော့ပါဘူး။ ဒီလိုပါ –

React

```
import { useState, useEffect } from "react";

function getData() {
  console.log("Getting data...");
  return "data";
}

function App() {
  const [data, setData] = useState("");
  useEffect(() => {
    const items = getData();
    setData(items);
  }, []);
```

```
  return <div>
    <h1>Data: {data}</h1>
  </div>;
}
```

နမူနာအရ `getData()` ကို Database ထဲကနေ Data ရယူတဲ့ Function လို့သဘောထားပါ။ သို့မဟုတ် API Service တစ်ခုခုကနေ Data လှမ်းယူတာလို့ သဘောထားရင်လည်း ရပါတယ်။ Component ကို Re-Render လုပ်တိုင်း ထပ်ခါထပ်ခါ သွားယူနေရင် အဆင်မပြေပါဘူး။ Read Only ယူတာဆိုရင် တော်သေးတယ်။ သွားပြင်တာဆိုရင် Re-Render လုပ်တိုင်း ထပ်ခါထပ်ခါသွားပြင်နေရင် အဆင်မပြေပါဘူး။ ဒီကိစ္စမျိုးကို Side-Effect လို့ခေါ်ပါတယ်။ အဲ့ဒီလို တခြားနေရာမှာ သွားပြီး သက်ရောက်မှုရှိနေတဲ့ Side-Effect လုပ်ဆောင်ချက်တွေကို ထပ်ခါထပ်ခါ အလုပ်မလုပ်သင့်ပါဘူး။ `useEffect` နဲ့ ရေးပေးရပါတယ်။

`useEffect` က မူအားဖြင့် ပထမဆုံးအကြိမ် Render လုပ်စဉ် (၁) ကြိမ်ပဲအလုပ်လုပ်ရမှာပါ။ ဒါပေမယ့် React Development Mode မှာ နှစ်ကြိမ် အလုပ်လုပ်ပါလိမ့်မယ်။ နှစ်ကြိမ်ရတဲ့ ရလဒ် (၂) ခုကို တိုက်စစ်ပြီး တူမတူစစ်ချင်ရင် စစ်လို့ရအောင်လို့ Run ပေးတာပါ။ သူလည်းပဲ Dependencies Array ထဲမှာပေးထားတဲ့ တန်ဖိုးပြောင်းရင်တော့ နောက်တစ်ကြိမ် ပြန်ခေါ်ပေးမှာဖြစ်ပါတယ်။ အနည်းဆုံး Dependencies Array အလွတ် မဖြစ်မနေပါဖို့ လိုအပ်ပါတယ်။

ဒီ Hook Functions နှစ်ခုကို လေ့လာသူတွေသာမက အတွေ့အကြုံရင့် Developer တွေပါ စိတ်ညစ်ကြပါတယ်။ ကောင်းကောင်းနားမလည်လို့ စိတ်ညစ်ရတာ ပါသလို၊ လက်တွေ့အသုံးပြုတဲ့အခါ မထင်မှတ်တဲ့ ပြဿနာတွေ ကြုံရတတ်လို့လည်း ပါပါတယ်။ ဒါကြောင့် နောက်ပိုင်းမှာ React က React Compiler လို နည်းပညာမျိုး ထပ်မံတီထွင်ထားပါတယ်။ `useMemo` သုံးစရာမလိုဘဲ အလားတူ လုပ်ဆောင်ချက်ကို အလိုအလျှောက် လုပ်ပေးနိုင်ပါတယ်။ ဒီစာရေးချိန်မှာ ထွက်ပေါ်ခါစ စမ်းသပ်အဆင့်ပဲ ရှိသေးလို့ ဒီအကြောင်းတော့ ထည့်ပြောနိုင်ဦးမှာ မဟုတ်ပါဘူး။

အလားတူပဲ `useEffect` ကို Data Fetching အတွက် အသုံးများကြတာမှာ အဆင်မပြေတဲ့ အခက်အခဲတွေ ရှိတတ်လို့ `useEffect` ကို သုံးမယ့်အစား React Query တို့ SWR တို့လို Third-Party Data Fetching Library တွေကို သုံးကြဖို့ အကြံပြုကြပါတယ်။ ဒီစာရေးနေချိန်မှာ Route Provider မှာ Data

Fetch လုပ်တာက ပိုကောင်းတယ်လို့ ဆိုနေကြပြန်ပါတယ်။ ဒီလိုပါပဲ နည်းပညာတွေက အမြဲပြောင်းနေတာပါ။ ဒါကြောင့် အမျိုးမျိုးပြောင်းနေတဲ့ Abstraction တွေအစား မူရင်းအခြေခံ ရေးနည်းတွေကိုပဲ အဓိကထား လေ့လာကြရမှာဖြစ်ပါတယ်။ အခြေခံပိုင်ရင် Abstraction က လိုတော့မှ လေ့လာအသုံးပြုလို့ ရနိုင်မှာဖြစ်ပါတယ်။

Mode Switcher

အခုလေ့လာလိုက်တဲ့ထဲက `useMemo` ကိုအသုံးပြုပြီး လုပ်လက်စပရောဂျက်မှာ MUI နဲ့ Light Mode, Dark Mode ပြောင်းလို့ရတဲ့ လုပ်ဆောင်ချက်နဲ့အတူ တခြားလုပ်ဆောင်ချက်တွေ ထပ်ဖြည့်ကြပါမယ်။ ပထမဆုံးအနေနဲ့ ရေးလက်စ `ThemedApp.jsx` ကုဒ်ကို အခုလို ပြင်ရပါမယ်။

React - src/ThemedApp.jsx

```
export default function ThemedApp() {
  const [showForm, setShowForm] = useState(false);
  const [mode, setMode] = useState("dark");

  const theme = createTheme({
    palette: { mode },
  });

  return (
    <ThemeProvider theme={theme}>
      <AppContext.Provider
        value={{ showForm, setShowForm, mode, setMode, }}>
        <App />
        <CssBaseline />
      </AppContext.Provider>
    </ThemeProvider>
  );
}
```

မူလက Component အပြင်မှာရေးထားတဲ့ `createTheme` လုပ်ဆောင်ချက်ကို Component ထဲကို ရွှေ့လိုက်ပါတယ်။ State Data နဲ့ ချိတ်တော့မှာဖြစ်လို့ အပြင်မှာရေးလို့မရတော့ပါဘူး။ Component ထဲမှာ

ရေးပေးရမှာပါ။ ပြီးတဲ့အခါ `mode` နဲ့ `setMode` ကို `useState` နဲ့ကြေညာလိုက်ပါတယ်။ Default Value အဖြစ် `dark` ကို သတ်မှတ်ထားပြီး `createTheme` မှာ အဲ့ဒီ `mode` State ကို ယူသုံးလိုက်တာပါ။ ဒါကြောင့် `mode` ပြောင်းရင် Theme လိုက်ပြောင်းမှာဖြစ်ပါတယ်။ `mode` ပြောင်းတဲ့အလုပ်ကို လိုတဲ့နေရာက ယူလို့ရအောင် `AppContext` Provider မှာလည်း ထည့်ပေးထားပါတယ်။

ဒီကုဒ်က အလုပ်လုပ်ပါတယ်။ ဒါပေမယ့် ပြဿနာက `createTheme` ကို Component Re-Render လုပ်တိုင်း အမြဲပြန် Run နေမှာပါ။ အဲ့ဒီလို မ Run စေချင်ပါဘူး။ လိုအပ်မှပဲ Run စေချင်ပါတယ်။ ဒါကြောင့် `createTheme` ကို `useMemo` နဲ့ အခုလိုပြောင်းရေးပေးဖို့လိုပါတယ်။

React - src/ThemedApp.jsx

```
const theme = useMemo(() => {
  return createTheme({
    palette: { mode },
  });
}, [mode]);
```

`useMemo` Function ထဲမှာ ခေါ်ထားတဲ့ `createTheme` ကို ပထမဆုံးတစ်ကြိမ်ပဲ Run တော့မှာပါ။ `theme` Value ကို `useMemo` က မှတ်ထားပေးမှာဖြစ်လို့ ထပ်ခါထပ်ခါ Run စရာမလိုတော့ဘဲ အသုံးပြုလို့ရသွားပါပြီ။ Dependencies Array ထဲမှာ `mode` ကို ထည့်ပေးထားပါတယ်။ ဒါကြောင့် `mode` Value ပြောင်းရင်တော့ `createTheme` ကို ထပ် Run ပေးပါလိမ့်မယ်။ ပြောင်းသွားတဲ့ `mode` Option တွေနဲ့ `createTheme` ကိုပြန် Run ဖို့လည်း လိုအပ်ပါတယ်။ ဒါကြောင့် ဒီကုဒ်ဟာ လိုတဲ့အချိန်ပဲ Run ပြီး မလိုတဲ့အချိန်တွေမှာ မ Run တော့လို့ ပိုကောင်းသွားပါပြီ။

`mode` ကို State နဲ့လည်း ချိတ်ပြီးပြီ။ `AppContext` Provider မှာလည်း `mode` နဲ့ `setMode` ကို ထည့်ပေးပြီးပြီဆိုရင် `Header.jsx` မှာ အခုလို ပြင်ပေးလိုက်လို့ ရပါပြီ။

React - src/components/Header.jsx

```
// Other imports

import {
  ...
  DarkMode as DarkModeIcon,
} from "@mui/icons-material";

export default function Header() {
  const { showForm, setShowForm, mode, setMode } = useApp();

  return (
    <AppBar position="static">
      <Toolbar>
        ...

        <Box>
          ...

          {mode === "dark" ? (
            <IconButton
              color="inherit"
              edge="end"
              onClick={() => setMode("light")}>
              <LightModeIcon />
            </IconButton>
          ) : (
            <IconButton
              color="inherit"
              edge="end"
              onClick={() => setMode("dark")}>
              <DarkModeIcon />
            </IconButton>
          )}
        </Box>
```

```
      </Toolbar>
    </AppBar>
  );
}
```

MUI စခါစက ခဏပြန်ပြတ်ထားတဲ့ Light Mode, Dark Mode ပြောင်းတဲ့ခလုပ်ကို ပြန်အသက်သွင်းလိုက်တာပါ။ Mode ပြောင်းပြီး စမ်းကြည့်လို့ရပါပြီ။

Other UI Changes

တစ်လက်စထဲ `createTheme` အကြောင်းပြောလက်စနဲ့၊ တခြား Theme Option တွေ ထပ်ထည့်ကြပါမယ်။ ဒီလိုပါ –

React - src/ThemedApp.jsx

```
const theme = useMemo(() => {
  return createTheme({
    palette: {
      mode,
      primary: deepPurple,
      banner: mode === "dark" ? grey[800] : grey[200],
      text: {
        fade: grey[500],
      },
    },
  });
}, [mode]);
```

`mode` က State ကိုသုံးထားတာဖြစ်လို့ လက်ရှိ `mode: "dark"` ဖြစ်မယ်ဆိုတာ သတိပြုပါ။ ပြီးတဲ့အခါ `primary` အတွက် `deepPurple` လို့ပြောထားပါတယ်။ `deepPurple` နဲ့ တခြားလိုအပ်တဲ့ `colors` တွေကို Import လုပ်ပေးဖို့လိုပါတယ်။ ဒီလိုသတ်မှတ်လိုက်တဲ့အတွက် App တစ်ခုလုံးရဲ့ Main Color Themes အတွက် `primary` Color ဟာ `deepPurple` ဖြစ်သွားပါပြီ။ မူလက `blue` ဖြစ်ပါလိမ့်မယ်။

လိုအပ်လို့ မဟုတ်ပါဘူး။ အခုလိုပြောင်းလို့ရတယ်ဆိုတာ နမူနာရအောင် ထည့်ပြောင်းလိုက်တာပါ။ ပြီးတဲ့အခါ `banner` အမည်နဲ့ `mode` ပေါ်မူတည်ပြီး အရောင်သတ်မှတ်ထားပါတယ်။ လိုတဲ့အခါ အခုလို ယူသုံးလို့ရသွားပါတယ်။

React

```
<Box sx={{ bgcolor: "banner" }}></Box>
```

ကိုယ့်ဘာသာ Custom Color Name သတ်မှတ်လိုက်သလို ဖြစ်သွားတာပါ။ အသုံးဝင်ပါတယ်။ ဆက်လက် သတ်မှတ်ထားတဲ့ `text` အမည်နဲ့ Custom Color ကို လိုအပ်ရင် အခုလို ယူသုံးရပါတယ်။

React

```
<Typography sx={{ color: "text.fade" }}></Typography>
```

ဒါလည်းပဲမဖြစ်မနေ လိုအပ်လို့မဟုတ်ပါဘူး။ MUI Theme မှာ အခုလို Custom Color တွေ သတ်မှတ်ထားပြီး ပြန်ယူသုံးလို့ရတယ်ဆိုတာ နမူနာရစေဖို့အတွက် ထည့်ပေးထားတာပါ။

`Header.jsx` မှာ Menu Icon Button လေးတစ်ခုလည်း ထည့်ခဲ့ကြပါတယ်။ Drawer Menu လေးတစ်ခု လည်း ထည့်ချင်လို့ပါ။ အခုတစ်ခါထဲ ထည့်လိုက်ကြပါမယ်။ `components/AppDrawer.jsx` အမည်နဲ့ ဖိုင်အသစ်တစ်ခု တည်ဆောက်ပြီး ဒီကုဒ်ကို ကူးထည့်ပေးလိုက်ပါ။

React - src/components/AppDrawer.jsx

```
import {
  Box,
  Drawer,
  Divider,
  List,
  ListItem,
  ListItemButton,
  ListItemIcon,
  ListItemText,
```

```
  Avatar,
  Typography,
} from "@mui/material";

import {
  Home as HomeIcon,
  Person as ProfileIcon,
  Logout as LogoutIcon,
  PersonAdd as RegisterIcon,
  Login as LoginIcon,
} from "@mui/icons-material";

import { deepPurple } from "@mui/material/colors";

import { useApp } from "../ThemedApp";

export default function AppDrawer() {
  const { showDrawer, setShowDrawer, auth, setAuth } = useApp();

  return (
    <div>
      <Drawer
        open={showDrawer}
        onClose={() => setShowDrawer(false)}>
        <Box
          sx={{
            mb: 6,
            width: 300,
            height: 140,
            bgcolor: "banner",
            position: "relative",
          }}>
          <Box
            sx={{
              gap: 2,
```

```
            display: "flex",
            flexDirection: "row",
            alignItems: "center",
            position: "absolute",
            left: 20,
            bottom: -30,
          }}>
          <Avatar
            sx={{
              width: 94,
              height: 94,
              color: "white",
              background: deepPurple[500],
            }}
          />
          <Typography sx={{ fontWeight: "bold" }}>
            Alice
          </Typography>
        </Box>
      </Box>
      <List>
        <ListItem>
          <ListItemButton>
            <ListItemIcon>
              <HomeIcon />
            </ListItemIcon>
            <ListItemText>Home</ListItemText>
          </ListItemButton>
        </ListItem>
        <Divider />

        {auth && (
          <>
            <ListItem>
              <ListItemButton>
```

```
              <ListItemIcon>
                <ProfileIcon />
              </ListItemIcon>
              <ListItemText>Profile</ListItemText>
            </ListItemButton>
          </ListItem>
          <ListItem>
            <ListItemButton onClick={() => setAuth(null)}>
              <ListItemIcon>
                <LogoutIcon color="error" />
              </ListItemIcon>
              <ListItemText>Logout</ListItemText>
            </ListItemButton>
          </ListItem>
        </>
      )}

      {!auth && (
        <>
          <ListItem>
            <ListItemButton>
              <ListItemIcon>
                <RegisterIcon />
              </ListItemIcon>
              <ListItemText>Register</ListItemText>
            </ListItemButton>
          </ListItem>
          <ListItem>
            <ListItemButton onClick={() => setAuth(true)}>
              <ListItemIcon>
                <LoginIcon />
              </ListItemIcon>
              <ListItemText>Login</ListItemText>
            </ListItemButton>
          </ListItem>
```

```
            </>
          )}
        </List>
      </Drawer>
    </div>
  );
}
```

ပါတဲ့ကုဒ်တွေ အတော်များပေမယ့် နားလည်ရခက်နိုင်တာတော့ မပါပါဘူး။ ကုဒ်တွေကို လိုက်ဖတ်ကြည့် စေချင်ပါတယ်။ MUI ရဲ့ Drawer Component ကို သုံးလိုက်တာပါ။ ထူးခြားချက်အနေနဲ့ showDrawer, setShowDrawer, auth နဲ့ setAuth တို့ကို useApp() ကနေ ယူသုံးထားတာကို သတိပြုပါ။ ဒါကြောင့် ThemedApp မှာ အဲ့ဒီ State တွေ သွားထည့်ဖို့လိုပါလိမ့်မယ်။

Drawer အဖွင့်အပိတ်ကို open Props နဲ့ ထိမ်းထားတာပါ။ ဒါကြောင့် showDrawer က true ဖြစ်နေရင် Drawer ကိုပြမှာဖြစ်ပြီး false ဖြစ်နေရင် ဖျောက်ထားပေးမှာပါ။ onClose မှာ setShowDrawer(false) နဲ့ ပိတ်တဲ့အချိန် Drawer ကို ပြန်ဖျောက်ခိုင်းထားပါတယ်။

ပြီးတဲ့အခါ စောစောက ကြေညာခဲ့တဲ့ banner Color ကိုသုံးပြီး Box တစ်ခုနဲ့ အဲ့ဒီ Box ထဲမှာ Avatar တစ်ခု Typography တစ်ခုထည့်ထားပါတယ်။ လောလောဆယ် အသေပါပဲ။ နောက်မှ Login ဝင်ထားတဲ့ User ရဲ့ အချက်အလက်အမှန်နဲ့ အစားထိုးပြကြပါမယ်။

Banner Box ရဲ့အောက်မှာ List တစ်ခုနဲ့ Home, Register, Login, Profile, Logout ခလုပ်တွေ တန်းစီပြီးထည့်ထားပါတယ်။ Register နဲ့ Login ကို auth က false ဖြစ်မှပြမှာပါ။ Profile နဲ့ Logout ကိုတော့ auth က true ဖြစ်နေမှ ပြမှာပါ။ စမ်းလို့ရအောင် Login မှာ setAuth(true) လုပ်ပေးထားပြီး Logout မှာ setAuth(false) လုပ်ပေးထားပါတယ်။

ဒီ AppDrawer Component ကို ThemedApp မှာ ထည့်သုံးမှာပါ။ AppDrawer ပါဝင်ပြီး ThemedApp.jsx ရဲ့ ကုဒ်အပြည့်အစုံကို ဖော်ပြပေးလိုက်ပါတယ်။ ကူးယူထည့်သွင်းလိုက်ပါ။

React - src/ThemedApp.jsx

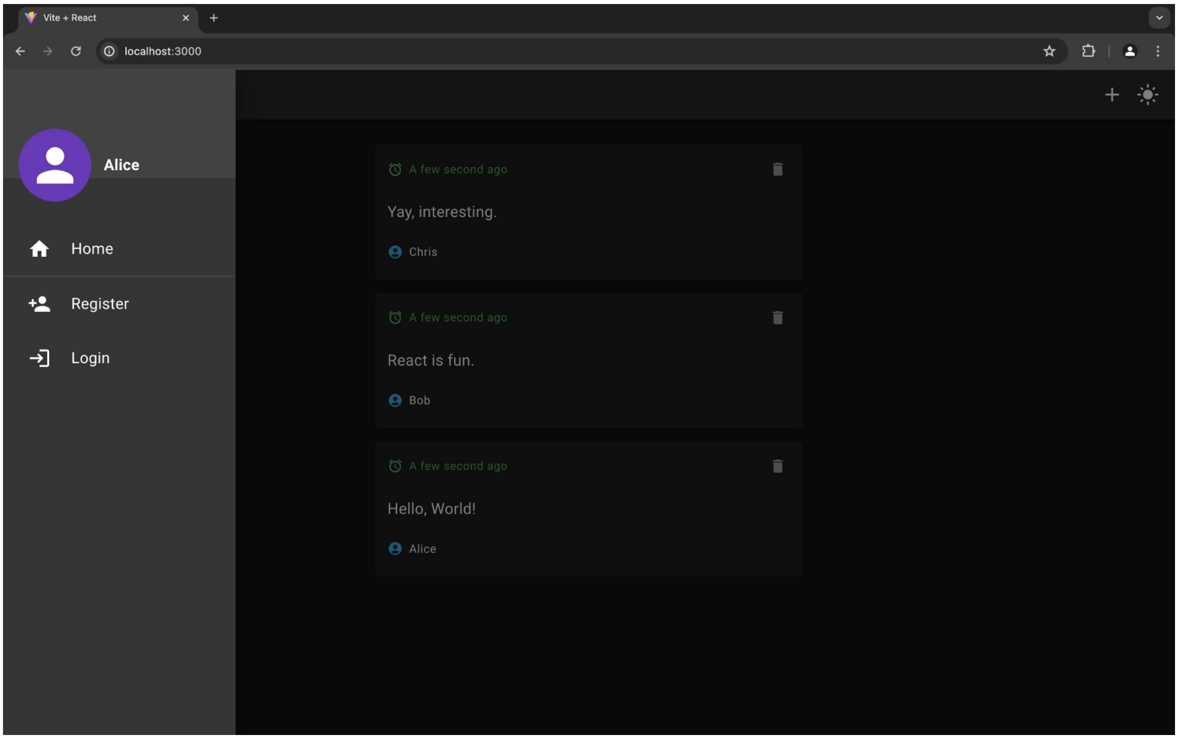
```
import { useState, createContext, useContext, useMemo } from "react";

import {
  CssBaseline,
  ThemeProvider,
  createTheme,
  Snackbar,
} from "@mui/material";

import App from "./App";
import AppDrawer from "./components/AppDrawer";

import { deepPurple, grey } from "@mui/material/colors";

const AppContext = createContext();

export function useApp() {
  return useContext(AppContext);
}

export default function ThemedApp() {
  const [showDrawer, setShowDrawer] = useState(false);
  const [showForm, setShowForm] = useState(false);
  const [globalMsg, setGlobalMsg] = useState(null);
  const [auth, setAuth] = useState(null);
  const [mode, setMode] = useState("dark");

  const theme = useMemo(() => {
    return createTheme({
      palette: {
        mode,
        primary: deepPurple,
        banner: mode === "dark" ? grey[800] : grey[200],
        text: {
```

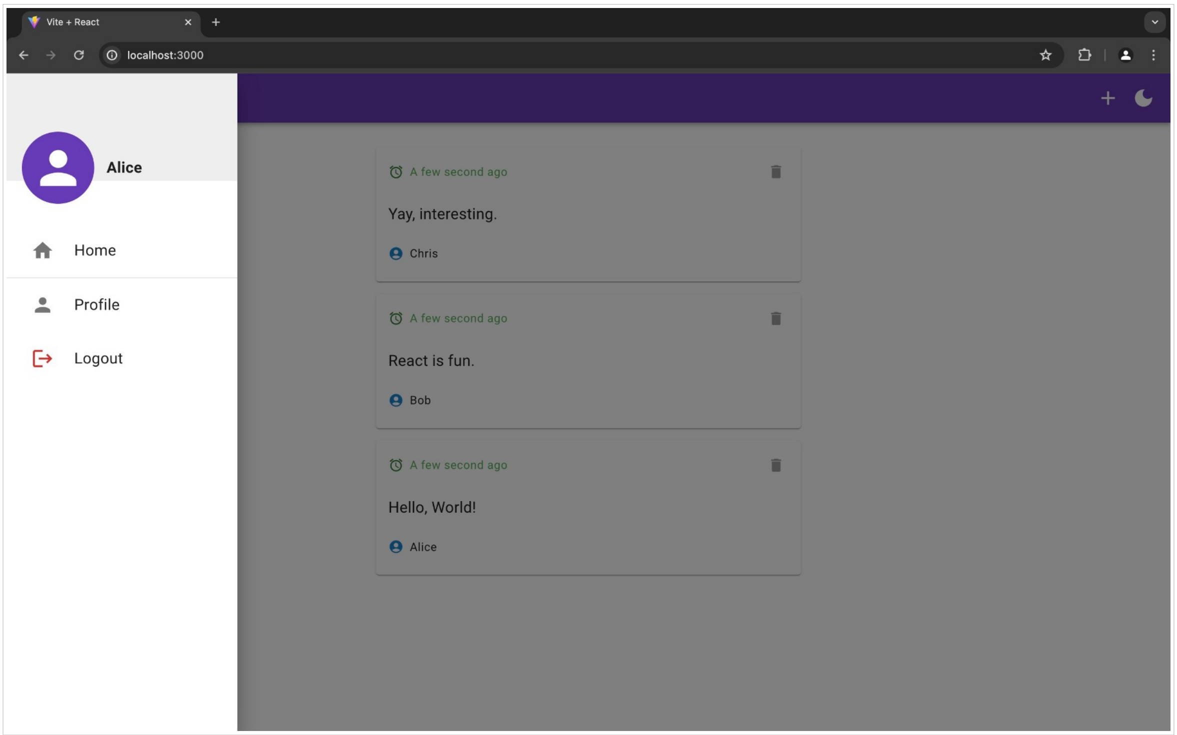
```
          fade: grey[500],
        },
      },
    });
  }, [mode]);

  return (
    <ThemeProvider theme={theme}>
      <AppContext.Provider
        value={{
          showDrawer,
          setShowDrawer,
          showForm,
          setShowForm,
          globalMsg,
          setGlobalMsg,
          auth,
          setAuth,
          mode,
          setMode,
        }}>
        <App />
        <AppDrawer />

        <Snackbar
          anchorOrigin={{
            horizontal: "center",
            vertical: "bottom",
          }}
          open={Boolean(globalMsg)}
          autoHideDuration={6000}
          onClose={() => setGlobalMsg(null)}
          message={globalMsg}
        />
```

```
        <CssBaseline />
      </AppContext.Provider>
    </ThemeProvider>
  );
}
```

showDrawer, setShowDrawer, auth, setAuth တွေကို useState နဲ့ ကြေညာပြီး Context Value မှာ ထည့်ပေးထားပါတယ်။ တစ်လက်စထဲ Snackbar Component နဲ့ Global Message တွေ ပြနိုင်းထားပါတယ်။ အဲ့ဒီ Global Message တွေကို ကြိုက်တဲ့နေရာကနေ Manage လုပ်လို့ရအောင် globalMsg နဲ့ setGlobalMsg တို့ကိုလည်း တစ်ပါထဲ ထည့်ကြေညာချိတ်ဆက်ပေးထားတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။

စမ်းကြည့်ရင်ရမယ့် ရလဒ်တွေက ဒီလိုပါ –





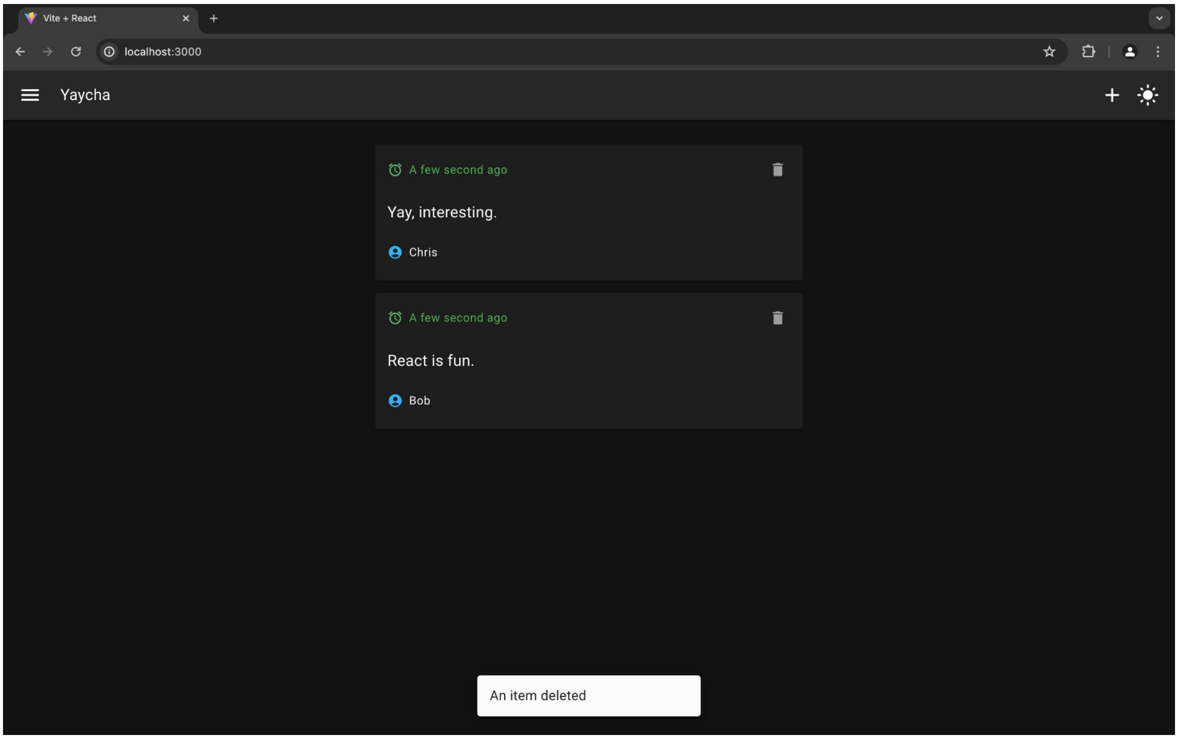
Global Message တွေလည်း စမ်းကြည့်လို့ရအောင် `App.jsx` မှာအခုလို ထည့်ကြည့်ပါ။

React - src/App.jsx

```
// imports

export default function App() {
  const { showForm, setGlobalMsg } = useApp();

  const [data, setData] = useState([...]);

  const remove = id => {
    setData(data.filter(item => item.id !== id));
    setGlobalMsg("An item deleted");
  };

  const add = (content, name) => {
    const id = data[0].id + 1;
```

```
    setData([{ id, content, name }, ...data]);
    setGlobalMsg("An item added");
  };

  ...
}
```

add နဲ့ remove Function တွေမှာ setGlobalMsg နဲ့ ပြစေချင်တဲ့ Message လေးတွေထည့်ပေးလိုက်တာပါ။ Global Message အတွက်သုံးထားတဲ့ ThemedApp က Snackbar မှာ autoHideDuration ထည့်ထားတာကို သတိပြုပါ။ ဒါကြောင့် Message က ပေါ်လာပြီး အချိန်ပြည့်တဲ့အခါ အလိုလို ပြန်ပျောက်သွားမှာပဲ ဖြစ်ပါတယ်။ ရလဒ်က အခုလိုဖြစ်မှာပါ –



အခန်း (၈) - React - React Router

ဒီအခန်းမှာ UI အတွက် လိုအပ်တဲ့ Page တွေကို ရေးသားစီမံတဲ့နေရာမှာ အသုံးဝင်တဲ့ React Router ကို လေ့လာကြမှာပါ။ လူသုံးများတဲ့ နည်းပညာတစ်ခုပါပဲ။ အသုံးပြုရလွယ်ကူပါတယ်။ လက်တွေ့စမ်းလို့ရတဲ့ ကုဒ်တွေမရေးခင်၊ သူ့ရဲ့အလုပ်လုပ်ပုံကို ရှင်းပြတဲ့အနေနဲ့ နမူနာကုဒ်လေးတချို့ ရေးပြပါမယ်။

အရင်ဆုံး လိုအပ်တဲ့ Package ကို ထည့်သွင်းရပါမယ်။

```
npm install react-router-dom
```

React Router မှာ Browser Router, Native Router, Hash Router စသည်ဖြင့် မူကွဲအမျိုးမျိုးရှိလို့ Install လုပ်ရတဲ့ Package အမည်က `react-router` မဟုတ်ဘဲ `react-router-dom` ဖြစ်နေတာပါ။ ဒီ Package ကနေ လိုအပ်တဲ့ လုပ်ဆောင်ချက်တွေကို အခုလို Import လုပ်ယူရပါမယ်။

React
```
const { createBrowserRouter, RouterProvider } from "react-router-dom";
```

`Home`, `About`, `Contact` ဆိုတဲ့ Component (၃) ခုရှိတယ်လို့ သဘောထားပါ။ ဒီ Component တွေကို URL Route ပေါ်မူတည်ပြီး ပြဖို့အတွက် အခုလို သတ်မှတ်ထားနိုင်ပါတယ်။

React

```
const router = createBrowserRouter([
  {
    path: "/",
    element: <Home />,
  },
  {
    path: "/about",
    element: <About />,
  },
  {
    path: "/contact",
    element: <Contact />,
  },
]);
```

ကုဒ်မှာတင် အဓိပ္ပါယ်ပေါ်လွင်ပါတယ်။ / URL အလွတ်ဆိုရင် `Home` Component ကိုပြပေးပြီး၊ `/about` ဆိုရင် `About` Component ကို ပြပေးမှာပါ။ `/contact` ဆိုရင်တော့ `Contact` Component ကို ပြပေးပါလိမ့်မယ်။ <u>React Router က လက်ရှိ URL ပေါ်မှာ မူတည်ပြီး မလိုတဲ့ Component တွေကို Unmount လုပ်ပြီး၊ လိုတဲ့ Component ကို Mount လုပ်ပေးမှာပါ</u>။ Mount လုပ်တယ်ဆိုတာ ထည့်သွင်းဖော်ပြပေးလိုက်တာပါပဲ။

ဒါကြောင့် အသုံးပြုတဲ့အခါ `Home`, `About`, `Contact` စသည်ဖြင့် Component တစ်ခုချင်းစီ ထည့်သုံးစရာမလိုတော့ပါဘူး။ ဒီ Component တွေအားလုံး စုစည်းပါဝင်တဲ့ `RouterProvider` Component ကို အခုလို အသုံးပြုရမှာဖြစ်ပါတယ်။

React

```
function App()
  return <RouterProvider router={router} />
}
```

သူ့ရဲ့အသုံးပြုနည်း အနှစ်ချုပ်က ဒါပါပဲ။ Route တွေ တစ်ခုနဲ့တစ်ခု Navigate လုပ်ပုံလုပ်နည်းတွေ ဆက်ကြည့်ရပါမယ်။ ဒီအတွက် အခြေခံကျတဲ့ လုပ်ဆောင်ချက်နှစ်ခုရှိပါတယ်။

React
```
import { Link, useNavigate } from "react-router-dom";
```

Link က HTML <a> Element လို Component တစ်ခုဖြစ်ပါတယ်။ ဥပမာ – Home Component မှာ ဒီလိုရေးသားထားတယ်ဆိုပါစို့။

React
```
function Home() {
  return <div>
    <h1>Home</h1>
    <Link to="/about">About</Link>
  </div>
}
```

ဒါဟာ <Link> ကိုအသုံးပြုပြီး About Component ကိုသွားလို့ရတဲ့ Link တစ်ခုထည့်သွင်းလိုက်တာပါပဲ။ ရိုးရိုး HTML <a> Element ကိုသုံးပြီး <a href="/about"> လို့ရေးရင်တောင် အလုပ်လုပ်ပါလိမ့်မယ်။ ဒါပေမယ့် Page Reload ဖြစ်မှာပါ။ ဒါကြောင့် <a> ကိုတော့ မသုံးရပါဘူး။ သူ့အစား <Link> ကိုသုံးပေးရတဲ့ သဘောပါ။

useNavigate() ကတော့ Component အနေနဲ့ မဟုတ်ဘဲ ကုဒ်အနေနဲ့ Navigate လုပ်လိုတဲ့အခါ သုံးနိုင်ပါတယ်။ ဥပမာ – Contact Component မှာ အခုလိုရေးထားတယ်ဆိုပါစို့။

React

```
function Contact() {
  const navigate = useNavigate();

  return <form>
    <input type="text" />
    <button type="submit">Submit</button>
    <button type="button" onClick={() => {
      navigate("/");
    }}>
      Go Home
    </button>
  </form>
}
```

useNavigate ကိုသုံးပြီး navigate ကို ကြေညာထားပါတယ်။ ပြီးတဲ့အခါ Button onClick မှာ navigate("/") လို့ပြောထားတဲ့အတွက် အဲ့ဒီ Button ကို နှိပ်လိုက်ရင် Home ကို ရောက်သွားမှာဖြစ်ပါတယ်။ ဒီနည်းနဲ့ Route တစ်ခုနဲ့တစ်ခု အပြန်အလှန်သွားချင်ရင် <Link> သို့မဟုတ် useNavigate ကို အသုံးပြုနိုင်မှာ ဖြစ်ပါတယ်။

နောက်ထပ်မှတ်သားစရာကတော့ Dynamic Route ဖြစ်ပါတယ်။ Router တွေ Create လုပ်စဉ်မှာ အခုလို ရေးလို့ရပါတယ်။

React

```
const router = createBrowserRouter([
  {
    path: "/profile/:id",
    element: <Profile />
  }
]);
```

နမူနာမှာ `path` ကို `/profile/:id` လို့ ပြောထားပါတယ်။ နောက်က `:id` ဟာ Dynamic Route Parameter ဖြစ်ပါတယ်။ `:id` နေရာမှာ လိုအပ်တဲ့ Value တွေထည့်သုံးနိုင်ပါတယ်။ ဥပမာ - `/profile/123` ဆိုရင်လည်း `<Profile>` ကိုပဲ ပြမှာဖြစ်ပြီး `/profile/789` ဆိုရင်လည်း `<Profile>` ကိုပဲပြမှာပါ။ ရှေ့က `/profile` ကအသေဖြစ်ပြီး၊ နောက်က `:id` က နှစ်သက်ရာ Value ဖြစ်လို့ရတာပါ။ ပုံမှန်အားဖြင့် Component တွေထဲမှာ Dynamic Route Value တွေကို အသုံးပြုဖို့ လိုပါတယ်။ အခုလို အသုံးပြုရပါတယ်။

React

```
import { useParams } from "react-router-dom";

function Profile() {
  const { id } = useParams();

  return <div>
    <h1>Profile {id}</h1>
  </div>
}
```

`useParams` ကိုသုံးပြီး Dynamic Route Value တွေကို Component ထဲမှာ ပြန်ယူသုံးလို့ရသွားတာပါ။ နောက်ဆုံးတစ်ခုအနေနဲ့ Template သို့မဟုတ် Nested Route အကြောင်းကြည့်ရပါမယ်။ အရင်ဆုံး Template Component ရှိတယ်ဆိုကြပါစို့။

React

```
import { Outlet } from "react-router-dom";

function Template() {
  return <div>
    <h1>App</h1>
    <Outlet />
  </div>
}
```

Template Component မှာ `Outlet` လို့ခေါ်တဲ့ React Router ရဲ့လုပ်ဆောင်ချက်ကို သုံးထားပါတယ်။ ပြီးတဲ့အခါ စစချင်းရေးခဲ့တဲ့ `router` ကို အခုလို ပြင်ရေးလိုက်ပါမယ်။

React

```
const router = createBrowserRouter([
  {
    path: "/",
    element: <Template />
    children: [
      {
        path: "/",
        element: <Home />,
      },
      {
        path: "/about",
        element: <About />,
      },
      {
        path: "/contact",
        element: <Contact />,
      },
    ]
  }
]);
```

နမူနာအရ `<Template>` က အဓိက Component ဖြစ်သွားပြီး `Home`, `About`, `Contact` တို့က သူ့ရဲ့ Children တွေဖြစ်သွားတာပါ။ ဒါကြောင့် `Home`, `About`, `Contact` တို့ကို Template ရဲ့ `<Outlet />` နေရာမှာ ထည့်ပြပေးသွားမှာပဲဖြစ်ပါတယ်။ URL က `/` အလွတ်ဆိုရင် Template ထဲမှာ `Home` Component ကိုထည့်ပြပေးမှာပါ။ URL က `/about` ဆိုရင်တော့ Template ထဲမှာ `About` Component ကိုထည့်ပြမှာ ဖြစ်သွားပါတယ်။ ဒါဟာလည်း လိုအပ်လေ့ရှိတဲ့ အခြေခံလိုအပ်ချက်တွေထဲမှာ တစ်ခုအပါအဝင်ဖြစ်ပါတယ်။

Adding Router

ဒီလောက်ဆိုရင် လုပ်လက်စပရောဂျက်မှာ React Router ကိုသုံးပြီး လိုအပ်မယ့် Route တွေ Page တွေ ထည့်လို့ ရသွားပါပြီ။ လိုအပ်တဲ့ ဖိုင်တွေ အရင်ထည့်ကြပါမယ်။ ဒီလိုပါ –

```
yaycha
├── node_modules/
├── README.md
├── index.html
├── package-lock.json
├── package.json
├── public/
├── src/
│   ├── Template.jsx
│   ├── ThemedApp.jsx
│   ├── assets/
│   ├── components/
│   │   ├── AppDrawer.jsx
│   │   ├── Form.jsx
│   │   ├── Header.jsx
│   │   ├── Item.jsx
│   │   └── UserList.jsx
│   ├── main.jsx
│   └── pages/
│       ├── Comments.jsx
│       ├── Home.jsx
│       ├── Likes.jsx
│       ├── Login.jsx
│       ├── Profile.jsx
│       └── Register.jsx
└── vite.config.js
```

src ထဲမှာ Template.jsx တိုးလာပြီး components ထဲမှာ UserList.jsx ဖိုင်တစ်ခု ထပ်တိုးသွားပါတယ်။ ပြီးတဲ့အခါ pages အမည်နဲ့ ဖိုဒါအသစ်တစ်ခု ထပ်ထည့်ပြီး Comments.jsx, Home.jsx, Likes.jsx, Login.jsx, Profile.jsx, Register.jsx အမည်တွေနဲ့

ဖိုင် (၆) ခု ထပ်တိုးထားပါတယ်။

ဆက်လက်ပြီး React Router ကို ပရောဂျက်ထဲမှာ အခုလို Install လုပ်ပေးလိုက်ပါ။

```
npm install react-router-dom
```

မူလက `App.jsx` မှာ ရေးထားတဲ့ ကုဒ်တွေကို `Template.jsx` နဲ့ `pages/Home.jsx` တို့ထဲမှာ နှစ်ပိုင်းခွဲထုတ်ထားပါတယ်။ ဒါကြောင့် `Template.jsx` မှာပါဝင်ရမယ့် ကုဒ်တွေက ဒီလိုပါ။

React - src/Template.jsx

```
import { Box, Container, Snackbar } from "@mui/material";

import { Outlet } from "react-router-dom";

import Header from "./components/Header";
import AppDrawer from "./components/AppDrawer";

import { useApp } from "./ThemedApp";

export default function Template() {
    const { globalMsg } = useApp();

  return (
    <Box>
      <Header />
      <AppDrawer />

      <Container maxWidth="sm" sx={{ mt: 4 }}>
        <Outlet />
      </Container>
```

```
      <Snackbar
        anchorOrigin={{
          horizontal: "center",
          vertical: "bottom",
        }}
        open={Boolean(globalMsg)}
        autoHideDuration={6000}
        onClose={() => setGlobalMsg(null)}
        message={globalMsg}
      />
    </Box>
  );
}
```

`<Outlet>` ပါသွားတာကို သတိပြုပါ။ တခြား Page တွေကို ဒီ Template ရဲ့ `<Outlet>` နေရာမှာ ပြပေးမယ့်သဘော ဖြစ်ပါတယ်။ မူလက `ThemedApp` ထဲမှာ ပါဝင်တဲ့ `AppDrawer` နဲ့ `Snackbar` တို့ကိုလည်း `Template` ထဲမှာ ရွှေ့ပြောင်း ထည့်သွင်းထားပါတယ်။ ဒီလုပ်ဆောင်ချက်တွေက Template ထဲမှာရှိမှ ပိုသဘာဝကျမှာမို့လို့ပါ။ `Home.jsx` မှာတော့ အခုလိုရေးပေးရမှာပါ။

React - src/pages/Home.jsx

```
import { useState } from "react";

import { Box } from "@mui/material";

import Form from "../components/Form";
import Item from "../components/Item";

import { useApp } from "../ThemedApp";

export default function Home() {
  const { showForm, setGlobalMsg } = useApp();
```

```
  const [data, setData] = useState([
    { id: 3, content: "Yay, interesting.", name: "Chris" },
    { id: 2, content: "React is fun.", name: "Bob" },
    { id: 1, content: "Hello, World!", name: "Alice" },
  ]);

  const remove = id => {
    setData(data.filter(item => item.id !== id));
    setGlobalMsg("An item deleted");
  };

  const add = (content, name) => {
    const id = data[0].id + 1;
    setData([{ id, content, name }, ...data]);
    setGlobalMsg("An item added");
  };

  return (
    <Box>
      {showForm && <Form add={add} />}

      {data.map(item => {
        return (
          <Item key={item.id} item={item} remove={remove} />
        );
      })}
    </Box>
  );
}
```

မူလက `App.jsx` မှာ ရေးထားတဲ့ `<Form>` နဲ့ `<Item>` တွေဖော်ပြတဲ့ ကုဒ်တွေပါပဲ။ ဒီလို လိုအပ်တဲ့ ဖွဲ့စည်းပုံရပြီဆိုရင် Route တွေစရေးလို့ရပါပြီ။ ဖိုင်တစ်ခု သီးခြားရေးရင်လည်း ရနိုင်ပေမယ့် `ThemedApp` ထဲမှာပဲ ရေးလိုက်ပါတော့မယ်။

React - src/ThemedApp.jsx

```
import { useState, createContext, useContext, useMemo } from "react";
import {
  CssBaseline,
  ThemeProvider,
  createTheme,
} from "@mui/material";
import {
  createBrowserRouter,
  RouterProvider,
} from "react-router-dom";

import { deepPurple, grey } from "@mui/material/colors";

import Template from "./Template";
import Home from "./pages/Home";

const AppContext = createContext();

export function useApp() {
  return useContext(AppContext);
}

const router = createBrowserRouter([
  {
    path: "/",
    element: <Template />,
    children: [
      {
        path: "/",
        element: <Home />,
      }
    ],
  },
]);
```

```
export default function ThemedApp() {
  const [showDrawer, setShowDrawer] = useState(false);
  const [showForm, setShowForm] = useState(false);
  const [globalMsg, setGlobalMsg] = useState(null);
  const [auth, setAuth] = useState(null);
  const [mode, setMode] = useState("dark");

  const theme = useMemo(() => {
    return createTheme({
      palette: {
        mode,
        primary: deepPurple,
        banner: mode === "dark" ? grey[800] : grey[200],
        text: { fade: grey[500] },
      },
    });
  }, [mode]);

  return (
    <ThemeProvider theme={theme}>
      <AppContext.Provider
        value={{
          showDrawer, setShowDrawer,
          showForm, setShowForm,
          globalMsg, setGlobalMsg,
          auth, setAuth,
          mode, setMode,
        }}>
        <RouterProvider router={router} />
        <CssBaseline />
      </AppContext.Provider>
    </ThemeProvider>
  );
}
```

`createBrowserRouter` နဲ့ `RouterProvider` တို့ကို `react-router-dom` ကနေ Import လုပ်ယူပါတယ်။ ပြီးတဲ့အခါ `createBrowserRouter` နဲ့ `router` တစ်ခုဖန်တီးပါတယ်။ `<Template>` ရဲ့ Children အဖြစ် `<Home>` ကိုသတ်မှတ်ပေးလိုက်တာပါ။ App ကို ဆက်မသုံးတော့ပါဘူး။ ဖျက်ချင်ရင် ဖျက်လိုက်လို့ ရပါတယ်။ App အစား `RouterProvider` ကို အဓိက Component အနေနဲ့ ထည့်သုံးပေးလိုက်တာ ဖြစ်ပါတယ်။ ဒါကြောင့် အခုနေ စမ်းကြည့်ရင် မူလအတိုင်း အလုပ်လုပ်သွားကို တွေ့ရပါလိမ့်မယ်။ ရလဒ်မှာ အပြောင်းအလဲ မရှိသေးပေမယ့် လိုအပ်တဲ့ Router ကို Setup လုပ်ပြီးပြီမို့လို့ တခြား Page တွေ ထပ်ထည့်လို့ ရသွားပြီဖြစ်ပါတယ်။

Other Pages

ဆက်ရေးမယ့် Page တွေများလို့ ကုဒ်တွေလည်း များပါလိမ့်မယ်။ ဒါပေမယ့် အသစ်အဆန်းတွေတော့ ပါမှာမဟုတ်ပါဘူး။ သိပြီးသားတွေ များပါလိမ့်မယ်။ ဒါကြောင့် လိုက်ရေးရင်တော့ ရေးကျင့်ရပါလိမ့်မယ်။ သိပြီးသားတွေများလို့ လိုက်မရေးချင်တော့ရင်လည်း အားလုံးပါပြီးသားကို ဒီလင့်ကနေ Download လုပ်ယူလိုက်လို့ ရပါတယ်။

https://github.com/eimg/rsd25/tree/main/chapter-8/

ရေးရမယ့်ကုဒ်တွေကိုလည်း တစ် Page ချင်း ဖော်ပြပေးလိုက်ပါတယ်။ လိုက်ဖတ်ကြည့်သင့်ပါတယ်။

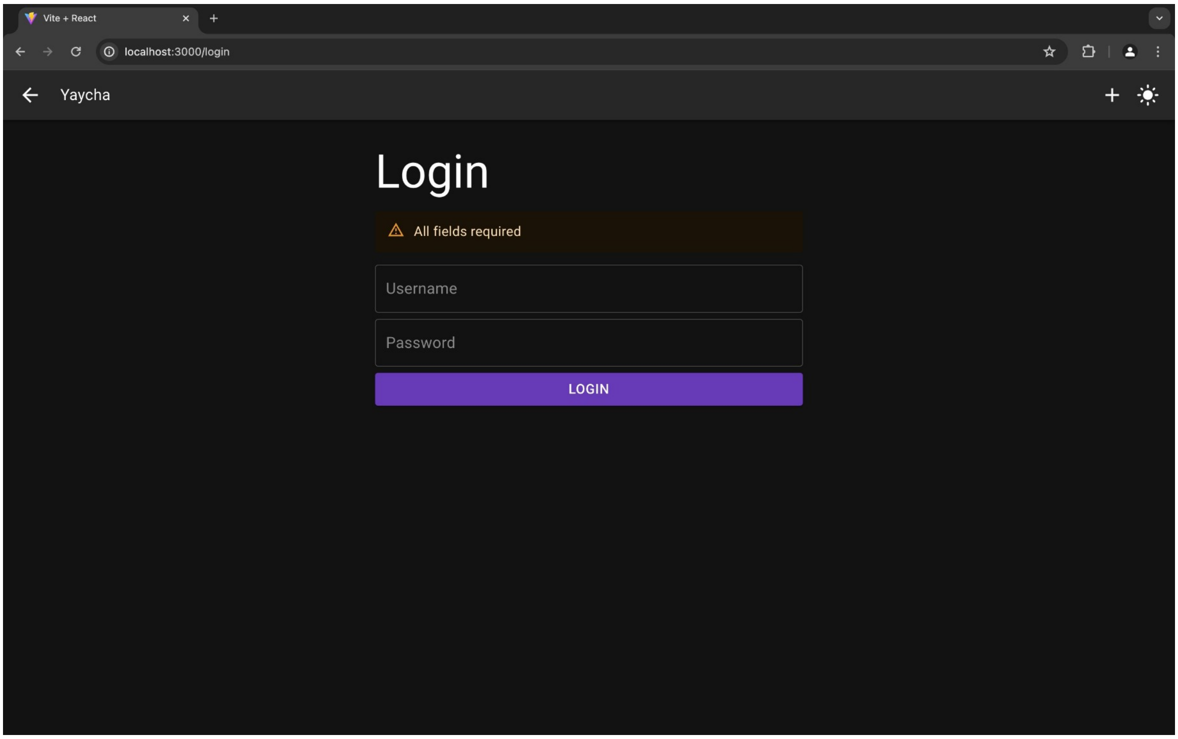
Login Page

React - src/pages/Login.jsx

```
import { Alert, Box, Button, TextField, Typography } from "@mui/material";

import { useNavigate } from "react-router-dom";
import { useApp } from "../ThemedApp";

export default function Login() {
  const navigate = useNavigate();
  const { setAuth } = useApp();
```

```
  return (
    <Box>
      <Typography variant="h3">Login</Typography>

      <Alert severity="warning" sx={{ mt: 2 }}>All fields required</Alert>

      <form onSubmit={e => {
        e.preventDefault();
        setAuth(true);
        navigate("/");
      }}>
        <Box
          sx={{ display: "flex", flexDirection: "column", gap: 1, mt: 2, }}>
          <TextField placeholder="Username" fullWidth />
          <TextField type="password" placeholder="Password" fullWidth />
          <Button type="submit" variant="contained" fullWidth>
            Login
          </Button>
        </Box>
      </form>
    </Box>
  );
}
```

Textfield (၂) ခုနဲ့ Button (၁) ခုပါတဲ့ Login Form ကိုပြပေးတဲ့ကုဒ်ဖြစ်ပါတယ်။ Login Error တွေရှိရင်ပြဖို့ Alert (၁) ခုလည်း ထည့်ထားပါသေးတယ်။ လောလောဆယ် Button ကိုနှိပ်လိုက်ရင် အချက်အလက်တွေ မှန်မမှန် မစစ်သေးဘဲ setAuth ကို true လုပ်ပေးလိုက်ပြီး navigate နဲ့ Home ကို သွားခိုင်းထားပါတယ်။ သူ့ရဲ့ဖော်ပြပုံက အခုလိုဖြစ်မှာပါ။



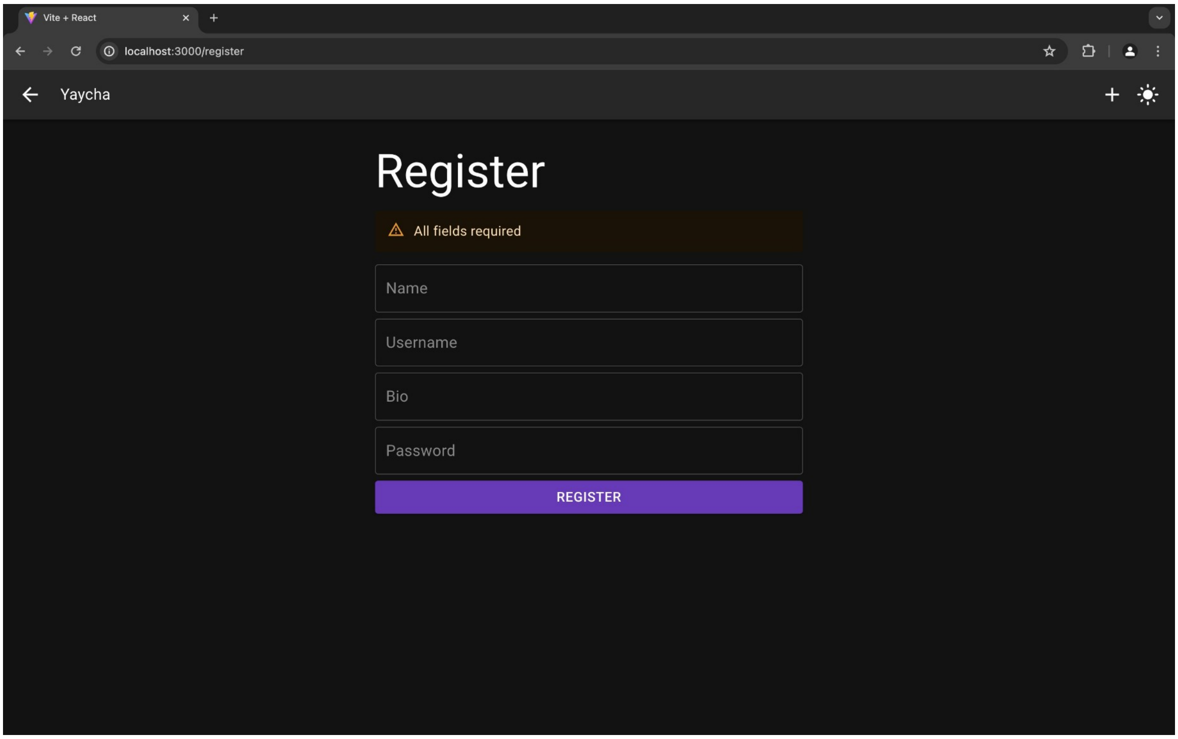
Register Page

React - src/pages/Register.jsx

```
import { Box, Button, TextField, Typography, Alert } from "@mui/material";

export default function Register() {
  return (
    <Box>
      <Typography variant="h3">Register</Typography>

      <Alert severity="warning" sx={{ mt: 2 }}>All fields required</Alert>

      <form>
        <Box
          sx={{ display: "flex", flexDirection: "column", gap: 1, mt: 2, }}>
          <TextField placeholder="Name" fullWidth />
          <TextField placeholder="Username" fullWidth />
          <TextField placeholder="Bio" fullWidth />
```

```
            <TextField type="password" placeholder="Password" fullWidth />
            <Button type="submit" variant="contained" fullWidth>
              Register
            </Button>
          </Box>
        </form>
      </Box>
    );
}
```

စောစောက Login Page လိုမျိုးပါပဲ။ ဒီတစ်ခါတော့ Textfield (၄) ခုပါသွားပါတယ်။ သူ့ရဲ့ဖော်ပြပုံကတော့ ဒီလိုပါ။



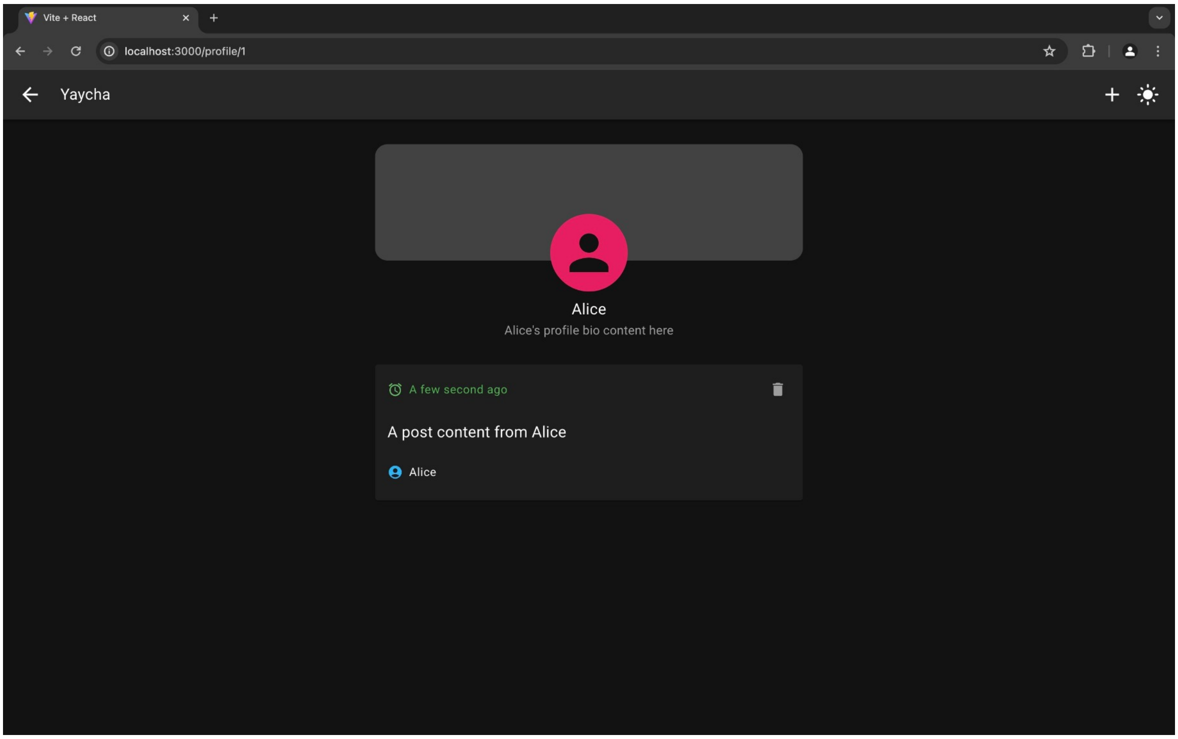
Profile Page

React - src/pages/Profile.jsx

```
import { Avatar, Box, Typography } from "@mui/material";
import { pink } from "@mui/material/colors";

import Item from "../components/Item";

export default function Profile() {
  return (
    <Box>
      <Box sx={{ bgcolor: "banner", height: 150, borderRadius: 4 }}></Box>
      <Box
        sx={{
          mb: 4,
          marginTop: "-60px",
          display: "flex",
          flexDirection: "column",
          justifyContent: "center",
          alignItems: "center",
          gap: 1,
        }}>

        <Avatar sx={{ width: 100, height: 100, bgcolor: pink[500] }} />

        <Box sx={{ textAlign: "center" }}>
          <Typography>Alice</Typography>
          <Typography sx={{ fontSize: "0.8em", color: "text.fade" }}>
            Alice's profile bio content here
          </Typography>
        </Box>
      </Box>

      <Item
        key={1}
        remove={() => {}}
```

```
          item={{
            id: 1,
            content: "A post content from Alice",
            name: "Alice",
          }}
        />
      </Box>
    );
}
```

ဒါလည်းပဲ အသစ်အဆန်းမဟုတ်ပါဘူး။ `Item` Component ကို ယူသုံးထားတာပဲ သတိပြုပါ။ နမူနာ သက်သက်ပါပဲ။ နောက်တော့မှ တကယ့် Data အမှန်နဲ့ ချိတ်ကြပါမယ်။ User တစ်ယောက် Login ဝင် လိုက်ရင် သူ့ရဲ့ Profile မှာ သူထည့်သွင်းထားတဲ့ Item တွေကို ပြချင်တာပါ။ သူ့ရဲ့ဖော်ပြပုံက ဒီလိုပါ။



Comments Page

React - src/pages/Comments.jsx

```
import { Box, Button, TextField } from "@mui/material";
import Item from "../components/Item";

export default function Comments() {
  return (
    <Box>
      <Item
        primary
        key={1}
        item={{
          id: 1,
          content: "Initial post content from Alice",
          name: "Alice",
        }}
        remove={() => {}} />

      <Item
        key={2}
        item={{
          id: 2,
          content: "A comment from Bob",
          name: "Bob",
        }}
        remove={() => {}} />
      <Item
        key={3}
        item={{
          id: 3,
          content: "A comment reply from Alice",
          name: "Alice",
        }}
        remove={() => {}} />
```

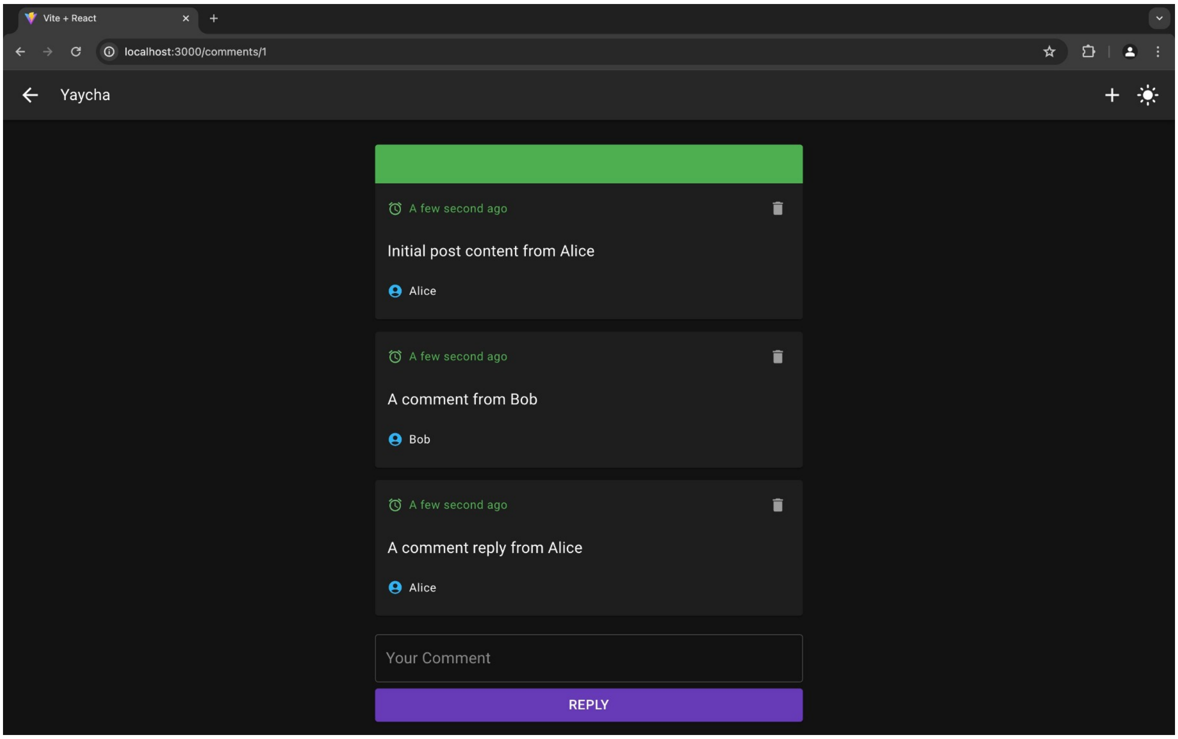
```
      <form>
        <Box
          sx={{ display: "flex", flexDirection: "column", gap: 1, mt: 3, }}>
          <TextField multiline placeholder="Your Comment" />
          <Button type="submit" variant="contained">Reply</Button>
        </Box>
      </form>
    </Box>
  );
}
```

Home Page ကနေ Post တစ်ခုကို နှိပ်လိုက်ရင်၊ အဲ့ဒီ Post နဲ့အတူရှိနေတဲ့ Comments တွေကို ပြချင်တာပါ။ Post အတွက်ရော၊ Comments အတွက်ပါ၊ `Item` Component ကိုပဲ အတူတူသုံးထားတာ သတိပြုပါ။ မတူအောင် ခွဲထားရင် ပိုကောင်းပေမယ့် ကုဒ်တွေလည်း များနေပြီမို့လို့ မခွဲတော့ပါဘူး။ တစ်ခုထဲကိုပဲ သုံးလိုက်ပါတော့မယ်။ ဒါကြောင့် ကွဲပြားသွားအောင် Post အတွက် Item မှာ `primary` ဆိုတာလေး ပါသွားတာကို သတိပြုပါ။ အောက်ဆုံးမှာ Comment အသစ်တွေထည့်စရာ Form တစ်ခုလည်း ထည့်ပေးထားပါတယ်။ ဒီကုဒ်နဲ့ ချိတ်ဆက်အလုပ်လုပ်တာ မှန်သွားအောင် `components/Item.jsx` မှာ အခုလိုလေး ပြင်ပေးဖို့ လိုပါလိမ့်မယ်။

React - src/components/Item.jsx

```
import { ..., CardMedia } from "@mui/material";
// Other Imports
import { green } from "@mui/material/colors";

export default function Item({ item, remove, primary }) {
  return <Card sx={{ mb: 2 }}>
    {primary && <Box sx={{ height: 50, bgcolor: green[500] }} />}
    ...
  </Card>;
}
```

`primary` Props သာ ရှိခဲ့မယ်ဆိုရင် သူများနဲ့မတူဘဲ ထင်ရှားအောင် Box နဲ့ နဖူးစီးအရောင်လေး ထည့်ပေးလိုက်တာပါ။ အဲ့ဒီနေရာမှာ နောက်ပိုင်း ပုံတွေ၊ ဗွီဒီယိုတွေနဲ့ ပြချင်ရင်လည်း ရနိုင်ပါတယ်။ ဒါလေးပြင်ပြီးရင် Comments Page ရဲ့ဖော်ပြပုံက အခုလိုဖြစ်မှာပါ။



Likes Page

React - src/pages/Likes.jsx

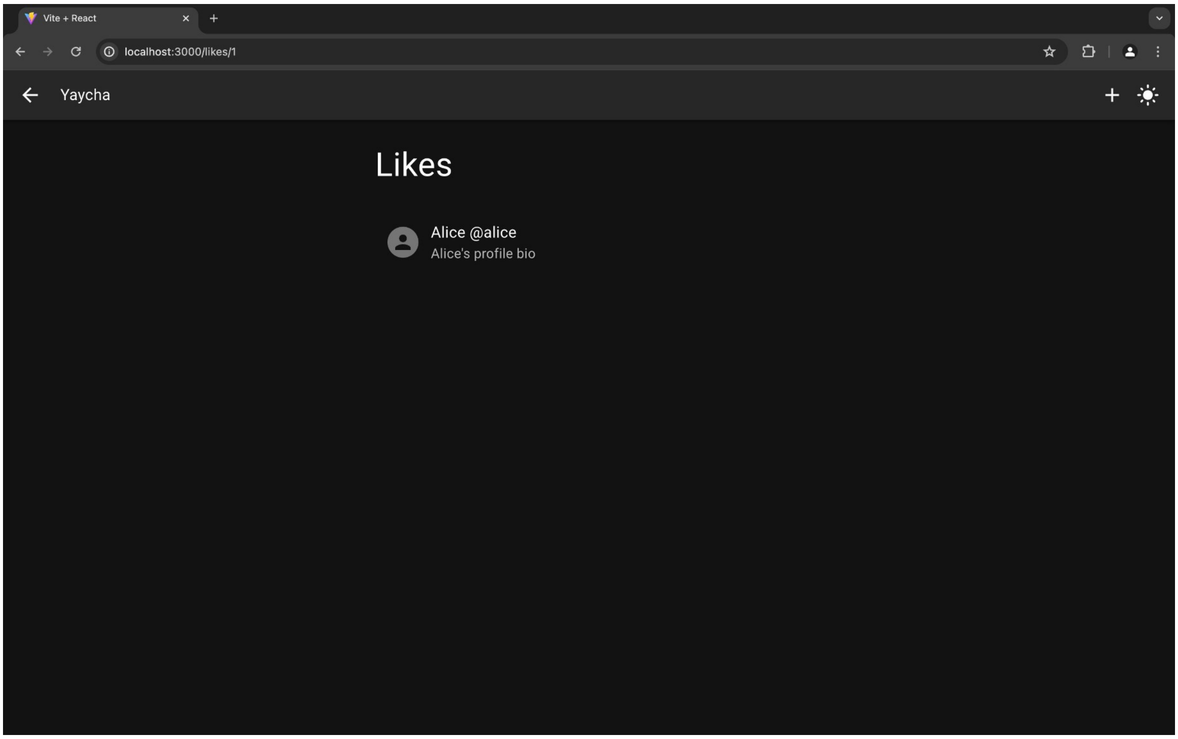
```
import UserList from "../components/UserList";

import { Box } from "@mui/material";

export default function Likes() {
  return <Box>
    <UserList title="Likes" />
  </Box>
}
```

ဒီကုဒ်ကတော့ တိုပါတယ်။ Post တွေ Comment တွေမှာ Like လုပ်ထားတဲ့ User စာရင်းကို ပြပေးချင်တာ ပါ။ `<UserList>` ကို ခေါ်သုံးထားတာ သတိပြုပါ။ ဒါကြောင့် `components/UserList.jsx` မှာ အခုလို ရေးပေးရပါမယ်။

React - src/components/UserList.jsx

```
import {
  Box,
  Typography,
  List,
  ListItem,
  Avatar,
  ListItemText,
  ListItemAvatar,
} from "@mui/material";

export default function UserList({ title }) {
  return (
    <Box>
      <Typography variant="h4" sx={{ mb: 3 }}>
        {title}
      </Typography>
      <List>
        <ListItem>
          <ListItemAvatar><Avatar /></ListItemAvatar>
          <ListItemText
            primary="Alice @alice"
            secondary="Alice's profile bio"
          />
        </ListItem>
      </List>
    </Box>
  );
}
```

List တွေ ListItem တွေကိုသုံးပြီး User တွေကို စာရင်းနဲ့ ပြဖို့အတွက် ပြင်ဆင်ထားတာပါ။ ထုံးစံ အတိုင်း Data အမှန်နဲ့ နောက်မှချိတ်ကြပါမယ်။ ဒါတွေရေးပြီးရင် Likes Page ရဲ့ဖော်ပြပုံက ဒီလိုဖြစ်မှာပါ။



ပါရမယ့် ကုဒ်တွေစုံသွားပါပြီ။ ဒါကြောင့် စမ်းကြည့်လို့ရအောင် Route တွေထပ်ဖြည့်ကြပါမယ်။ ThemedApp ရဲ့ router က အခုလိုပုံစံ ဖြစ်ရပါမယ်။

React - src/ThemedApp.jsx

```
import Template from "./Template";

import Home from "./pages/Home";
import Login from "./pages/Login";
import Register from "./pages/Register";
import Likes from "./pages/Likes";
import Profile from "./pages/Profile";
import Comments from "./pages/Comments";
```

```
const router = createBrowserRouter([
  {
    path: "/",
    element: <Template />,
    children: [
      {
        path: "/",
        element: <Home />,
      },
      {
        path: "/login",
        element: <Login />,
      },
      {
        path: "/register",
        element: <Register />,
      },
      {
        path: "/comments/:id",
        element: <Comments />,
      },
      {
        path: "/profile/:id",
        element: <Profile />,
      },
      {
        path: "/likes/:id",
        element: <Likes />,
      },
    ],
  },
]);
```

ဒါလည်း လုပ်ဆောင်ချက်အသစ်မပါဘဲ လိုအပ်တဲ့ Component တွေကို Import လုပ်ပြီး သက်ဆိုင်ရာ `path` တွေနဲ့ တွဲပေးလိုက်တာပါပဲ။ `comments`, `profile` နဲ့ `likes` တို့မှာ Dynamic Route Parameter တွေ ပါသွားတာကို သတိပြုပါ။

ဆက်လက်ပြီး `components/AppDrawer.jsx` မှာပါတဲ့ List Item တွေက အခုလိုဖြစ်ရပါမယ်။

React - src/components/AppDrawer.jsx

```
<List>
  <ListItem>
    <ListItemButton onClick={() => navigate("/")}>
      <ListItemIcon>
        <HomeIcon />
      </ListItemIcon>
      <ListItemText>Home</ListItemText>
    </ListItemButton>
  </ListItem>
  <Divider />

  {auth && (
    <>
      <ListItem>
        <ListItemButton onClick={() => navigate("/profile/1")}>
          <ListItemIcon>
            <ProfileIcon />
          </ListItemIcon>
          <ListItemText>Profile</ListItemText>
        </ListItemButton>
      </ListItem>
      <ListItem>
        <ListItemButton onClick={() => setAuth(null)}>
          <ListItemIcon>
            <LogoutIcon color="error" />
          </ListItemIcon>
```

```
          <ListItemText>Logout</ListItemText>
        </ListItemButton>
      </ListItem>
    </>
  )}

  {!auth && (
    <>
      <ListItem>
        <ListItemButton onClick={() => navigate("/register")}>
          <ListItemIcon>
            <RegisterIcon />
          </ListItemIcon>
          <ListItemText>Register</ListItemText>
        </ListItemButton>
      </ListItem>
      <ListItem>
        <ListItemButton onClick={() => navigate("/login")}>
          <ListItemIcon>
            <LoginIcon />
          </ListItemIcon>
          <ListItemText>Login</ListItemText>
        </ListItemButton>
      </ListItem>
    </>
  )}
</List>
```

`onClick` တွေထည့်ပေးလိုက်ပြီး `navigate` နဲ့ သက်ဆိုင်ရာ `path` ကိုသွားခိုင်းလိုက်တာပါ။ Profile အတွက် `path` က `/profile/1` ဖြစ်တာကိုသတိပြုပါ။ Dynamic Parameter ကို ယာယီ 1 နဲ့ခဏထည့်စမ်းလိုက်တာပါ။ `navigate` ကိုသုံးလို့ရဖို့အတွက် လိုတဲ့နေရာမှာ အခုလိုတွေ ချိတ်ဆက်ပေးဖို့လိုတာကို မမေ့ပါနဲ့။

```
import { useNavigate } from "react-router-dom";
```

```
const navigate = useNavigate();
```

ဒါဆိုရင် `Login`, `Register`, `Profile` နဲ့ `Home Route` တွေကို စမ်းကြည့်လို့ရသွားပါပြီ။ `Comment` ကို စမ်းကြည့်လို့ရဖို့အတွက် `components/Item.jsx` ကိုအခုလိုပြင်ပေးပါ။

React - src/components/Item.jsx

```
// Other imports

import { useNavigate } from "react-router-dom";

export default function Item({ item, remove, primary }) {
    const navigate = useNavigate();

  return (
    <Card sx={{ mb: 2 }}>
      {primary && <CardMedia sx={{ height: 50, bgcolor: green[500] }} />}

        <CardContent onClick={() => navigate("/comments/1")}>
          <Box ...>
            ...
            <IconButton
              sx={{ color: "text.fade" }}
              size="small"
              onClick={e => {
                remove(item.id);
                e.stopPropagation();
              }}>
              <DeleteIcon
                color="inherit"
                fontSize="inherit"
```

```
                />
            </IconButton>
          </Box>
          ...
        </CardContent>
    </Card>
  );
}
```

`CardContent` မှာ `onClick` နဲ့ `Comments Page` ကို `navigate` လုပ်ထားပါတယ်။ ဒီနေရာမှာ စိတ်ဝင်စားဖို့ ကောင်းတာက Delete ခလုပ်မှာ ပြင်လိုက်တဲ့ကုဒ် ဖြစ်ပါတယ်။ Delete ခလုပ်ဟာ `CardContent` ထဲမှာ ရှိနေလို့ Delete ကို နှိပ်ရင် `CardContent` ကိုနှိပ်သလိုလည်း ဖြစ်နေနိုင်ပါတယ်။ ဒါကြောင့် Delete ခလုပ်မှာ `stopPropagation()` နဲ့ ကျန်တဲ့ Event တွေ ဆက်မလုပ်ဖို့ သတ်မှတ်ထားပါတယ်။ မဟုတ်ရင် Delete နှိပ်လိုက်တိုင်း Item လည်းပျက်သွားမယ်၊ Comments Page ကိုလည်း ရောက်သွားပါလိမ့်မယ်။ သဘာဝမကျပါဘူး။ ဒါကြောင့် Delete လုပ်ပြီးရင် ရပ်ခိုင်းလိုက်တာပါ။

ဒီနေရာမှာ ဒီလောက်ဆိုရင် လုံလောက်ပါပြီ။ Likes Page ကလွဲရင် Route အားလုံးကို UI မှာပဲ တစ်ခုပြီး တစ်ခု နှိပ်ပြီးစမ်းကြည့်လို့ရသွားပါပြီ။

ဒီအထိ အားလုံးကူးယူရေးသားခဲ့ရင် ချီးကျူးပါတယ်။ Repetitive ဖြစ်နေတဲ့ ကုဒ်တွေရေးရတာ ပျင်းဖို့ကောင်းပေမယ့် အလေ့အကျင့်ဆိုတာ လေ့ကျင့်ယူမှပဲ ရတာပါ။ ရလိုက်တဲ့ အလေ့အကျင့်က အလကားမဖြစ်ပါဘူး။ ရေးရင်းနဲ့ ဘယ်နားကျန်သွားလည်း မသိဘူး။ ပြန်စမ်းတာ အဆင်မပြေတော့ဘူးဆိုရင်တော့ အားလုံးမှန်အောင် စမ်းထားပြီးသား ကုဒ်ကို အောက်ကလင့်မှာ ရယူပြီး နောက်သင်ခန်းစာတွေကို လိုက်လုပ်နိုင်ပါတယ်။

https://github.com/eimg/rsd25/tree/main/chapter-8/

အခန်း (၉) - JavaScript - Promises, async / await

JavaScript ဟာ Single Threaded Language ဖြစ်ပါတယ်။ Single Threaded Language တွေဟာ ကားတွေက တစ်စီးပြီးတစ်စီး ရှေ့နောက် တန်းစီပြီးသွားရတဲ့ တစ်လမ်းမောင်း လမ်းတစ်ခုလိုပါပဲ။ A, B, C ကား သုံးစီးမှာ အလယ်က B နှေးနေရင် သူ့နောက်က C က ကျော်တက်လို့မရဘဲ စောင့်ပြီးလိုက်ရမှာပါ။ ဥပမာ – ဒီ PHP ကုဒ်ကို လေ့လာကြည့်ပါ။

PHP - sync.php

```php
<?php

function A() {
  echo "Function A \n";
}

function B() {
  sleep(2);
  echo "Function B \n";
}

function C() {
  echo "Function C \n";
}
```

```
A();
B();
C();
```

A, B, C Function သုံးခုရှိရာမှာ Function B က အချိန်ယူပြီး အလုပ်လုပ်ဖို့လိုတဲ့၊ နှေးတဲ့ Function ပါ။ ဒီကုဒ်ကို Run ကြည့်လိုက်ရင် ရမယ့်ရလဒ်က အခုလိုဖြစ်ပါလိမ့်မယ်။

```
php sync.php

Function A
Function B
Function C
```

A အလုပ်လုပ်တယ်။ ပြီးတဲ့အခါ B အလုပ်လုပ်တယ်။ ခဏစောင့်ရပါလိမ့်မယ်။ ပြီးမှ C ကို ဆက်အလုပ်လုပ်သွားမှာပဲ ဖြစ်ပါတယ်။

PHP က Single Threaded Language တော့ မဟုတ်ပါဘူး။ ဒါပေမယ့် Web Server နဲ့ ပူးတွဲအသုံးပြုတဲ့အခါ Web Server ရဲ့အကန့်အသတ်ကြောင့် အများအားဖြင့် Single Threaded ပုံစံပဲ အလုပ်လုပ်လေ့ရှိပါတယ်။

JavaScript ကတော့ Single Threaded Language ဆိုပေမယ့် Event Loop ခေါ် ထူးခြားတဲ့ လုပ်ဆောင်ချက်တစ်ခု ပါဝင်ပါတယ်။ ဒီလုပ်ဆောင်ချက်ရဲ့အကူအညီနဲ့ နှေးတဲ့အလုပ်တွေကို ဘေးခဏဖို့ထားလို့ရသွားပါတယ်။ တစ်လမ်းမောင်း ဥပမာနဲ့ ဆက်ကြည့်ရင် A, B, C ကားသုံးစီးမှာ B ကနှေးနေတယ်ဆိုရင် ဘေးခဏ ဖယ်ပေးနိုင်ပါတယ်။ ဒါကြောင့် C က နှေးတဲ့ B ကို စောင့်စရာမလိုတော့ဘဲ ကျော်တက်ပြီးသွားလို့ ရသွားမှာ ဖြစ်ပါတယ်။ ဒီ JavaScript ကုဒ်ကို လေ့လာကြည့်ပါ။

JavaScript - sync.js

```
function A() {
  console.log("Function A");
}
```

```
function B() {
  return setTimeout(() => {
    console.log("Function B");
  }, 2000);
}

function C() {
  console.log("Function C");
}

A();
B();
C();
```

စောစောကလိုပါပဲ။ Function A, B, C (၃) ခုရှိတာမှာ B က အချိန်ယူပြီးအလုပ်လုပ်ဖို့လိုတဲ့ Function ဖြစ်ပါတယ်။ သူ့ကို Run ကြည့်လိုက်ရင် အခုလိုရလဒ်ကို ရပါလိမ့်မယ်။

```
node sync.js

Function A
Function C
Function B
```

နှေးတဲ့ Function B ကို စောင့်စရာမလိုဘဲ Function C က ကျော်တက်ပြီး အလုပ်လုပ်သွားတာ ဖြစ်ပါတယ်။ ဒီနေရာမှာ အရေးကြီးတာက Function B တစ်ခုလုံးကို ဘေးဖယ်ပေးလိုက်တာ မဟုတ်ဘဲ၊ Function B ထဲမှာပါတဲ့ အချိန်ကြာတဲ့ setTimeout ရဲ့ Callback Function ကို ဘေးဖယ်ပေးလိုက်တာ ဖြစ်ပါတယ်။

JavaScript က Run တဲ့အချိန်မှာ အလုပ်လုပ်ရမယ့် မူလ Function တွေကို Call Stack ခေါ် စာရင်းတစ်ခု ထဲထည့်ထားပြီး၊ ဘေးဖယ်ထားလို့ရတဲ့ Callback Function တွေကို Task Queue ခေါ် သီးခြား စာရင်းထဲမှာ ထည့်ထားလိုက်မှာပါ။ Event Loop က Call Stack ကို စောင့်ကြည့်ပြီး Call Stack ထဲမှာ လုပ်စရာ

အလုပ် ကုန်ပြီဆိုရင် Task Queue ထဲက Callback တွေကို Call Stack ထဲရွှေ့ပြီး ဆက်အလုပ်လုပ်စေပါတယ်။

ဒါတွေနားရှုပ်တယ်ထင်ရင် အထက်မှာပြောခဲ့သလိုဘဲ၊ တစ်လမ်းမောင်းလမ်းမှာ နှေးနေတဲ့ကားကို ဘေးခဏချပေးပြီး နောက်ကားက ကျော်တက်ခွင့်ပြုလိုက်တယ်။ ပြီးမှ ဘေးချထားတဲ့ကားက နောက်ကနေ ဆက်လိုက်သွားတယ်လို့ မြင်ကြည့်ရင်ရပါတယ်။

ဒီနည်းကို သုံးတဲ့အတွက် Synchronous ပုံစံ အစီအစဉ်အတိုင်း တစ်ခုပြီးမှတစ်ခု အလုပ်လုပ်ရတဲ့ Single Threaded JavaScript မှာ၊ လိုအပ်ရင် အလုပ်တွေကို Asynchronous ပုံစံ တစ်ခုကိုတစ်ခု စောင့်စရာမလို ဘဲလုပ်ခိုင်းလို့ ရသွားပါတယ်။

Promises

Promises ဟာ Asynchronous ကုဒ်တွေကို စီမံလို့ပိုကောင်းတဲ့ ရေးနည်းဖြစ်ပါတယ်။ ဥပမာ – စောစောက ကုဒ်နမူနာကို အခုလို ပြင်ရေးကြည့်ပါမယ်။

JavaScript - promise.js

```javascript
function A() {
  return "Function A";
}

function B() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("Function B");
    }, 2000);
  });
}

function C() {
  return "Function C";
}
```

```
console.log(A());
console.log(B());
console.log(C());
```

Function B က Promise Object တစ်ခုကို ပြန်ပေးထားပါတယ်။ ဒါကြောင့် ဒီကုဒ်ကို Run ကြည့်ရင် ရလဒ်က အခုလိုဖြစ်နေမှာပါ။

```
node promise.js

Function A
Promise { <pending> }
Function C
```

Function B ကိုအလုပ်လုပ်တာ မပြီးသေးလို့ Pending ဖြစ်နေပေမယ့် Function C က ဆက်အလုပ်လုပ်သွားပါတယ်။ Function B အလုပ်လုပ်လို့ ပြီးသွားတဲ့အခါ ဘာဆက်လုပ်ရမလဲ မပြောရသေးပါဘူး။

Function B ထဲက Promise ကုဒ်ကို လေ့လာကြည့်ပါ။ `resolve` နဲ့ `reject` လို့ခေါ်တဲ့ Parameter နှစ်ခုပါဝင်တာကို တွေ့ရပါလိမ့်မယ်။ ဒီနှစ်ခုလုံးက Function တွေဖြစ်ကြပါတယ်။ Success ဖြစ်ရင် `resolve()` ကို သုံးပြီး Fail ဖြစ်ခဲ့ရင် `reject()` ကို သုံးရတာပါ။ ရှုပ်ထွေးပြီး မျက်စိလည်စရာကိစ္စဖြစ်နေလို့ `reject` အကြောင်းကို မပြောသေးဘဲ ချန်ထားချင်ပါတယ်။ `resolve` အကြောင်းပဲ ကြည့်ကြပါမယ်။

နမူနာအရ Function B ကို ခေါ်လိုက်ရင် Promise Object ကို ပြန်ရမှာပါ။ ဒီ Object ပေါ်မှာ Success ဖြစ်ရင်လုပ်ရမယ့် အလုပ်ကို `then()` နဲ့ ပေးနိုင်ပါတယ်။ <u>`then()` နဲ့ ပေးလိုက်တဲ့ Function ဟာ Promise ရဲ့ `resolve()` ဖြစ်သွားမှာပါ</u>။ ဒါကြောင့် အခုလို ပြင်စမ်းကြည့်ရမှာပါ။

JavaScript - promise.js

```
console.log(A());
B().then(res => console.log(res));
console.log(C());
```

`then()` နဲ့ ပေးလိုက်တဲ့ Functionဟာ Promise ရဲ့ Resolve ဖြစ်သွားပါတယ်။ ဒါကြောင့် လုပ်စရာရှိတဲ့ အလုပ်တွေ လုပ်ပြီးတဲ့တော့မှ ရလဒ်ကို Output အနေနဲ့ ရိုက်ထုတ်ပေးမှာဖြစ်လို့ စမ်းကြည့်ရင် အခုလိုရပါလိမ့်မယ်။

```
node promise.js

Function A
Function C
Function B
```

Promise မပါဘဲ ရေးထားတဲ့ ရိုးရိုး `setTimeout` ကုဒ်နဲ့ ရလဒ်တူသွားတာပါ။ ဒါများ ဘာထူးလို့လည်း။ ထူးပါတယ်။ လိုအပ်ရင် အခုလို ပြင်ရေးလို့ရသွားပါတယ်။

JavaScript - promise.js

```
console.log(A());

B().then(res => {
  console.log(res);
  console.log(C());
});
```

`C` ကို ကြိုမလုပ်ခိုင်းတော့ဘဲ `B` အလုပ်လုပ်လို့ ပြီးမှသာ ဆက်လုပ်ဖို့ ရေးပေးလိုက်တာပါ။ ဒါကြောင့် ရလဒ်က အခုလို အစီအစဉ်အတိုင်း ဖြစ်သွားပါလိမ့်မယ်။

```
node promise.js

Function A
Function B
Function C
```

အပေါ်ယံကြည့်ရင် ဘာမှမထူးဘူးလို့ ထင်ရပေမယ့် ပြိုင်တူအလုပ်လုပ်တဲ့ Asynchronous ကုဒ်တွေရေးတဲ့ အခါ အခုလို စောင့်သင့်တဲ့အလုပ်ကိုစောင့်ပြီး၊ စောင့်စရာမလိုတဲ့အလုပ်တွေကို မစောင့်ဘဲ အလုပ်လုပ် အောင်၊ စီမံရေးသားလို့ရခြင်းဟာ အလွန်အသုံးဝင်တဲ့ လုပ်ဆောင်ချက်ဖြစ်ပါတယ်။

Reject စမ်းကြည့်ချင်ရင် Function B ကို အခုလို ပြင်ရေးလိုက်လို့ ရပါတယ်။

JavaScript - promise.js

```javascript
function B(ok) {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if(ok) {
        resolve("Function B OK");
      } else {
        reject("Function B Fails");
      }
    }, 2000);
  });
}
```

`ok` က `true` ဖြစ်ရင် `resolve` လုပ်ပေးပြီး `false` ဖြစ်ရင် `reject` လုပ်ပေးမှာပါ။ အခုလိုစမ်းကြည့် နိုင်ပါတယ်။

JavaScript - promise.js

```
B(true)
  .then(res => console.log(res))
  .catch(rej => console.log(rej));      // Function B OK

B(false)
  .then(res => console.log(res))
  .catch(rej => console.log(rej));      // Function B Fails
```

then() အတွက်ပေးလိုက်တဲ့ Function ကို resolve အဖြစ်သုံးပြီး၊ catch() အတွက်ပေးလိုက်တဲ့ Function ကို reject အဖြစ် သုံးလိုက်တာပါ။ ဒီနည်းနဲ့ Success ဖြစ်ရင် ဘာလုပ်ရမလဲ၊ Fail ဖြစ်ရင် ဘာလုပ်ရမလဲ ဆိုတာကို အသုံးပြုစဉ်မှာ ပြောလိုက်လို့ရသွားပါတယ်။

async / await

async / await ဆိုတာ အခြေခံအားဖြင့် Promise သုံးနည်း နောက်တစ်နည်းပါပဲ။ စောစောက then() နဲ့ ရေးလိုက်တဲ့ ကုဒ်ကို အခုလို ရေးလို့လည်း ရပါတယ်။

JavaScript - async.js

```
A();
await B();
C();
```

B ကို await ခံပြီးရေးထားတဲ့အတွက် B အလုပ်လုပ်လို့ ပြီးမှသာ C ကို ဆက်လုပ်သွားမှာဖြစ်ပါတယ်။ then() တွေ catch() တွေနဲ့ ရေးရတာထက်စာရင် အများကြီးပိုရှင်းသွားတာပါ။ ဒါပေမယ့် await Statement ဟာ async Function အတွင်းမှာပဲ သုံးခွင့်ရှိပါတယ်။ နောက်ဆိုရင် async Function မပါဘဲ ရေးလို့ရအောင် လုပ်နေကြပေမယ့် အခြေခံမူအားဖြင့် await ဆိုတာ async Function အတွင်းမှာပဲ သုံးရတာပါ။ ဒါကြောင့် စမ်းလက်စကုဒ်ကို အခုလို ပြင်ပြီး စမ်းကြည့်လို့ရပါတယ်။

JavaScript - async.js

```
async function app() {
  console.log(A());
  console.log(await B());
  console.log(C());
}

app();
```

မူလ Function A, B, C တို့ကို ပြောင်းစရာမလိုပါဘူး။ ခေါ်သုံးပုံပဲ ပြောင်းသွားတာပါ။ အားလုံးကို `async` Function တစ်ခုထဲမှာ ခေါ်သုံးထားပါတယ်။ `await B()` လို့ပြောထားတဲ့အတွက် စမ်းကြည့်လိုက်ရင် ရလဒ်က အခုလိုဖြစ်မှာပါ။

```
node async.js

Function A
Function B
Function C
```

B အလုပ်လုပ်လို့ပြီးအောင် စောင့်ပြီးမှ `C` က ဆက်အလုပ်လုပ်သွားတာပါ။ `async / await` မှာ `catch` လိုမျိုး Reject ဖြစ်ခဲ့ရင် ဘာလုပ်ရမလဲဆိုတာတော့ မပါပါဘူး။ ဒါကြောင့် လိုအပ်ရင် ဒီလိုရေးလို့ရပါတယ်။

JavaScript - async.js

```
async function app() {
  console.log(A());

  try {
    console.log(await B());
  } catch (e) {
    console.log(e);
  }
```

```
  console.log(C());
}
```

`try … catch` Statement ကို သုံးပြီး `await` Statement ကို ရေးလိုက်တာပါ။ ဒါကြောင့် Resolve ဖြစ်ရင် ပုံမှန်အတိုင်းဆက်အလုပ်လုပ်ပြီး Reject ဖြစ်ခဲ့ရင် `catch` Statement အလုပ်လုပ်သွားမှာ ဖြစ်ပါတယ်။

`async` Function ကို သပ်သပ်ရေးပြီးမှ ပြန်ခေါ်တာမျိုး မလုပ်ချင်ရင်လည်း `IIFE` ရေးနည်းကို သုံးလိုက်လို့ရနိုင်ပါတယ်။ ဒီလိုပါ –

JavaScript - async.js

```
(async () => {
  console.log(A());
  console.log(await B());
  console.log(C());
})();
```

Function ရေးပြီးမှ ပြန်ခေါ်သုံးတာမျိုး မဟုတ်တော့ဘဲ၊ တစ်ခါထဲရေး တစ်ခါထဲ Run လိုက်တာပါ။

Asynchronous Programming က ခက်ပါတယ်။ ပေးထားတဲ့ကုဒ်လေးတွေ ချရေးကြည့်၊ စမ်းကြည့်ရင်းနဲ့ သူ့သဘောကို ကောင်းကောင်းနားလည်အောင် ကြိုးစားရမှာပါ။ ဒီသဘာသဘာဝကို သိထားမှ ရေရှည်အတွက် ကောင်းမှာမို့လို့ အခုလို အခန်းတစ်ခန်းယူပြီး ရှည်ရှည်ဝေးဝေး ရှင်းပြနေပေမယ့် လက်တွေ့အသုံးပြုတဲ့အခါ ဒီလောက်တော့လည်း မခက်လှဘူးဆိုတာကို တွေ့ရပါလိမ့်မယ်။

အခန်း (၁၀) - Database - SQLite, Prisma ORM

အရင်က Rockstar Developer စာအုပ်နဲ့ သင်ခန်းစာတွေမှာ Database အတွက် NoSQL Database နည်းပညာတစ်ခုဖြစ်တဲ့ MongoDB ကို အစဉ်အဆက် အသုံးပြုခဲ့ပါတယ်။ အစပိုင်းမှာ အနာဂါတ်နည်းပညာတစ်ခုရယ်လို့ အရမ်းဟော့နေတဲ့အတွက် မဖြစ်မနေသိထားသင့်လို့ အသုံးပြုခဲ့တာပါ။ နောက်ပိုင်း လူပြောနည်းလာပေမယ့် သူ့နေရာနဲ့သူအသုံးဝင်ဆဲဖြစ်လို့ MongoDB ကိုပဲ ဆက်လက်အသုံးပြုခဲ့ပါသေးတယ်။

ဒီစာအုပ်မှာတော့ SQLite ကို ပြောင်းလဲအသုံးပြု ဖော်ပြသွားမှာပါ။ ဘာဖြစ်လို့လဲဆိုတော့ အစောပိုင်းမှာ မတူကွဲပြားတဲ့ နည်းပညာတစ်ခုအနေနဲ့ အစပြုခဲ့ပေမယ့် လက်ရှိအချိန်မှာ MongoDB ကိုလည်း Schema တွေ Relationship တွေနဲ့ ရိုးရိုး Relational Database တစ်ခုဆန်ဆန် သုံးလာကြပါတယ်။ Relational Database နည်းပညာတွေကလည်း MongoDB မှာလို Unstructured Data တွေကို စီမံနိုင်လာကြပါတယ်။ တစ်မျိုးရဲ့ အားသာချက်ကို နောက်တစ်မျိုးမှာလည်း ရနိုင်လာပြီဖြစ်လို့ NoSQL နဲ့ RDBMS ဆိုတာ ဟိုးအရင်တုံးကလို ဘာမှမဆိုင်တဲ့ ဆန့်ကျင်ဘက်နည်းပညာတွေ မဟုတ်ကြတော့ပါဘူး။

ပြီးတော့ ORM (Object Relational Mapping) နည်းပညာတစ်ခုကို ကြားခံထားပြီး လေ့လာသွားကြမှာဖြစ်လို့ နောက်ကွယ်က Database က သိပ်ပြဿနာ မဟုတ်တော့ပါဘူး။ Database ကို တိုက်ရိုက်ကိုင်တွယ်စရာလည်း သိပ်လိုမှာ မဟုတ်ပါဘူး။ နောက်ကွယ်က Database က ဘာပဲဖြစ်ဖြစ် လေ့လာမှုပုံစံက အတူတူပဲ ဖြစ်သွားမှာပါ။ အရွေးမှားသွားရင် ဘယ်လိုလုပ်မလဲ ဆိုတာမျိုးကိုလည်း သိပ်စိတ်ပူစရာ မလိုပါဘူး။ ဥပမာ – ပရောဂျက်စခါစမှာ SQLite လို ပေါ့ပါးလွယ်ကူတဲ့ Database လေးနဲ့စလိုက်ပြီး နောက်မှ PostgreSQL (သို့မဟုတ်) MongoDB ပြောင်းသုံးဖို့လိုအပ်လာတယ် ဆိုတာမျိုး ဖြစ်လာရင်လည်း၊ ရေးထားတဲ့ ကုဒ်တွေကို သိပ်အများကြီးပြင်စရာမလိုဘဲ အလွယ်တကူ ပြောင်းသုံးလို့ရနိုင်မှာ ဖြစ်ပါ

တယ်။ <u>လိုအပ်တဲ့ အပြောင်းအလဲ Migration ကို ORM က လုပ်ပေးနိုင်တဲ့အတွက် ဖြစ်ပါတယ်</u>။

ဒါကြောင့် စခါစမှာ သိပ်ခေါင်းစားစရာမလိုဘဲ ပေါ့ပါးလွယ်ကူတာနဲ့ စချင်တဲ့အတွက် SQLite ကိုအသုံးပြုဖို့ ရွေးချယ်တာပါ။ SQLite ဟာ ဖိုင်ကိုအခြေခံတဲ့ Standalone Database ဖြစ်လို့ Server Installation တွေ Setup တွေလုပ်စရာ မလိုအပ်ပါဘူး။ Database ဖိုင်ကို ပရောဂျက်ထဲမှာထားပြီး ချိတ်ဆက်အသုံးပြုလိုက် ယုံပါပဲ။ တော်တော်လေး ရိုးရှင်းအဆင်ပြေပါတယ်။

နောက်ပြီးတော့ ပရောဂျက်စခါစမှာ SQLite နဲ့စလိုက်မယ်လို့သာ ပြောတာပါ၊ အခုဒီစာရေးနေစဉ် ကာလမှာ SQLIte ကို User သန်းပေါင်းများစွာ အသုံးပြုနေကြတဲ့ Production ပရောဂျက်တွေထိ လက်တွေ့သုံးလို့ရတယ်ဆိုတာကို သုံးပြတဲ့သူတွေ ရှိနေပါတယ်။ Cloud Database တွေရဲ့ ဈေးနှုန်းကြီးမြင့်မှု၊ လိုအပ်တာထက် ပိုမိုရှုပ်ထွေးနေမှုတွေကို လူတွေစိတ်ကုန်နေကြချိန်နဲ့ SSD Storage တွေ ဈေးသက်သာချိန် တိုက်ဆိုင်ပြီး၊ အခုဆိုရင် SQLite ကို Production မှာသုံးဖို့ထိ စိတ်ဝင်စားမှု များနေပါတယ်။

SQLite ကို Prisma ORM နည်းပညာနဲ့ တွဲသုံးကြပါမယ်။ စာဖတ်သူက SQL Query တွေ သိထားပြီးဖြစ် လိမ့်မယ်လို့ ယူဆထားပါတယ်။ ဒါကြောင့် SQL Query တွေ တိုက်ရိုက်ရေးပြီး အခြေခံတွေ ပြန်လေ့လာမယ့်အစား လိုချင်တဲ့ရလဒ်ကို ပိုမြန်မြန်ရနိုင်တဲ့ Prisma ORM ကို အသုံးပြုဖို့ ရွေးချယ်ထားတာပါ။

Prisma ORM

Prisma ဟာ အသုံးပြုရလွယ်ကူပြီး MySQL, PostgreSQL, MSSQL, SQLite, MongoDB စသည်ဖြင့် Database နည်းပညာအမျိုးမျိုးနဲ့ ပူးတွဲအသုံးပြုနိုင်ပါတယ်။ စမ်းသပ်လေ့လာနိုင်ဖို့အတွက် နှစ်သက်ရာ အမည်နဲ့ ဖိုဒါအသစ်တစ်ခု ဆောက်လိုက်ပါ။ ပြီးတဲ့အခါ အခုလို Install လုပ်ပြီး စမ်းကြည့်လို့ရပါတယ်။

```
npm i prisma
```

Install လုပ်ပြီးရင် Prisma ကို အခုလို အစပြုပေးရပါတယ်။

```
npx prisma init --datasource-provider sqlite

✔ Your Prisma schema was created at prisma/schema.prisma
  You can now open it in your favorite editor.
...
```

နမူနာမှာ Datasource အဖြစ် sqlite ကို သတ်မှတ်ပေးထားပါတယ်။ ဒီ Command ကို Run လိုက်တဲ့အခါ သူ့ Output မှာပြောထားတဲ့အတိုင်း `prisma/schema.prisma` ဆိုတဲ့ ဖိုင်တစ်ခုနဲ့အတူ `.env` ဖိုင်တစ်ခုလည်း ရရှိသွားပါလိမ့်မယ်။ `.env` ဖိုင်ကို ဖွင့်ကြည့်လိုက်ရင် အခုလို တွေ့ရမှာပါ။

.env

```
DATABASE_URL="file:./dev.db"
```

Database ဖိုင်တဲ့ တည်နေရာအဖြစ် `prisma` ဖိုဒါထဲမှာ `dev.db` အမည်ကို သတ်မှတ်ပေးထားတာဖြစ်ပါတယ်။ မိမိနှစ်သာရာတည်နေရာနဲ့ အမည်ကို ပေးမယ်ဆိုရင် ပေးလို့ရပါတယ်။

`prisma/schema.prisma` ဖိုင်ကို ဖွင့်ကြည့်ရင်တော့ အခုလို တွေ့ရမှာပါ။

Prisma - prisma/schema.prisma

```
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema

generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "sqlite"
  url      = env("DATABASE_URL")
}
```

datasource အဖြစ် sqlite နဲ့အတူ env() Function နဲ့ .env ဖိုင်ကိုဖတ်ပြီး Database ဖိုင် တည်နေရာကို သတ်မှတ်ထားပါတယ်။ စမ်းကြည့်နိုင်ဖို့အတွက် အဲ့ဒီဖိုင်ထဲမှာ အခုလို ရေးပေးလိုက်ပါ။

Prisma - schema.prisma

```
model User {
  id     Int     @id @default(autoincrement())
  name   String
  bio    String?
  posts Post[]
}

model Post {
  id       Int    @id @default(autoincrement())
  content String
  user     User   @relation(fields: [userId], references: [id])
  userId   Int
}
```

User နဲ့ Post ဆိုတဲ့ Model နှစ်ခု သတ်မှတ်ပေးလိုက်တာပါ။ Table Relationship အကြောင်း အခြေခံ လောက် သိတယ်ဆိုရင် အများကြီးရှင်းပြနေစရာမလိုဘဲ ရေးထားတဲ့ ကုဒ်ကို ဖတ်ကြည့်ယုံနဲ့တင် အဓိပ္ပါယ် ပေါ်လွင်နိုင်ပါတယ်။ ထူးခြားချက် တချို့တော့ ရှိနေပါတယ်။

User Model မှာ id, name, bio, posts ဆိုတဲ့ အမည်တွေနဲ့ Field (၄) ခုပါဝင်ပါတယ်။ id ဟာ Field Type Integer ဖြစ်ပြီး @id Function နဲ့ Primary Key ဖြစ်ကြောင်း သတ်မှတ်ထားပါတယ်။ @default Function နဲ့ Auto Increment လည်း သတ်မှတ်ပေးထားပါတယ်။

name ကတော့ Field Type String ဖြစ်ပါတယ်။ Prisma က အသုံးပြုမယ့် Database အမျိုးအစားပေါ် မူတည်ပြီး VARCHAR သို့မဟုတ် TEXT ကို သုံးပေးသွားမှာပါ။ လက်ရှိ SQLite ကိုရွေးထားတာဖြစ်လို့ name ရဲ့ Type က Database ဘက်မှာ TEXT ဖြစ်သွားမှာပါ။ bio အတွက် String? လို့သတ်မှတ်တဲ့ နေရာမှာ ? လေးပါသွားတာကို တွေ့ပါလိမ့်မယ်။ Optional ဆိုတဲ့အဓိပ္ပါယ်ဖြစ်လို့ bio မပါခဲ့ရင်လည်း လက်ခံပေးသွားမှာ ဖြစ်ပါတယ်။ Null လက်ခံလိုက်တဲ့ သဘောပါပဲ။

`posts` ကတော့ Integer တို့ String တို့လို့ Database Column Type တွေ မဟုတ်တော့ဘဲ `Post[]` ဖြစ်သွားပါတယ်။ User မှာ Post တွေအများကြီးရှိမယ်လို့ သတ်မှတ်လိုက်တာပါ။ <u>One to Many Relationship တစ်ခုကို သတ်မှတ်လိုက်တာပါ</u>။

ဒါကြောင့် Post Model ရဲ့ဖွဲ့စည်းပုံကိုလည်း ဆက်ပြီး သတ်မှတ်ပေးထားပါတယ်။ အဓိကအကျဆုံးထူးခြား ချက်က `user` Field ဖြစ်ပါတယ်။ သူလည်း Database Column Type မဟုတ်ဘဲ `User` Model ကို ပေး ထားပါတယ်။ ဒါကြောင့် Post မှာ User ရှိမယ်ဆိုတဲ့ အဓိပ္ပါယ်ဖြစ်သွားပါတယ်။ အဲ့ဒီ Relationship အတွက် လိုအပ်တဲ့ သတ်မှတ်ချက်တွေကို `@relation` Function နဲ့ ပြောပေးထားပါတယ်။ Post ရဲ့ `userId` နဲ့ User ရဲ့ `id` တို့ကို ချိတ်ဆက်ပေးရမှာပါ။

ဒီ Model Schema ကို သုံးပြီးလိုအပ်တဲ့အလုပ်တွေ လုပ်ပေးနိုင်ဖို့ အခုလို Run ရပါမယ်။

```
npx prisma migrate dev --name=init

Environment variables loaded from .env
Prisma schema loaded from prisma/schema.prisma
Datasource "db": SQLite database "dev.db" at "file:./dev.db"

Applying migration `20240624133519_init`

The following migration(s) have been created and applied from new
schema changes:

migrations/
  └─ 20240624133519_init/
    └─ migration.sql

Your database is now in sync with your schema.

✔ Generated Prisma Client (v5.15.1) to ./node_modules/@prisma/client
in 29ms
```

`migrate` Command က Schema ကိုသုံးပြီး လိုအပ်တဲ့ Table တွေဆောက်ပေးမှာပါ။ `dev` Option က လိုအပ်တဲ့ Migration SQL Query တွေကို Record လုပ်ပြီးသိမ်းပေးစေဖို့အတွက်ပါ။ နောက်ဆုံးက `--`

name Option ကိုမထည့်ရင် ပေးဖို့လာတောင်းပါလိမ့်မယ်။ မြန်သွားအောင် တစ်ခါထဲ ထည့်ပေးလိုက်တာ ပါ။ Migration Query တွေကို သိမ်းတဲ့အခါ အသုံးပြုပေးမယ့် အမည်ဖြစ်ပါတယ်။

ဒီ Command က Table တွေ Query တွေသာမက၊ Prisma Client ကိုပါ Generate လုပ်ပေးသွားမှာပါ။ သူ Generate လုပ်ပေးတဲ့ Prisma Client ကိုသုံးပြီး Data တွေကို Manage လုပ်ရမှာဖြစ်ပြီး ခဏနေမှ ဆက်ကြည့်ကြပါမယ်။

ဒီ Command ကို Run လိုက်တဲ့အခါ `prisma/migrations` ဆိုတဲ့ ဖိုဒါတစ်ခုနဲ့အတူ Migration မှတ်တမ်းတွေ သိမ်းပေးသွားတာကို တွေ့ရမှာဖြစ်ပါတယ်။ အဲ့ဒီထဲက SQL ဖိုင်ကိုဖွင့်ကြည့်လိုက်ရင် အခုလို တွေ့ရပါလိမ့်မယ်။

Output - SQL

```sql
-- CreateTable
CREATE TABLE "User" (
    "id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    "name" TEXT NOT NULL,
    "bio" TEXT
);

-- CreateTable
CREATE TABLE "Post" (
    "id" INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    "content" TEXT NOT NULL,
    "userId" INTEGER NOT NULL,
    CONSTRAINT "Post_userId_fkey" FOREIGN KEY ("userId") REFERENCES
"User" ("id") ON DELETE RESTRICT ON UPDATE CASCADE
);
```

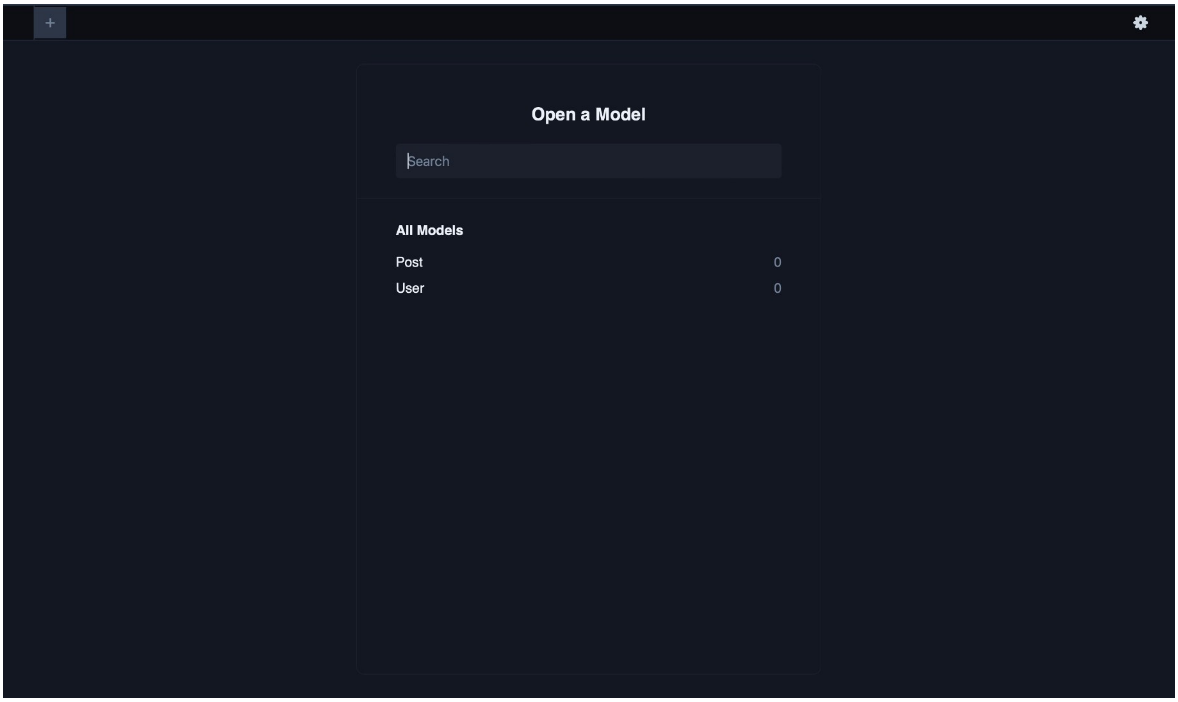
User Table နဲ့ Post Table တို့ကို တည်ဆောက်ရာမှာ အသုံးပြုသွားတဲ့ Query ဖြစ်ပါတယ်။ သတိပြု ကြည့်ရင် User Table မှာ `id`, `name`, `bio` (၃) ခုပဲပါပါတယ်။ `posts` မပါပါဘူး။ Post တွေက User မှာ ထည့်သိမ်းစရာမလိုလို့ပါ။ Post Table မှာလည်း `id`, `content`, `userId` (၃) ခုပဲပါပါတယ်။

`user` မပါပါဘူး။ ဒါပေမယ့် Table နှစ်ခုချိတ်ဆက်အလုပ်လုပ်ဖို့အတွက် Foreign Key Relationship Index ကို တစ်ခါထဲ ထည့်ပေးသွားမှာ ဖြစ်ပါတယ်။

နောက်ထပ်ထူးခြားချက်ကတော့ Model Name နဲ့ Table Name ကို အတူတူပဲ ပေးထားခြင်းဖြစ်ပါတယ်။ တချို့ ORM စနစ်တွေမှာ Naming Convention မတူတာမျိုးဖြစ်နိုင်ပါတယ်။ ဥပမာ `User` Model အတွက် Table Name က `users` ဖြစ်ပြီး `Post` Model အတွက် Table Name က `posts` ဖြစ်နိုင်တာမျိုးပါ။ Prisma မှာတော့ အဲ့ဒီလို Convention မျိုးကိုမသုံးဘဲ Model Name နဲ့ Table Name ကို အတူတူပဲ ပေးပါတယ်။ Migrate Command Run ပြီးရင် လိုအပ်တဲ့ Table တွေ တည်ဆောက်လို့ ပြီးပါပြီ။ Prisma မှာ Table တွေ Data တွေကို Manage လုပ်လို့ရတဲ့စနစ် တစ်ခါထဲ ပါဝင်ပါတယ်။ အခုလို စမ်းကြည့်နိုင်ပါတယ်။

```
npx prisma studio
```

ရလဒ်ကအခုလိုဖြစ်မှာပါ။



User နဲ့ Post Table နှစ်ခုရှိနေပြီး သူတို့မှာပါဝင်တဲ့ Column တွေ မှန်မမှန် စစ်ကြည့်နိုင်ပါတယ်။ လောလောဆယ် Data တော့ ရှိဦးမှာ မဟုတ်ပါဘူး။ Prisma နဲ့ အခြေခံ CRUD လုပ်ဆောင်ချက်တွေ ဘယ်လိုလုပ်ရလဲ ဆက်ကြည့်ကြပါမယ်။

Create

create.js အမည်နဲ့ ဖိုင်တစ်ခုတည်ဆောက်ပြီး အခုလိုရေးပေးလိုက်ပါ။

JavaScript - create.js

```javascript
const { PrismaClient } = require("@prisma/client");

const prisma = new PrismaClient();

const { PrismaClient } = require("@prisma/client");

const prisma = new PrismaClient();

prisma.user
  .create({
    data: {
      name: "Bob",
      bio: "profile bio",
      posts: {
        create: [
          { content: "First Post" },
          { content: "Second Post" },
        ],
      },
    },
  });
```

ရေးထားတဲ့ ကုဒ်မှာတင် အဓိပ္ပါယ်ပေါ်လွင်ပြီးဖြစ်ပါတယ်။ Prisma Client ကို Import လုပ်ပါတယ်။ ပြီးတဲ့အခါ `prisma.user.create()` နဲ့ User Model ထဲမှာ ထည့်ချင်တဲ့ Data တွေပေးလိုက်တာပါ။

ထူးခြားချက်ကတော့ User ကိုထည့်ရင်းနဲ့ တစ်ခါထဲ Post တွေပါ တွဲထည့်လိုက်ခြင်းပဲ ဖြစ်ပါတယ်။

`create()` အပါအဝင် Prisma Function တွေဟာ Promise ကို အသုံးပြုပြီး အလုပ်လုပ်တဲ့ Function တွေပါ။ ဒါကြောင့် Error Handling တွေဘာတွေ သေချာလုပ်ချင်ရင် အခုလိုရေးလို့ရပါတယ်။

JavaScript - create.js

```
const { PrismaClient } = require("@prisma/client");

const prisma = new PrismaClient();

prisma.user
  .create({
    data: {
      name: "Bob",
      bio: "profile bio",
      posts: {
        create: [
          { content: "First Post" },
          { content: "Second Post" },
        ],
      },
    },
  })
  .then(() => {
    console.log("Inserted User Bob with Posts");
  })
  .catch(e => {
    console.error(e);
    process.exit(1);
  })
  .finally(() => {
    prisma.$disconnect();
  });
```

`then()` နဲ့ Success ဖြစ်ရင် ဆက်လုပ်ရမယ့်အလုပ်ကို ပြောထားပါတယ်။ `catch()` နဲ့ Error ရှိခဲ့ရင် လုပ်ရမယ့် အလုပ်ကို ပြောထားပါတယ်။ `finally()` နဲ့ Success ပဲဖြစ်ဖြစ် Error ပဲဖြစ်ဖြစ် ဘာပဲ ဖြစ်ဖြစ် ဆက်လုပ်ရမယ့် အလုပ်ကို ပြောထားပါတယ်။

ရေးထားတဲ့ ကုဒ်ကို အခုလို Run ပေးလိုက်ပါ။

```
node create.js

Inserted User Bob with Posts
```

Table တွေထဲမှာ ပေးလိုက်တဲ့ Data တွေကို ထည့်ပေးသွားမှာပါ။ ပေးလိုက်တာ မပြည့်စုံရင်ဖြစ်ဖြစ်၊ မှားနေရင်ဖြစ်ဖြစ်၊ သင့်တော်တဲ့ Error တွေကို ပြပေးပါလိမ့်မယ်။

Read

Data တွေပြန်ထုတ်ယူဖို့အတွက် `findMany()`, `findFirst()`, `findUnique()` စတဲ့ Function တွေကို အသုံးပြုနိုင်ပါတယ်။ ဒီလိုပါ –

JavaScript
```javascript
prisma.user.findMany().then(data => console.log(data));
```

`async / await` နဲ့ရေးချင်ရင်တော့ ဒီလိုဖြစ်ပါလိမ့်မယ်။

JavaScript
```javascript
async function read() {
  const data = await prisma.user.findMany();
  console.log(data);
}

read();
```

နောက်တစ်ခန်းမှာ Back-end API တစ်ခုရေးသားပုံကို ဆက်လက်လေ့လာတဲ့အခါ လိုအပ်သလို ဖြည့်စွက် လေ့လာသွားကြပါမယ်။ ဒီနေရာမှာတော့ အခြေခံအသုံးပြုပုံလောက် စမ်းကြည့်ထားရင် ရပါတယ်။ `read.js` အမည်နဲ့ ဖိုင်တစ်ခုတည်ဆောက်ပြီး ဒီကုဒ်တွေကို ရေးစမ်းကြည့်ပါ။

JavaScript - read.js

```
const { PrismaClient } = require("@prisma/client");

const prisma = new PrismaClient();

prisma.user
  .findFirst({
    where: { id: 1 },
    include: { posts: true },
  })
  .then(data => console.log(data))
  .catch(e => {
    console.error(e);
    process.exit(1);
  })
  .finally(() => {
    prisma.$disconnect();
  });
```

ဒီတစ်ခါ `findFirst()` ကိုသုံးထားပြီး `where` နဲ့ Filter Option ထည့်ပေးထားပါတယ်။ အဓိကကျတဲ့ ထူးခြားချက်ကတော့ `include` Option နဲ့ Relationship Data တွေကိုပါ တစ်ခါထဲ ထုတ်ယူရရှိနိုင်ခြင်း ဖြစ်ပါတယ်။ Run ကြည့်လိုက်ရင် အခုလို ရပါလိမ့်မယ်။

```
node read.js

{
  id: 1,
  name: 'Bob',
  bio: 'profile bio',
```

```
  posts: [
    { id: 1, content: 'First Post', userId: 1 },
    { id: 2, content: 'Second Post', userId: 1 }
  ]
}
```

သက်ဆိုင်ရာ User Data နဲ့အတူ Related Posts တွေကိုလည်း တစ်ခါထဲရရှိသွားပါပြီ။

Update

Data တွေ Update လုပ်ဖို့အတွက် `update()`, `updateMany()`, `upsert()` စတဲ့ Function တွေကို အသုံးပြုနိုင်ပါတယ်။ ဒီလိုပါ။

JavaScript

```javascript
prisma.user.update({
  where: { id: 1 },
  data: { name: "Bobby" }
});
```

နမူနာအရ `User 1` ရဲ့ `name` ကို Bobby လို့ပြောင်းပေးလိုက်မှာဖြစ်ပါတယ်။ ဒီနေရာမှာ ထူးခြားပြီး အသုံးဝင်တာက `upsert()` Function ပါ။ သူက ပေးလိုက်တဲ့ `where` ကိုကြည့်ပြီး မရှိသေးရင် အသစ် ထည့်ပေးပါတယ်။ ရှိပြီးသားဆိုရင် Update လုပ်ပေးပါတယ်။ `update.js` အမည်နဲ့ ဖိုင်တစ်ခု တည်ဆောက်လိုက်ပြီး အခုလို ရေးစမ်းကြည့်ပါ။

JavaScript - update.js

```javascript
const { PrismaClient } = require("@prisma/client");

const prisma = new PrismaClient();

prisma.user
  .upsert({
    where: { id: 1 },
```

```
    update: { name: "Bobby" },
    create: { name: "Bobby", bio: "Bobby's bio" },
  })
  .then(result => {
    console.log(result);
  })
  .catch(e => {
    console.log(e);
    process.exit(1);
  })
  .finally(() => {
    prisma.$disconnect();
  });
```

နမူနာအရ `id: 1` ကို ရှာပြီး တွေ့ရင်ပြင်ရမယ့် Data နဲ့ မတွေ့ရင် ထည့်ရမယ့် Data တွေကိုပေးလိုက်တာပါ။ စမ်းကြည့်ရင် အခုလိုဖြစ်ပါလိမ့်မယ်။

```
node update.js

{ id: 1, name: 'Bobby', bio: 'profile bio' }
```

ရှိနေပြီးဖြစ်လို့ Update လုပ်ပေးသွားတာပါ။

Delete

Delete လုပ်ဆောင်ချက်အတွက် `delete.js` အမည်နဲ့ အခုလိုရေးစမ်းကြည့်ပါ။

JavaScript - delete.js

```
const { PrismaClient } = require("@prisma/client");

const prisma = new PrismaClient();
```

```
prisma.user
  .delete({
    where: { id: 1 },
  })
  .then(() => console.log("Deleted User 1"))
  .catch(e => {
    console.error(e);
    process.exit(1);
  })
  .finally(() => {
    prisma.$disconnect();
  });
```

ပြီးရင် စမ်းကြည့်လိုက်ပါ။

```
node delete.js

PrismaClientKnownRequestError: Invalid `.delete()` invocation in …
```

အဆင်မပြေပါဘူး။ User ကိုဖျက်ဖို့ကြိုးစားပေမယ့် အဲ့ဒီ User ကို Reference လုပ်ပြီးသုံးထားတဲ့ Post တွေရှိနေလို့ ဖျက်ခွင့်မပေးတာပါ။ ဖြစ်သင့်တဲ့ Error ပါပဲ။ ဒီအတိုင်းပဲ ထားသင့်ပါတယ်။ အဲ့ဒီလို Error မပေးဘဲ ဖျက်လို့ရချင်ရင် Schema မှာ အခုလို သတ်မှတ်ထားခဲ့လို့ ရပါတယ်။

Prisma

```
model Post {
  id      Int     @id @default(autoincrement())
  content String
  user    User?   @relation(fields: [userId], references: [id])
  userId  Int
}
```

User? ဖြစ်သွားပါပြီ။ ? သင်္ကေတလေး ပါသွားတာပါ။ ဒါဆိုရင်တော့ Relationship က Optional ဖြစ်သွားလို့ Reference ရှိနေလည်း ဖျက်လို့ရသွားပါလိမ့်မယ်။ နောက်တစ်နည်းကတော့ Cascade Delete ကို အသုံးပြုလိုက်ခြင်းပါပဲ။ ဒီလိုပါ –

Prisma

```
model Post {
  id      Int    @id @default(autoincrement())
  content String
  user    User @relation(fields: [userId], references: [id], onDelete: Cascade)
  userId  Int
}
```

`onDelete: Cascade` ပါသွားလို့၊ User ကို ဖျက်လိုက်ရင် User ရော သူနဲ့ဆိုင်တဲ့ Post တွေပါ အကုန်တစ်ခါထဲ ပျက်သွားမှာပါ။ ရနိုင်တဲ့ နည်းတွေကို ပြောပြတာပါ။ လောလောဆယ် ဒီနည်းတွေကို မသုံးသေးပါဘူး။ ကိုယ့်ဘာသာပဲ Manual ဖျက်လိုက်ပါမယ်။ `delete.js` ကို အခုလိုပြင်စမ်းကြည့်ပါ။

JavaScript - delete.js

```javascript
const { PrismaClient } = require("@prisma/client");

const prisma = new PrismaClient();

prisma.post.deleteMany({
    where: { userId: 1 }
}).then(() => {
    prisma.user
    .delete({
      where: { id: 1 },
    })
    .then(() => {
      console.log("Deleted User 1 and their posts");
    });
```

```
}).finally(() => {
    prisma.$disconnect();
});
```

User 1 ရဲ့ Posts တွေကို အရင်ဖျက်လိုက်ပြီးမှ User 1 ကို ဖျက်လိုက်တာပါ။ စမ်းကြည့်လိုက်ရင် အကုန်ပျက်ပြီး အလုပ်လုပ်သွားတယ်ဆိုတာကို တွေ့ရမှာဖြစ်ပါတယ်။

Back-end API

နောက်တစ်ခန်းမှာ Back-end API တစ်ခု ဖန်တီးကြမှာပါ။ Prisma ORM ကို ပူးတွဲအသုံးပြုမှာဖြစ်လို့ အခုကတည်းက လိုအပ်တဲ့ ကြိုတင်ပြင်ဆင်မှုတွေကို လုပ်ခဲ့ချင်ပါတယ်။ **`yaycha-api`** အမည်နဲ့ ပရောဂျက်ဖိုဒါသစ်တစ်ခု တည်ဆောက်လိုက်ပါ။ ပြီးတဲ့အခါ သူ့ထဲမှာ အခုလို လိုအပ်တာတွေ တန်းစီပြီး Run ပေးလိုက်ပါ။

```
npm init -y
npm i prisma @faker-js/faker bcrypt
npx prisma init --datasource-provider sqlit
```

`prisma` အပြင် `faker` နဲ့ `bcrypt` တို့ကိုလည်း ထည့်သွင်းထားပါတယ်။ ဒီနည်းပညာတွေကိုသုံးပြီး နမူနာ Sample Data တွေ တစ်ခါထဲထည့်သွားမှာမို့လို့ပါ။ ရရှိလာတဲ့ `prisma/schema.prisma` ဖိုင်ထဲမှာ ဒီကုဒ်ကို ကူးယူပြီး ထပ်တိုးပေးလိုက်ပါ။

Prisma - prisma/schema.prisma

```
model User {
    id       Int      @id @default(autoincrement())
    name     String
    username String   @unique
    bio      String?
    password String
    posts    Post[]
    comments Comment[]
```

```
    created  DateTime  @default(now())
}

model Post {
    id       Int       @id @default(autoincrement())
    content  String
    user     User      @relation(fields: [userId], references: [id])
    userId   Int
    comments Comment[]
    created  DateTime  @default(now())
}

model Comment {
    id      Int    @id @default(autoincrement())
    content String
    user    User   @relation(fields: [userId], references: [id])
    userId  Int
    post    Post   @relation(fields: [postId], references: [id])
    postId  Int
    created  DateTime  @default(now())
}
```

User, Post နဲ့ Comment ဆိုတဲ့ Model (၃) ခုဖြစ်ပါတယ်။ User မှာ လိုအပ်မယ့် `username` တွေ `password` တွေလည်း ပါသွားပါပြီ။ `username` ကို `@unique` လုပ်ပေးထားတာ သတိပြုပါ။ ပြီးတော့ User မှာ `Post[]` တွေရော `Comment[]` တွေရော Relationship အနေနဲ့ ပါဝင်ပါသေးတယ်။

Post ရဲ့ ထူးခြားချက်က သူ့မှာလည်း `Comment[]` တွေ Relationship အဖြစ် ချိတ်ဆက်ထားခြင်းဖြစ်ပါတယ်။ ဒါကြောင့် Comment အတွက် User နဲ့ရော Post နဲ့ပါ Relationship ရှိနေမှာဖြစ်ပါတယ်။ Comment Model မှာ လိုအပ်တဲ့ Relationship နှစ်မျိုးလုံးကို ထည့်ရေးပေးထားပါတယ်။

ပြီးတဲ့အခါ Migrate လုပ်ပေးလိုက်ပါ။

```
npx prisma migrate dev --name=init
```

လိုအပ်တဲ့ Model Schema တွေ Table တွေ ရသွားပါပြီ။ နမူနာ Data တချို့ တစ်ခါထဲ ထည့်လိုက်ပါမယ်။ `prisma` ဖိုဒါထဲမှာ `seeds` အမည်နဲ့ ဖိုဒါတစ်ခု ဆောက်ပေးပါ။ ပြီးတဲ့အခါ `seeds` ဖိုဒါထဲမှာ `UserSeeder.js`, `PostSeeder.js`, `CommentSeeder.js` နဲ့ `main.js` ဆိုတဲ့ ဖိုင် (၄) ခု ထပ်ဆောက်ပေးပါ။ ဒါကြောင့် ပရောဂျက်ဖိုဒါရဲ့ ဖွဲ့စည်းပုံက အခုလိုဖြစ်နေရပါမယ်။

```
yaycha-api
├── node_modules/
├── package-lock.json
├── package.json
└── prisma/
    ├── dev.db
    ├── dev.db-journal
    ├── migrations/
    ├── schema.prisma
    └── seeds/
        ├── CommentSeeder.js
        ├── PostSeeder.js
        ├── UserSeeder.js
        └── main.js
```

`UserSeeder.js` ထဲမှာ ဒီကုဒ်ကိုရေးပေးပါ။

JavaScript - prisma/seeds/UserSeeder.js

```javascript
const { PrismaClient } = require("@prisma/client");
const { faker } = require('@faker-js/faker');
const bcrypt = require("bcrypt");

const prisma = new PrismaClient();

async function UserSeeder() {
  const password = await bcrypt.hash("password", 10);
```

```
  console.log("User seeding started...");

  for (let i = 0; i < 10; i++) {
    const firstName = faker.person.firstName();
    const lastName = faker.person.lastName();

    const name = `${firstName} ${lastName}`;
    const username = `${firstName}${lastName[0]}`.toLocaleLowerCase();
    const bio = faker.person.bio();

    await prisma.user.upsert({
      where: { username },
      update: {},
      create: { name, username, bio, password },
    });
  }

  console.log("User seeding done.");

}

module.exports = { UserSeeder };
```

User အတွက် `for` Loop နဲ့ Record (၁၀) ခုထည့်ပေးတဲ့ကုဒ်ဖြစ်ပါတယ်။ `bcrypt` နဲ့ `"password"` ကို `hash` လုပ်ထားပါတယ်။ ဒါကြောင့် သိမ်းတဲ့အခါ `hash` တွေကိုသိမ်းမှာဖြစ်ပြီး User အားလုံးရဲ့ Password က `password` ပဲ ဖြစ်ပါတယ်။ `name`, `username` နဲ့ `bio` တို့ကိုတော့ `faker` ရဲ့ အကူအညီနဲ့ Random Name တွေ Random Bio တွေ အသုံးပြုထားပါတယ်။

နောက်ထပ်သတိပြုရမှာက `create()` နဲ့မထည့်ဘဲ `upsert()` နဲ့ထည့်ထားခြင်း ဖြစ်ပါတယ်။ `username` က Unique ဖြစ်တဲ့အတွက် `username` တူ ရှိနေရင် ထပ်မထည့်စေချင်လို့ ဖြစ်ပါတယ်။ `update` အတွက် `{ }` အလွတ်ပေးထားတာကို တွေ့ရပါလိမ့်မယ်။

ဆက်လက်ပြီး `PostSeeder.js` ဖိုင်မှာ ဒီကုဒ်ကို ရေးပေးပါ။

JavaScript - prisma/seeds/PostSeeder.js

```
const { PrismaClient } = require("@prisma/client");
const { faker } = require("@faker-js/faker");

const prisma = new PrismaClient();

async function PostSeeder() {
  const data = [];

  for (let i = 0; i < 20; i++) {
    const content = faker.lorem.paragraph();
    const userId = faker.number.int({ min: 1, max: 10 });

    data.push({ content, userId });
  }

  console.log("Post seeding started...");
  await prisma.post.createMany({ data });
  console.log("Post seeding done.");
}

module.exports = { PostSeeder };
```

အလားတူပဲ `for` Loop နဲ့ Post Record (၂၀) ထည့်ထားတာပါ။ `content` အတွက် `faker` Paragraph ကို သုံးထားပြီး `userId` အတွက် Random 1-10 ကို ပေးထားပါတယ်။ စောစောက User (၁၀) ခုထည့်ခဲ့တာနဲ့ ချိတ်ဆက်သွားစေဖို့ပါ။

ဆက်လက်ပြီး `CommentSeeder.js` မှာ အခုလို ရေးပေးပါ။

JavaScript - prisma/seeds/CommentSeeder.js

```
const { PrismaClient } = require("@prisma/client");
const { faker } = require("@faker-js/faker");

const prisma = new PrismaClient();

async function CommentSeeder() {
  const data = [];

  for (let i = 0; i < 40; i++) {
    const content = faker.lorem.paragraph();
    const userId = faker.number.int({ min: 1, max: 10 });
    const postId = faker.number.int({ min: 1, max: 20 });

    data.push({ content, userId, postId });
  }

  console.log("Comment seeding started...");
  await prisma.comment.createMany({ data });
  console.log("Comment seeding done.");
}

module.exports = { CommentSeeder };
```

သူလည်း ပုံစံတူရေးနည်းနဲ့ Comment အခု (၄၀) ထည့်ပေးတဲ့ကုဒ်ဖြစ်ပါတယ်။ ဒီကုဒ်တွေအားလုံးကို ချိတ်ဆက်အလုပ်လုပ်တဲ့ ကုဒ်ကို `main.js` မှာ အခုလိုရေးပေးပါ။

JavaScript - prisma/seeds/main.js

```
const { PrismaClient } = require("@prisma/client");

const prisma = new PrismaClient();

const { UserSeeder } = require("./UserSeeder");
```

```
const { PostSeeder } = require("./PostSeeder");
const { CommentSeeder } = require("./CommentSeeder");

async function main() {
  try {
    await UserSeeder();
    await PostSeeder();
    await CommentSeeder();
  } catch (e) {
    console.error(e);
    process.exit(1);
  } finally {
    await prisma.$disconnect();
  }
}

main();
```

ထပ်တိုးလုပ်ဆောင်ချက်တွေ မပါပါဘူး။ ရေးပြီးသား `UserSeeder`, `PostSeeder` နဲ့ `CommentSeeder` တို့ကို ချိတ်ဆက်ပြီး Run ပေးလိုက်တာပါ။ ဒီကုဒ်တွေကို Prisma က Run ပေးနိုင်အောင် `package.json` မှာ အခုလို ထည့်သွင်းပေးဖို့ လိုအပ်ပါတယ်။

JSON - package.json

```json
"prisma": {
  "seed": "node prisma/seeds/main.js"
},
```

ဒါဆိုရင် Prisma က Sample Data ထည့်ဖို့အတွက် ဘယ်ကုဒ်ကို Run ရမလဲ သိသွားပါပြီ။ နည်းလမ်းနှစ်မျိုးနဲ့ Run နိုင်ပါတယ်။

```
npx prisma db seed
```

(သို့မဟုတ်)

```
npx prisma migrate reset
```

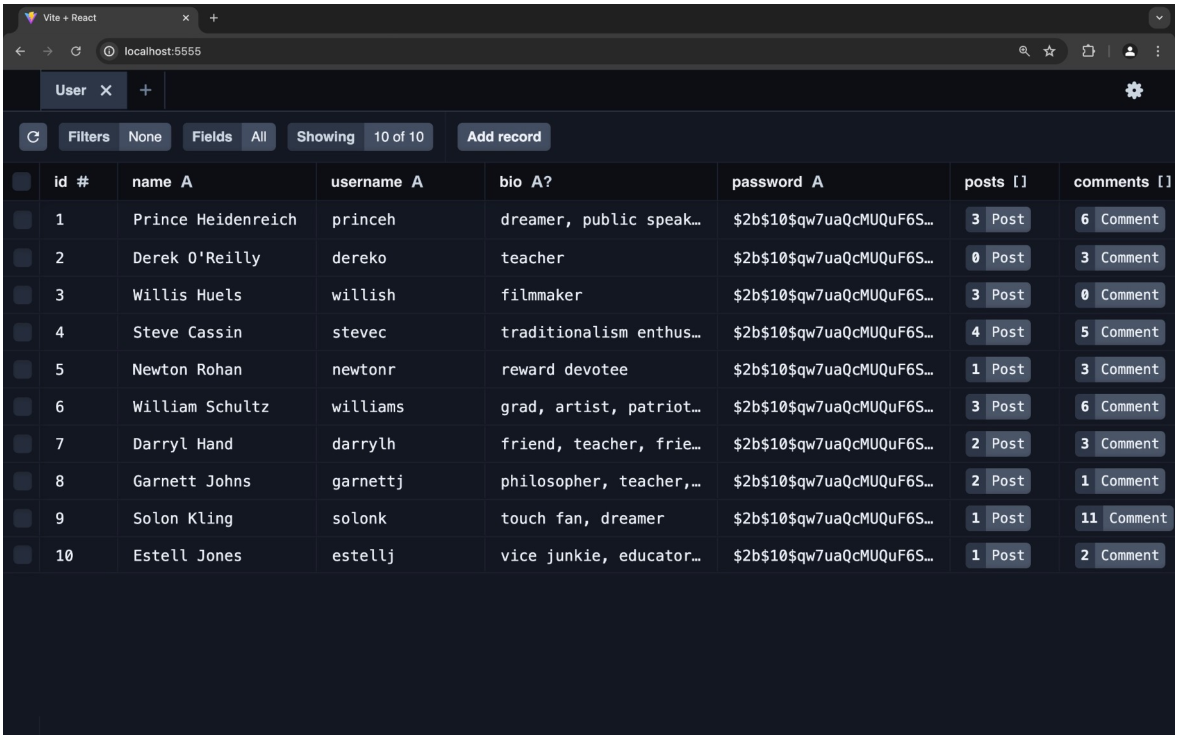
`db seed` က Seed ကုဒ်တွေကိုပဲ Run ပြီး Sample Data တွေကို ထည့်ပေးသွားမှာပါ။ `migrate reset` ကတော့ Migration တွေကိုပါ အစအဆုံးပြန် Run ပြီး Seed ကုဒ်တွေကိုလည်း တစ်ခါထဲထည့် Run ပေးသွားမှာပါ။ `migrate reset` က ပိုအသုံးဝင်ပါတယ်။ ပရောဂျက်ကို စမ်းချင်သလို စိတ်တိုင်းကျစမ်းပြီး `migrate reset` Run လိုက်တဲ့အခါ Table တွေ Sample Data တွေအားလုံး မူလအတိုင်း အသစ်ပြန်ဖြစ်သွားမှာဖြစ်လို့ အသုံးဝင်တာပါ။ ဒါကြောင့် Run ကြည့်လိုက်ပါ။

```
npx prisma migrate reset

Environment variables loaded from .env
Prisma schema loaded from prisma/schema.prisma
Datasource "db": SQLite database "dev.db" at "file:./dev.db"

✔ Are you sure you want to reset your database? All data will be lost.
… yes

Applying migration `20240625025853_init`

Database reset successful

The following migration(s) have been applied:

migrations/
  └─ 20240625025853_init/
    └─ migration.sql

✔ Generated Prisma Client (v5.15.1) to ./node_modules/@prisma/client
in 30ms

Running seed command `node prisma/seeds/main.js` ...
User seeding started...
User seeding done.
Post seeding started...
Post seeding done.
```

```
Comment seeding started...
Comment seeding done.

🌱  The seed command has been executed.
```

Reset လုပ်မှာ သေချာလားလို့ Confirmation တောင်းပါလိမ့်မယ်။ Yes လို့ ပြောပေးလိုက်ရင် Migration တွေရော Sample Data ထည့်တဲ့ Seed ကုဒ်တွေကိုပါ အကုန် Run ပေးသွားမှာပဲဖြစ်ပါတယ်။ ဒါကြောင့် လိုအပ်တဲ့ Table တွေရော Data တွေရော ရသွားပါပြီ။

```
npx prisma studio
```



id #	name A	username A	bio A?	password A	posts []	comments []
1	Prince Heidenreich	princeh	dreamer, public speak…	$2b$10$qw7uaQcMUQuF6S…	3 Post	6 Comment
2	Derek O'Reilly	dereko	teacher	$2b$10$qw7uaQcMUQuF6S…	0 Post	3 Comment
3	Willis Huels	willish	filmmaker	$2b$10$qw7uaQcMUQuF6S…	3 Post	0 Comment
4	Steve Cassin	stevec	traditionalism enthus…	$2b$10$qw7uaQcMUQuF6S…	4 Post	5 Comment
5	Newton Rohan	newtonr	reward devotee	$2b$10$qw7uaQcMUQuF6S…	1 Post	3 Comment
6	William Schultz	williams	grad, artist, patriot…	$2b$10$qw7uaQcMUQuF6S…	3 Post	6 Comment
7	Darryl Hand	darrylh	friend, teacher, frie…	$2b$10$qw7uaQcMUQuF6S…	2 Post	3 Comment
8	Garnett Johns	garnettj	philosopher, teacher,…	$2b$10$qw7uaQcMUQuF6S…	2 Post	1 Comment
9	Solon Kling	solonk	touch fan, dreamer	$2b$10$qw7uaQcMUQuF6S…	1 Post	11 Comment
10	Estell Jones	estellj	vice junkie, educator…	$2b$10$qw7uaQcMUQuF6S…	1 Post	2 Comment

Prisma Studio မှာ Table Structure တွေ Data တွေမှန်မမှန် ပြန်စစ်ကြည့်ပါ။ အကယ်၍ အမှား တစ်စုံတစ်ရာပါသွားခဲ့ရင်လည်း Schema တွေ Seed တွေကို လိုသလိုပြင်ပြီး `migrate reset` ကို ပြန် Run ပေးလိုက်ယုံပဲဖြစ်ပါတယ်။

အခန်း (၁၁) - Express - Back-end API

Express ဟာ သူ့အမည်အတိုင်းပဲ Back-end API တွေကို လွယ်လွယ်ကူကူနဲ့ အလျှင်အမြန် ဖန်တီးလို့ရတဲ့ နည်းပညာပါ။ နောက်ပိုင်းမှာ သူ့ထက်ပိုခေတ်မီပြီး ပိုပြည့်စုံတဲ့ နည်းပညာတွေ တစ်ခုပြီးတစ်ခု ထွက်ပေါ်လာပေမယ့်၊ အခုချိန်ထိ Back-end API တစ်ခုဖန်တီးချင်တယ်ဆိုရင် Express က အသင့်တော်ဆုံး နည်းပညာ ဖြစ်နေဆဲပါပဲ။ Express နဲ့ API Service တစ်ခုကို အခုလို အလျှင်အမြန်ဖန်တီးနိုင်ပါတယ်။

JavaScript

```javascript
const express = require("express");
const app = express();

app.get("/info", (req, res) => {
  res.json({ msg: "Express API" });
});

app.listen(8000, () => {
  console.log("Express API started at 8000...");
});
```

ဒါပါပဲ။ သူ့ထက် တိုတာ ရှင်းတာတော့ မရှိတော့ပါဘူး။ Express ကို Import လုပ် Run လိုက်ပြီး app Object တစ်ခုတည်ဆောက်ပါတယ်။ `/info` အမည်နဲ့ URL Route တစ်ခု သတ်မှတ်လိုက်ပြီး `listen(8000)` နဲ့ Port နံပါတ် 8000 မှာ API Server တစ်ခု Run လိုက်တာပါပဲ။

အခေါ်အဝေါ်အမှန်ကတော့ Run နေတဲ့ Express Program ကို Service လို့ခေါ်ပြီး၊ သူ့ကိုဆက်သွယ်လို့ရတဲ့ `/info` ဆိုတဲ့ URL ကို API လို့ခေါ်တာပါ။ အားလုံးကို ပေါင်းပြီး API လို့ပဲ ခေါ်ကြ ပြောကြလေ့ရှိပါတယ်။ သိသင့်တာလေးတချို့ ထည့်ကြည့်ကြပါမယ်။

API Route တွေ သတ်မှတ်တဲ့အခါ `GET`, `POST`, `PUT`, `PATCH`, `DELETE` စတဲ့ HTTP Method တွေကို ကိုယ်စားပြုတဲ့ ဒီ Method တွေ အသုံးများမှာ ဖြစ်ပါတယ်။

- `app.get()`
- `app.post()`
- `app.put()`
- `app.patch()`
- `app.delete()`

`app.get()` နဲ့ သတ်မှတ်ထားတဲ့ Route ကို ဆက်သွယ်ချင်ရင် HTTP GET Method နဲ့ ဆက်သွယ်ရမှာဖြစ်ပြီး `app.post()` နဲ့ သတ်မှတ်ထားတဲ့ Route ကို ဆက်သွယ်ချင်ရင် HTTP POST Method နဲ့ ဆက်သွယ်ရမှာပါ။ အချက်အလက်တွေ ရယူတဲ့အလုပ်တွေအတွက် `get()` ကိုသုံးရပြီး အသစ်ထည့်သွင်းတဲ့အလုပ်တွေအတွက် `post()` ကို သုံးရပါတယ်။ ပြင်ဆင်မှုတွေ ပြုလုပ်လိုတဲ့အခါ `put()` သို့မဟုတ် `patch()` ကိုသုံးကြပါတယ်။ ပြန်ဖျက်တဲ့အလုပ်တွေအတွက် `delete()` ကို သုံးရပါတယ်။

API အကြောင်း ဒီထက်ပိုအသေးစိတ်ကို API လိုတိုရှင်း စာအုပ်မှာ လေ့လာနိုင်ပါတယ်။ အောက်က လိပ်စာမှာ ရယူပါ။ ဒီနေရာမှာတော့ လိုသလောက်ကိုပဲ ခပ်မြန်မြန်ရွေးထုတ်ပြီး ပြောလိုက်တာပါ။

https://eimaung.com/api

ဒါတွေက မဖြစ်မနေလိုက်နာရမယ့် သတ်မှတ်ချက်တွေ မဟုတ်ဘဲ အလေ့အကျင့်ကောင်းအနေနဲ့ လိုက်နာကြတဲ့ Convention တွေပါ။ ဥပမာ `get()` ကိုသုံးပြီး ဖျက်တဲ့အလုပ်တွေ လုပ်ဖို့ရေးထားတဲ့ ရေးတဲ့အတိုင်း အလုပ်လုပ်ပေးသွားမှာပါပဲ။ ဒါပေမယ့် သူ့နေရာနဲ့သူ မှန်အောင်သုံးရင် ပိုပြီးအသုံးပြုလို့ကောင်းတဲ့ API တွေကိုရရှိမှာ ဖြစ်ပါတယ်။

URL Route တွေသတ်မှတ်တဲ့အခါ Dynamic Route တွေလည်း သတ်မှတ်နိုင်ပါတယ်။ ဒီလိုပါ။

JavaScript

```
app.get("/users/:id", (req, res) => {
  const { id } = req.params;

  res.json({ user: `User ${id}` });
});
```

နမူနာအရ URL မှာ `:id` ခေါ် Dynamic Parameter ပါသွားတာပါ။ `/users/1` ဆိုရင် `:id` 1 ဖြစ်ပြီး `/users/99` ဆိုရင် `:id` 99 ဖြစ်ပါတယ်။ `:id` နေရာမှာ ကြိုက်တဲ့ Value ကို ပေးလို့ရတဲ့သဘော ဖြစ်သွားတာပါ။ `:id` အပါအဝင် URL မှာပါလာတဲ့ Dynamic Parameter Value တွေကို လိုချင်ရင် နမူနာမှာ ပြထားသလို `req.params` ကနေ ယူလို့ရပါတယ်။

Response နဲ့ပက်သက်ရင် အသုံးများမှာက –

- `res.json()`
- `res.status().json()`
- `res.sendStatus()`

တို့ဖြစ်ပါတယ်။ JSON ကို Standard Data Format အနေနဲ့ သုံးကြလို့ အများအားဖြင့် API က Data တွေ ပြန်ပေးချင်တဲ့အခါ `res.json({ })` နဲ့ပဲ ပေးတာများမှာပါ။ ရံဖန်ရံခါ လိုအပ်ရင်တော့ Status Code တွေ ထည့်သတ်မှတ်ပြီး ပေးလို့ရပါတယ်။ ဥပမာ – `res.status(201).json({ })` ဆိုရင် Response Code 201 နဲ့အတူ Data ကို ပြန်ပေးလိုက်တာပါ။ အသုံးများတဲ့ Status Code တွေကတော့ –

- `200 OK`
- `201 Created`
- `204 No Content`
- `400 Bad Request`
- `401 Unauthorized`
- `403 Forbidden`

- `404 Not Found`
- `500 Internal Server Error`

ကျယ်ပြန့်တဲ့ အကြောင်းအရာဖြစ်လို့ အသေးစိတ်သိချင်ရင် **API လိုတိုရှင်း** စာအုပ်မှာ ဆက်လက်လေ့လာပါ။ ဒီနေရာမှာ အသုံးများတာတွေကိုပဲ ရွေးထုတ်ဖော်ပြထားပါတယ်။ 200 OK က အားလုံးအဆင်ပြေတဲ့အခါ အသုံးပြုတဲ့ Default Status Code ဖြစ်ပါတယ်။ 2 နဲ့စတဲ့ကုဒ်တွေအားလုံးက Success ကုဒ်တွေပါ။ 4 နဲ့ စတဲ့ကုဒ်တွေက Error ကုဒ်တွေပါ။ Client Error လို့ခေါ်ပါတယ်။ API ကြောင့် မဟုတ်ဘဲ Client ကြောင့်ဖြစ်တဲ့ Error တွေမှာ 4 နဲ့စတဲ့ ကုဒ်တွေကို သုံးကြပါတယ်။ 5 နဲ့စတဲ့ကုဒ်တွေကတော့ Server Error ပါ။ စာရင်းထဲမှာ တစ်ခုပဲ ပေးထားပါတယ်။ အသုံးပြုပုံကို သူ့နေရာသူ ဆက်ကြည့်ကြပါမယ်။

Project

လက်တွေ့စမ်းကြည့်လို့ရအောင် လုပ်လက်စ `yaycha-api` ပရောဂျက်ဖိုဒါထဲမှာ လိုအပ်တဲ့ Package တွေကို အခုလို Install လုပ်ပေးပါ။

```
npm i express cors nodemon
```

အဓိကအသုံးပြုချင်တာက `express` ပါ။ အသုံးလိုမှာမို့လို့ ကြိုထည့်လိုက်တဲ့ တခြား Package တွေအကြောင်းကိုတော့ သူ့နေရာနဲ့သူ ထည့်ပြောပေးသွားပါမယ်။ Package တွေထည့်ပြီးရင် `index.js` အမည်နဲ့ ဖိုင်တစ်ခု တည်ဆောက်လိုက်ပြီး အခုလို ရေးပေးပါ။

JavaScript - API - index.js

```javascript
const express = require("express");
const app = express();

const cors = require("cors");
app.use(cors());

app.get("/info", (req, res) => {
    res.json({ msg: "Yaycha API" });
});
```

```
app.listen(8000, () => {
    console.log("Yaycha API started at 8000...");
});
```

နမူနာမှာအသုံးပြုထားတဲ့ `cors` ဆိုတာ Cross-Origin Resource Sharing ရဲ့အတိုကောက်ပါ။ Security နည်းပညာတစ်ခုဖြစ်ပြီး Default အနေနဲ့ Browser တွေက Host မတူရင် API ကို ဆက်သွယ်အသုံးပြုခွင့် မပြုပါဘူး။

အခုဖန်တီးမယ့် API ကို Host မတူတဲ့ React App က လာသုံးရင် အသုံးပြုခွင့်ပေးဖို့ လိုပါတယ်။ ဥပမာ API က `localhost:8000` မှာ Run ပြီး React App က `localhost:3000` မှာ Run မှာပါ။ `localhost` ချင်းတူပေမယ့် Port မတူပါဘူး။ ဒါကြောင့် ဒီလိုမတူတဲ့စနစ်ချင်း ဆက်သွယ်မှုကို ခွင့်ပြုပေးဖို့လိုပါတယ်။ ကိုယ့်ဘာသာ ရေးစရာမလိုပါဘူး။ `cors` Package ကို ခေါ်ထည့်လိုက်ရင် ရပါတယ်။ Express မှာ `app.use()` နဲ့ ဖြည့်စွက် လုပ်ဆောင်ချက်တွေ ထည့်ပေးလို့ ရပါတယ်။ `app.use(cors())` ဆိုတာ လက်ရှိ app Object မှာ `cors` Package ကပေးတဲ့ လုပ်ဆောင်ချက်တွေ ဖြည့်စွက်လိုက်တာပါပဲ။

`/info` Route သတ်မှတ်ပေးထားပါတယ်။ ဘာလုပ်ဆောင်ချက်မှ ဆန်းဆန်းပြားပြား မပါသေးပါဘူး။ စာတစ်ကြောင်းကို JSON Response အနေနဲ့ ပြန်ပေးထားပါတယ်။ အခုလို Run ကြည့်နိုင်ပါတယ်။

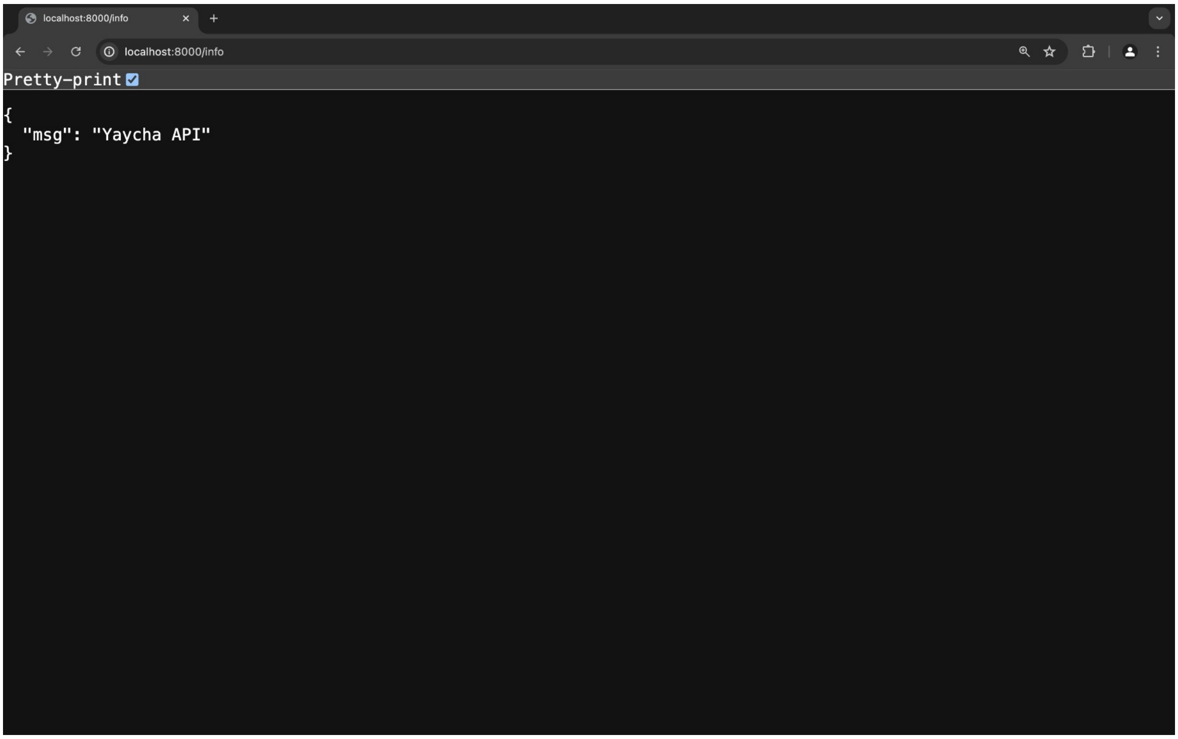
```
npx nodemon index.js

[nodemon] 3.1.4
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node index.js`
Yaycha API started at 8000...
```

Node နဲ့ မ Run ဘဲ စောစောက Install လုပ်လိုက်တဲ့ `nodemon` နဲ့ Run ထားပါတယ်။ ရိုးရိုး Node နဲ့ Run ဆိုရင် Run ပြီးနောက် ပြင်လိုက်တဲ့ကုဒ်တွေ အသက်ဝင်မှာ မဟုတ်ပါဘူး။ ဒါကြောင့် တစ်ခုခုပြင်ပြီး

တိုင်း API Server ကို ရပ်ပြီး ပြန် Run ပေးဖို့လိုပါတယ်။ `nodemon` ကတော့ ကုဒ်ဖိုင်ကို ပြင်ပြီးသိမ်းလိုက်ရင် API Server ကို အလိုအလျှောက် Restart လုပ်ပေးပါတယ်။ ဒါကြောင့် တစ်ခုခုပြင်ပြီးတိုင်း ကိုယ့်ဘာသာ ပိတ်လိုက်ဖွင့်လိုက် လုပ်ပေးစရာမလိုတော့ပါဘူး။ နောက်ပိုင်း Node Version တွေမှာ အလားတူ လုပ်ဆောင်ချက် ပါလာပေမယ့် စမ်းသပ်အဆင့်ပဲရှိသေးလို့ `nodemon` ကိုပဲ အသုံးပြုလိုက်တာပါ။ Server Run ပြီးရင် Browser မှာ ဒီလိုရိုက်ထည့်လိုက်ပါ –

http://localhost:8000/info

အခုလိုရလဒ်ကို တွေ့မြင်ရမှာဖြစ်ပါတယ်။



ဆက်လက်ပြီး Prisma နဲ့ တွဲသုံးမှာဖြစ်လို့ လိုအပ်မယ့် လုပ်ဆောင်ချက်တွေ ထပ်ထည့်ကြပါမယ်။ ပထမဦးဆုံး `prismaClient.js` အမည်နဲ့ ဖိုင်တစ်ခုဖန်တီးပြီး ဒီကုဒ်ကိုရေးပေးပါ။

JavaScript - API - prismaClient.js

```
const { PrismaClient } = require('@prisma/client');
const prisma = new PrismaClient();

module.exports = prisma;
```

ဘာလုပ်ဆောင်ချက်မှ ထူးထူးခြားခြားမပါပါဘူး။ Prisma Client ကို လိုတဲ့နေရာကနေ ယူသုံးလို့ရအောင် Export ထုတ်ပေးထားတာပါ။ `index.js` ကို အခုလို ပြင်ပေးလိုက်ပါ။

JavaScript - API - index.js

```
const express = require("express");
const app = express();

const prisma = require("./prismaClient");

const cors = require("cors");
app.use(cors());

app.get("/info", (req, res) => {
  res.json({ msg: "Yaycha API" });
});

const server = app.listen(8000, () => {
  console.log("Yaycha API started at 8000...");
});

const gracefulShutdown = async () => {
  await prisma.$disconnect();
  server.close(() => {
    console.log("Yaycha API closed.");
    process.exit(0);
  });
};
```

```
process.on("SIGTERM", gracefulShutdown);
process.on("SIGINT", gracefulShutdown);
```

SIGTERM တို့ SIGINT တို့လို့ Event တွေဟာ Ctrl + C နှိပ်ပြီး Server ကို ရပ်လိုက်ခြင်းအပါအဝင် API Server ကို Terminate လုပ်လိုက်ချိန်နဲ့ Interrupt လုပ်လိုက်ချိန်တွေမှာ အလုပ်လုပ်ပါတယ်။ Service ကို ရပ်လိုက်ချိန်မှာ Prisma Client ကို အလိုအလျှောက် Disconnect လုပ်ပေးဖို့ ထည့်ရေးပေးလိုက်တာပါ။ ဒါကြောင့် ကျန် Prisma Operation တွေမှာ တစ်ခုချင်း Manual Disconnect လိုက်လုပ်ပေးနေဖို့ မလိုအပ်တော့ပါဘူး။

Routers

နောက်ပိုင်းမှာ လိုအပ်တဲ့ ဖြည့်စွက်လုပ်ဆောင်ချက်တွေ ဆက်လုပ်ပါမယ်။ ဒီနေရာမှာတော့ အခြေခံ CRUD လုပ်ဆောင်ချက်တွေနဲ့ စကြပါမယ်။ ကုဒ်တွေကို တစ်နေရာထဲမှာ အကုန်စုမရေးဘဲ Router File တွေနဲ့ ခွဲပြီးရေးပါမယ်။ ဒါကြောင့် `routers` ဆိုတဲ့အမည်နဲ့ ဖိုဒါတစ်ခုဆောက်ပြီး အထဲမှာ `content.js` နဲ့ `user.js` ဖိုင် (၂) ခုဆောက်လိုက်ပါ။ **yaycha-api** ဖိုဒါရဲ့ ဖွဲ့စည်းပုံ အခုလို ဖြစ်သွားရပါမယ်။

```
yaycha-api
├── index.js
├── node_modules/
├── package-lock.json
├── package.json
├── prisma/
│   ├── dev.db
│   ├── migrations/
│   ├── schema.prisma
│   └── seeds/
│       ├── CommentSeeder.js
│       ├── PostSeeder.js
│       ├── UserSeeder.js
│       └── main.js
├── prismaClient.js
│
```

```
└── routers/
    ├── content.js
    └── user.js
```

ပြီးတဲ့အခါ `content.js` ထဲမှာ အခုလို ရေးပေးလိုက်ပါ။

JavaScript - API - routers/content.js

```javascript
const express = require("express");
const router = express.Router();

const prisma = require("../prismaClient");

router.get("/posts", async (req, res) => {
  try {
    const data = await prisma.post.findMany({
      include: {
        user: true,
        comments: true,
      },
      orderBy: { id: "desc" },
      take: 20,
    });

    res.json(data);
  } catch (e) {
    res.status(500).json({ error: e });
  }
});

module.exports = { contentRouter: router };
```

ဒီတစ်ခါ Express နဲ့ app Object ကို Create မလုပ်တော့ဘဲ `router` Object ကို Create လုပ်ထားပါတယ်။ အဲ့ဒီ Router ကိုသုံးပြီး `/posts` Route တစ်ခုရေးထားပါတယ်။ Express က Router Function

အနေနဲ့ `async` Function ကို လက်ခံအလုပ်လုပ်ပေးနိုင်ပါတယ်။ ဒါကြောင့် `async` Function ဖြစ်တယ်ဆိုတာကို သတိပြုပါ။

Prisma Client ကို အသစ်မလုပ်ဘဲ ကြိုလုပ်ထားပြီးသား Client ကို Import လုပ်ယူပြီးသုံးထားပါတယ်။ Error တွေ Handle လုပ်ဖို့အတွက် `try - catch` Statement နဲ့ရေးပါတယ်။ Prisma နဲ့ Post တွေကို ထုတ်ယူတဲ့အခါ `include` Option ကိုသုံးပြီး သက်ဆိုင်ရာ User တွေ Comment တွေကိုပါ တစ်ခါထဲ ထုတ်ယူထားပါတယ်။ ဖြည့်စွက်ချက်အနေနဲ့ `orderBy` Option ကိုသုံးပြီး Post တွေကို `id` နဲ့ `desc` ပြောင်းပြန်စီထုတ်ယူပါတယ်။ နောက်ပိုင်းမှာ Post တွေများလာရင် မလိုအပ်ဘဲ Data တွေအရမ်းများ မနေအောင် `take: 20` နဲ့ Record အခု (၂၀) ပဲ ကန့်သတ်ထုတ်ယူထားပါတယ်။

Success ဖြစ်မယ်ဆိုရင် `res.json()` နဲ့ ရလာတဲ့ Data တွေကို ပြန်ပေးပြီး၊ Error ရှိခဲ့မယ်ဆိုရင် 500 Status Code နဲ့အတူ Error တွေကို ပြန်ပေးထားပါတယ်။ 500 Internal Server Error ကို အခုလို Database အပါအဝင် Server-Side Error တွေပြန်ပို့တဲ့အခါ ပူးတွဲအသုံးပြုသင့်ပါတယ်။

နောက်ဆုံးမှာ `router` ကို `contentRouter` အမည်နဲ့ Export ထုတ်ပေးထားပါတယ်။ ဒါကြောင့် `index.js` မှာ သူ့ကို အခုလို ချိတ်ဆက်ထည့်သွင်းပေးပါ။

JavaScript - API - index.js

```
const express = require("express");
const app = express();

const prisma = require("./prismaClient");

const cors = require("cors");
app.use(cors());

const { contentRouter } = require("./routers/content");
app.use("/content", contentRouter);

...
```

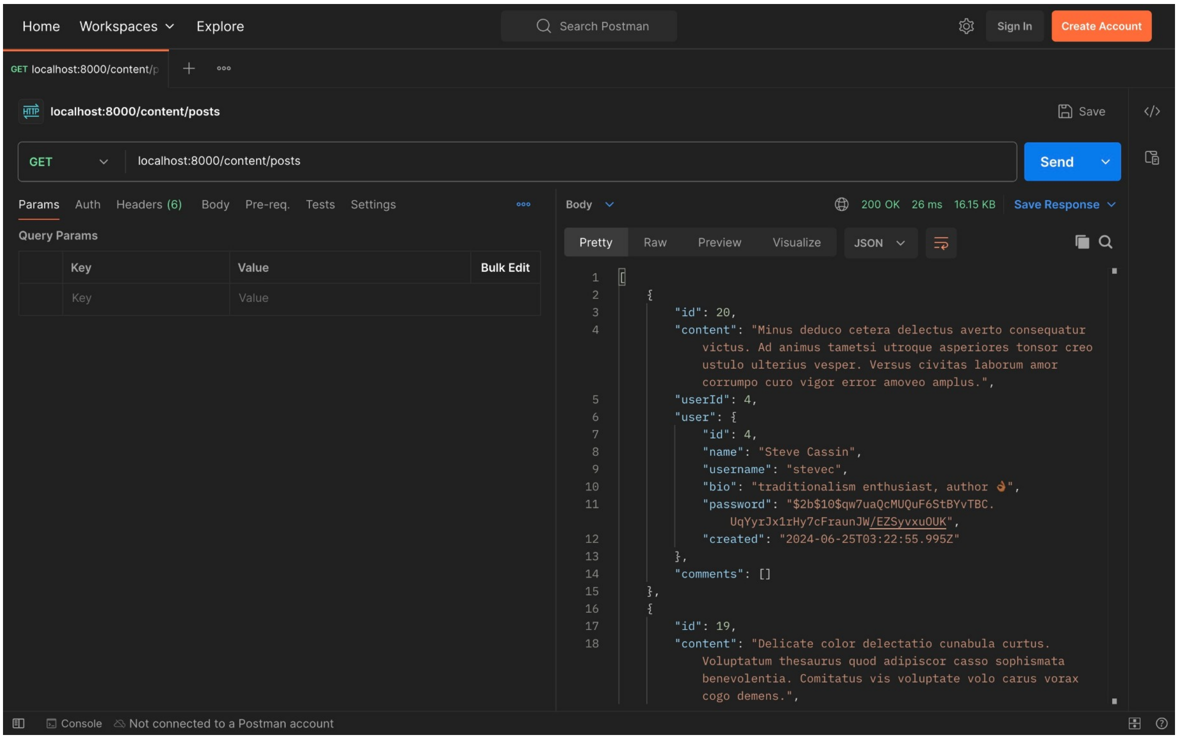
```
const server = app.listen(8000, () => {
  console.log("Yaycha API started at 8000...");
});

...
```

`app.use()` နဲ့ `contentRouter` ကို app မှာ ထည့်သုံးပေးလိုက်တာပါ။ အဲ့ဒီလိုသုံးတဲ့အခါ ရှေ့က `/content` URL လေးထည့်ပေးထားတာကို သတိပြုပါ။ ဒါကြောင့် `contentRouter` ရဲ့ လုပ်ဆောင်ချက်တွေကို လိုအပ်တဲ့အခါ ရှေ့က `/content` နဲ့ စပေးရမှာ ဖြစ်ပါတယ်။

API Server ကို မ Run ရသေးရင် Run လိုက်ပါ။ ပြီးတဲ့အခါ စမ်းကြည့်လို့ရပါပြီ။ စမ်းကြည့်တဲ့အခါ ရိုးရိုး Browser နဲ့စမ်းကြည့်တာထက် API Testing Software တစ်ခုခုနဲ့ စမ်းကြည့်တာက ပိုထိရောက်ပါလိမ့်မယ်။ ဒီစာအုပ်မှာ Postman လို့ခေါ်တဲ့ API Testing Software ကို အသုံးပြုဖော်ပြပါမယ်။ အောက်ပါလိပ်စာမှာ Download ရယူနိုင်ပါတယ်။

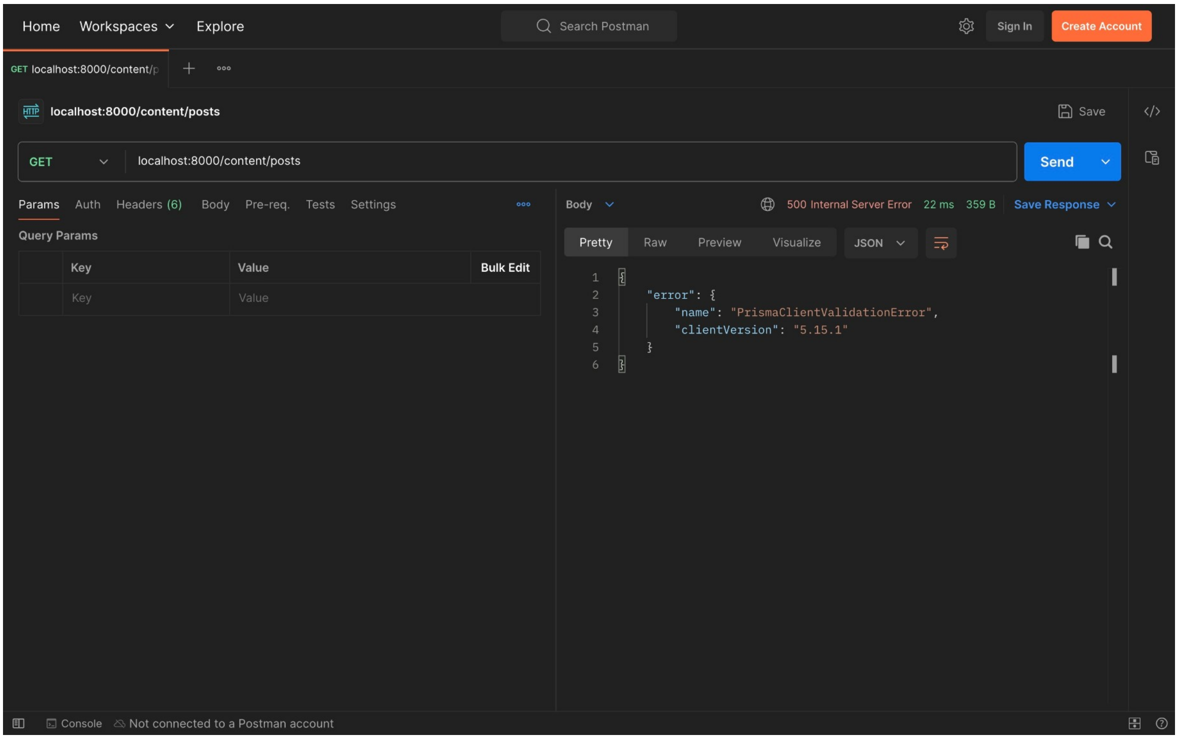
Postman – https://www.postman.com

Postman ရဲ့ Method မှာ `GET` ကိုရွေးပါ။ URL နေရာမှာ `localhost:8000/content/posts` ကို ထည့်သွင်းပြီး Send လုပ်လိုက်ရင် အခုလိုရလဒ်ကို ရရှိမှာ ဖြစ်ပါတယ်။



Post အခု (၂၀) ကို ပြောင်းပြန်စီပြီးရရှိခြင်းနဲ့အတူ သက်ဆိုင်ရာ User တွေ Comment တွေလည်း ပါလာတယ်ဆိုတာကို တွေ့ရမှာပဲ ဖြစ်ပါတယ်။ User Data တွေပါလာတဲ့အခါ Password လို Sensitive ဖြစ်တဲ့ အရာတွေလည်း ပါနေပါလိမ့်မယ်။ ပါမလာအောင် ချန်ခဲ့လို့ရတဲ့နည်းတွေ ရှိပေမယ့် ဒီနေရာမှာ ထည့်မကြည့်သေးပါဘူး။

အကယ်၍ ကုဒ်မှာ အမှားတစ်ခုခုရှိခဲ့ရင်တော့ အခုလိုရလဒ်ကို ရနိုင်ပါတယ်။



content.js မှာ အခုလို ရေးဖြည့်ပေးလိုက်ပါ။

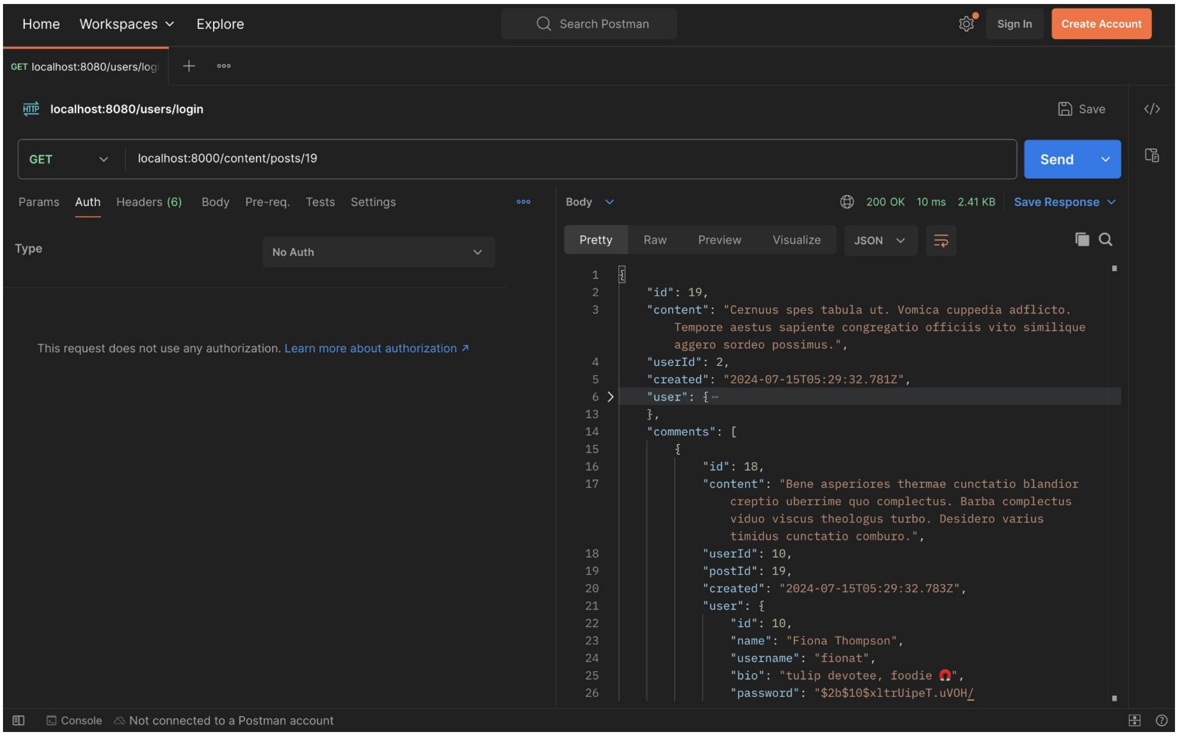
JavaScript - API - routers/content.js

```
router.get("/posts/:id", async (req, res) => {
  const { id } = req.params;

  try {
    const data = await prisma.post.findFirst({
      where: { id: Number(id) },
      include: {
        user: true,
        comments: {
          include: { user: true },
        },
      },
    });
```

```
    res.json(data);
  } catch (e) {
    res.status(500).json({ error: e });
  }
});
```

ဒီတစ်ခါ Dynamic Route ဖြစ်သွားပါပြီ။ `findFirst()` နဲ့ Post တစ်ခုထုတ်ယူပါတယ်။ လက်ခံရရှိတဲ့ `id` Route Parameter က String ဖြစ်မှာပါ။ String အတိုင်းပေးလို့မရတဲ့အတွက် `Number(id)` နဲ့ Number ပြောင်းပြီးပေးထားတာကို သတိပြုရပါမယ်။ Prisma က Type Sensitive ဖြစ်ပါတယ်။

`include` နဲ့ ရေးထားတဲ့ `post → comment` Relationship ကို သတိပြုကြည့်ပါ။ ထပ်ဆင့် `include` နဲ့ `comment → user` Relationship Data တွေကိုပါ တစ်ခါထဲ ယူထားတာကို တွေ့ရပါလိမ့်မယ်။



နမူနာမှာ URL ကို `localhost:8000/content/posts/19` နဲ့စမ်းပြထားပါတယ်။ Post 19 နဲ့ အတူ ထုံးစံအတိုင်း သက်ဆိုင်ရာ User တွေ Comment နဲ့ Comment User တွေကို တစ်ခါထဲ ပြန်ရတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။ ပို့စ်တွေ အများကြီးလိုချင်ရင်လည်း ရပါပြီ။ တစ်ခုထဲကို `id` နဲ့ အတိအကျလိုချင်ရင် လည်း ရသွားပါပြီ။

အလားတူလုပ်ဆောင်ချက်မျိုး User အတွက်လည်း ရသွားအောင် `user.js` မှာ အခုလိုရေးပေးပါ။

JavaScript - API - routers/user.js

```
const express = require("express");
const router = express.Router();

const prisma = require("../prismaClient");

router.get("/users", async (req, res) => {
  const data = await prisma.user.findMany({
    include: {
      posts: true,
      comments: true,
    },
    orderBy: { id: "desc" },
    take: 20,
  });

  res.json(data);
});

router.get("/users/:id", async (req, res) => {
  const { id } = req.params;

  const data = await prisma.user.findFirst({
    where: { id: Number(id) },
    include: {
      posts: true,
```

```
      comments: true,
    },
  });

  res.json(data);
});

module.exports = { userRouter: router };
```

`content.js` မှာ ရေးခဲ့တဲ့ကုဒ်နဲ့ ခပ်ဆင်ဆင်ပါပဲ။ Error Handle လုပ်တဲ့ `try - catch` တွေတော့ အပြည့်အစုံမရေးတော့ပါဘူး။ ဖော်ပြတဲ့ကုဒ်တွေ တိုပြီးဖတ်ရလွယ်အောင် ခဏချန်လို့ရတဲ့ကုဒ်တွေ ချန်လိုက်တဲ့သဘော ဖြစ်ပါတယ်။ ကိုယ့်ဘာသာ နောက်မှ ပြည့်စုံအောင် ဖြည့်ရေးလိုက်ပါ။

နောက်တစ်ချက် သတိပြုသင့်တာကတော့ User နဲ့ဆိုင်တဲ့ Post တွေ Comment တွေကိုပါ တစ်ခါထဲ ထုတ်ယူထားတဲ့အတွက် Post တွေ Comment တွေ အရမ်းများနေရင် ဘယ်လိုလုပ်မလဲဆိုတဲ့ အချက်ပါ။ အဲ့ဒါကိုလည်း ကန့်သတ်လို့ရပေမယ့် ဒီနေရာမှာတော့ ထည့်ကန့်သတ်မထားပါဘူး။

စမ်းကြည့်ဖို့အတွက် `userRouter` ကို `index.js` မှာ ချိတ်ပေးဖို့လိုပါသေးတယ်။

JavaScript - API - index.js

```
const express = require("express");
...

app.use(express.json());
app.use(express.urlencoded({ extended: true }));

const { userRouter } = require("./routers/user");
app.use("/", userRouter);

const { contentRouter } = require("./routers/content");
app.use("/content", contentRouter);
```

```
...

const server = app.listen(8000, () => {
  console.log("Yaycha API started at 8000...");
});

...
```

နှစ်ချက်သတိပြုပါ။ `useRouter` ကို ချိတ်တဲ့အခါ ရှေ့က Route URL ထပ်မပေးတော့ပါဘူး။ `"/"` အလွတ်ထားခဲ့ပါတယ်။ ဒါကြောင့် စမ်းကြည့်တဲ့အခါ `localhost:8000/users` နဲ့ စမ်းကြည့်ရမှာပါ။

လက်စနဲ့ `express.json()` နဲ့ `express.urlencoded()` တို့ကိုလည်း ချိတ်ပေးထားပါတယ်။ ရှေ့ဆက်ရေးတဲ့ အခါ ဒီလုပ်ဆောင်ချက်တွေ လိုအပ်မှာမို့လို့ပါ။ Client က Request Data တွေကို ပို့တဲ့အခါ Form Data, URL Encoded, JSON စသည်ဖြင့် Format အမျိုးမျိုးနဲ့ ပို့လာနိုင်ပါတယ်။ `express.json()` က Request နဲ့ပါလာတဲ့ JSON Data တွေကို လက်ခံအလုပ်လုပ်ပေးပါတယ်။ `express.urlencoded()` ကတော့ Request နဲ့ပါလာတဲ့ `x-www-form-urlencoded` Data တွေကို လက်ခံအလုပ်လုပ်ပေးပါတယ်။ ဒါကြောင့် ဒီ API က JSON နဲ့ `x-www-form-urlencoded` ဆိုတဲ့ Request Data နှစ်မျိုးကို လက်ခံအလုပ်လုပ်နိုင်သွားပြီ ဖြစ်ပါတယ်။

`urlencoded()` အတွက်ပေးလိုက်တဲ့ `{ extended: true }` Option က ဘာအတွက်လဲဆိုတာတော့ ထည့်မပြောတော့ပါဘူး။ နောင်လိုအပ်မှပဲ ဆက်လေ့လာကြည့်လိုက်ပါ။ လောလောဆယ် ဒီအတိုင်း ရေးရတယ်လို့ပဲ ပုံသေမှတ်ထားပေးပါ။

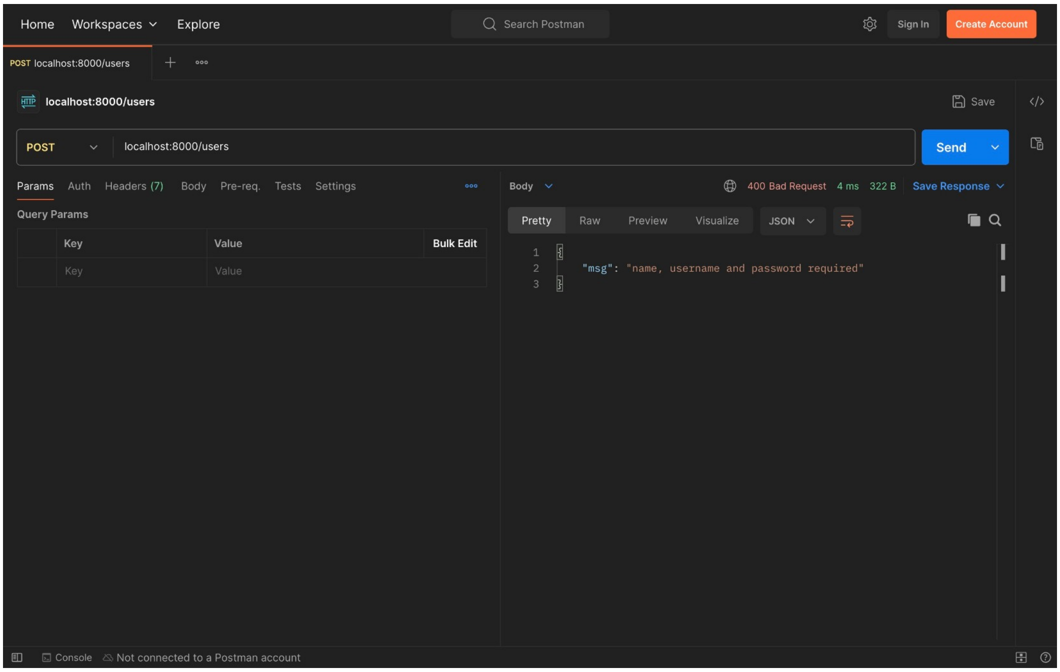
User အသစ်တွေထပ်ထည့်တဲ့ လုပ်ဆောင်ချက်အတွက် `user.js` မှာ အခုလို ရေးဖြည့်ပေးပါ။

JavaScript - API - routers/user.js

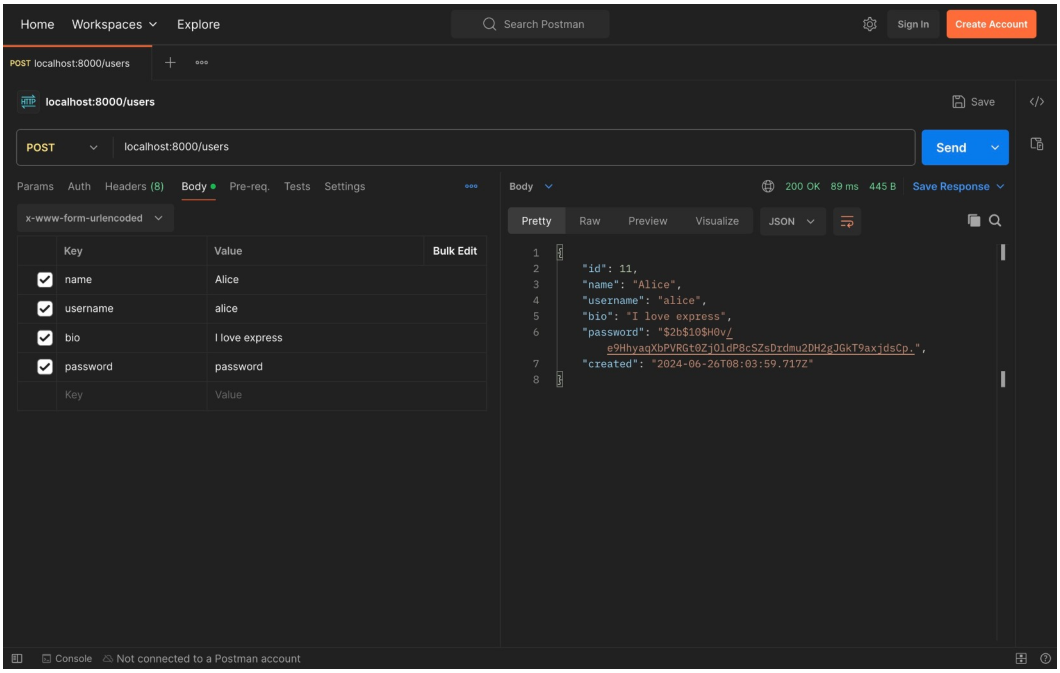
```
router.post("/users", async (req, res) => {
  const { name, username, bio, password } = req.body;

  if (!name || !username || !password) {
    return res
      .status(400)
      .json({ msg: "name, username and password required" });
  }

  const hash = await bcrypt.hash(password, 10);
  const user = await prisma.user.create({
    data: { name, username, password: hash, bio, },
  });

  res.json(user);
});
```

ဒီတစ်ခါ အသစ်ထည့်တဲ့အလုပ်မို့လို့ `router.post()` ကိုသုံးတာ ဖြစ်သွားပါပြီ။ စောစောက `index.js` မှာ ကြိုရေးခဲ့တဲ့ကုဒ်အရ Request နဲ့အတူပါလာတဲ့ JSON Data တွေ URL Encoded Data တွေမှာ `req.body` မှာ ရှိနေမှာ ဖြစ်ပါတယ်။ ဒါကြောင့် `name`, `username`, `bio` နဲ့ `password` တို့ကို `req.body` ကနေရယူပါတယ်။

Validation စစ်တဲ့သဘောမျိုးနဲ့ `name`, `username` နဲ့ `password` မပါရင် 400 Bad Request Status Code နဲ့ ဆက်အလုပ်မလုပ်အောင် return ပြန်ပေးထားပါတယ်။ ပိုပြည့်စုံတဲ့ Validation လုပ်ဆောင်ချက်တွေ လိုအပ်ရင်တော့ Express Validator လို Package မျိုးကို အသုံးပြုလို့ရပါတယ်။ ဒီနေရာမှာတော့ ရိုးရိုးပဲ `if` နဲ့ စစ်ထားပါတယ်။ `password` ကို `bcryp` နဲ့ Hash လုပ်ထားတာကိုလည်း သတိပြုပါ။ ပြီးတဲ့အခါ `prisma.create()` နဲ့ Data တွေကိုသိမ်းလိုက်ပြီး ပြန်ရလာတဲ့ရလဒ်ကို Response အဖြစ်ပြန်ပေးထားပါတယ်။ စမ်းကြည့်လို့ရပါပြီ။ Method ကို `POST` ရွေးပြီး `localhost:8000/users` ကို သွားလိုက်တဲ့အခါ 400 Bad Request ပြန်လာတာကို တွေ့ရမှာဖြစ်ပါတယ်။



လိုအပ်တဲ့ Data တွေမပါလို့ပါ။ ဒါကြောင့် Body → `x-www-form-urlencoded` ကိုရွေးပြီး သင့်တော်တဲ့အချက်အလက်တွေ ထည့်ပေးလိုက်ရင်တော့ အခုလို ရလဒ်အမှန်ကို ရရှိမှာပါ။

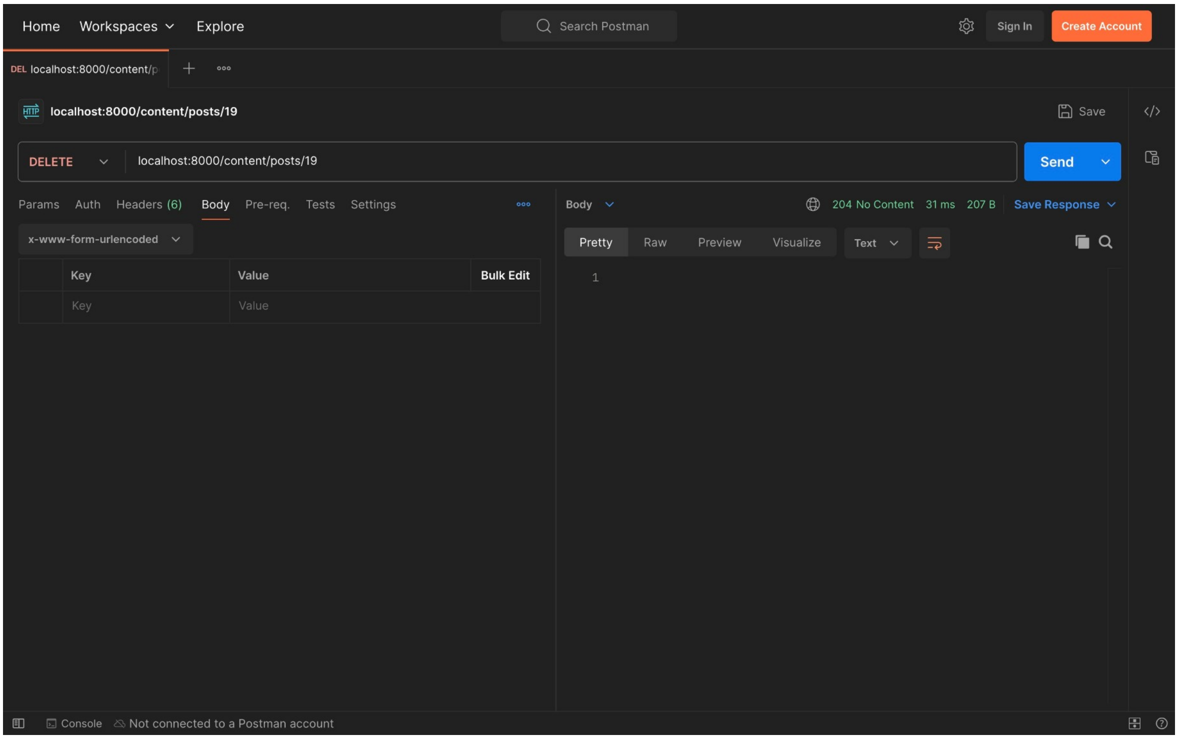


ဒီနည်းနဲ့ User တွေ အသစ်ထည့်တဲ့လုပ်ဆောင်ချက်ရ သွားပါပြီ။ ပြန်ဖျက်တဲ့ လုပ်ဆောင်ချက်စမ်းကြည့်ဖို့ အတွက် `content.js` မှာ အခုလိုရေးကြည့်ပါ။

JavaScript - API - routers/content.js

```
router.delete("/posts/:id", async (req, res) => {
  const { id } = req.params;

  await prisma.comment.deleteMany({
    where: { postId: Number(id) },
  });

  await prisma.post.delete({
    where: { id: Number(id) },
  });

  res.sendStatus(204);
});

router.delete("/comments/:id", async (req, res) => {
  const { id } = req.params;

  await prisma.comment.delete({
    where: { id: Number(id) },
  });

  res.sendStatus(204);
});
```

ပြန်ဖျက်ချင်တာဖြစ်လို့ `router.delete()` ကိုသုံးထားပါတယ်။ Post Delete အတွက် သူနဲ့ဆိုင်တဲ့ Comment တွေကို အရင်ဖျက်ပြီးမှ Post ကိုဖျက်ထားပါတယ်။ ပြီးတဲ့အခါ Success ဖြစ်ကြောင်းလည်း ပြောချင်ပေမယ့် ဘာ Data မှလည်းပြန်မပေးချင်တဲ့အတွက် `res.sendStatus()` နဲ့ 204 No Content ကို ပြန်ပေးထားပါတယ်။



ဒီအခန်းမှာ ဒီလောက်ပဲ လုပ်ကြပါမယ်။ ကျန်လိုအပ်တဲ့ လုပ်ဆောင်ချက်တွေကို နောက်အခန်းတွေမှာ လိုအပ်သလို ဆက်ဖြည့်သွားကြမှာပါ။

အခန်း (၁၂) - React - React Query

အခုဆိုရင် React, MUI နဲ့ React Router ပေါင်းစပ်ပြီး Front-end App လေးတစ်ခုလည်း လုပ်လက်စ ရှိနေပြီ။ Express နဲ့ Prisma ကို ပေါင်းစပ်ပြီး Back-end API လေးလည်း ရနေပြီ။ ဒီနှစ်ခုကို ပေါင်းစပ်ကြပါမယ်။ React မှာ API နဲ့ချိတ်ဆက်ဖို့အတွက် Data Fetching Pattern တွေ အမျိုးမျိုးရှိပါတယ်။

ပထမဆုံးနည်းလမ်းကတော့ သီးခြားနည်းပညာတွေမပါဘဲ `fetch()` ကိုပဲ React ရဲ့ `useEffect()`, `useState()` Hook တွေနဲ့ ပူးတွဲအသုံးပြုခြင်းဖြစ်ပါတယ်။ ဒီနည်းဟာ အစစအရာအရာ ကိုယ့်ဘာသာ Manual လုပ်ရတဲ့သဘောဖြစ်လို့ လုပ်ဆောင်ချက်တွေများတဲ့ ပရောဂျက်ကြီးတွေမှာ အခက်အခဲရှိနိုင်ပါတယ်။ ဒါပေမယ့် အခြေခံအကျဆုံးနည်းလမ်း ဖြစ်တဲ့အတွက် သိထားသင့်ပါတယ်။ ခဏနေတော့ လက်တွေ့ ကုဒ်နမူနာတချို့ ရေးကြည့်ကြပါမယ်။

နောက်တစ်နည်းကတော့ React Query တို့ SWR တို့လို Data Fetching Library တွေ အသုံးပြုခြင်း ဖြစ်ပါတယ်။ ဒီနည်းမှာ အခြေခံလိုအပ်ချက်တွေကို Library တွေက အလိုအလျောက်လုပ်ပေးလို့ နေရာတကာ ကိုယ့်ဘာသာ အကုန်လိုက်လုပ်စရာ မလိုတော့ပါဘူး။ React Query အကြောင်းကိုလည်း မကြာခင် ဆက်လက်လေ့လာကြပါမယ်။

နောက်တစ်နည်းကတော့ Router ရဲ့ Data Loading လုပ်ဆောင်ချက်ကို အသုံးပြုခြင်း ဖြစ်ပါတယ်။ လက်ရှိအသုံးပြုထားတဲ့ React Router မှာ အခုလိုရေးလို့ရပါတယ်။

React

```
const router = createBrowserRouter([
  {
    path: "/",
    loader: async () => {
      // fetch and return data
    },
    element: <Home />
  }
]);
```

`loader` Option နဲ့ လိုအပ်တဲ့ Data ကို ယူထားတာပါ။ ဒီလိုယူထားတဲ့ Data ကို Component မှာ အခုလို ပြန်လည်အသုံးပြုနိုင်ပါတယ်။

React

```
function Home() {
  const data = useLoaderData();
  // ...
}
```

ဒီနည်းနဲ့ သက်ဆိုင်ရာ Route Element ကို Render မလုပ်ခင် လိုအပ်တဲ့ Data ကို ကြိုယူပြီးဖြစ်လို့ Component မှာ Effect တွေ State တွေနဲ့ သပ်သပ်ထပ် ယူစရာမလိုတော့ပါဘူး။ React Query လို နည်းပညာမျိုးနဲ့ အခြေခံ သဘောသဘာဝ တူညီပေမယ့်၊ မတူကွဲပြားတဲ့ ရေးနည်းဖြစ်သွားပါတယ်။ ဒီနည်းကို နှစ်သက်သူတွေလည်း အများအပြား ရှိကြပါတယ်။

နောက်ဆုံးနည်းလမ်းကတော့ Server-side Data Fetching ဖြစ်ပါတယ်။ Client App က API ကို လှမ်း Fetch လုပ်တာမျိုး မဟုတ်တော့ဘဲ၊ လိုအပ်တဲ့ Data တွေကို Server-side မှာကတည်းက ယူလာတဲ့နည်းလို့ ဆိုရပါမယ်။ Next.js နဲ့ React Server Component တို့လို နည်းပညာတွေကို အသုံးပြုရပါတယ်။ ဒီအကြောင်းတွေကို နောက်ပိုင်းအခန်းတွေမှာ ဆက်လက်လေ့လာကြပါမယ်။

Data Fetching

ရှေ့ဆက်ပြီး အဓိကထား လေ့လာအသုံးပြုသွားချင်တာက React Query ပါ။ ဒါပေမယ့် ရိုးရိုး Manual ရေးနည်းကိုလည်း သိထားဖို့လိုပါတယ်။ အခြေခံကျကျ သိထားမှ ရှိနေတဲ့ အခက်အခဲတွေကို ပိုမြင်ပြီး ထိရောက်အောင်အသုံးချနိုင်မှာပါ။ ပထမဆုံးအနေနဲ့ **yaycha-api** Server ကို Run ထားဖို့ မမေ့ပါနဲ့။ ပြီးတဲ့အခါ လုပ်လက်စ **yaycha** App မှာ `.env` အမည်နဲ့ဖိုင်တစ်ခု တည်ဆောက်ပြီး အခုလိုရေးပေးပါ။

.env

```
VITE_API="http://localhost:8000"
```

ပရောဂျက်အတွင်း လိုတဲ့နေရာကနေ ခေါ်သုံးလို့ရတဲ့ Environment Variable တစ်ခု သတ်မှတ်လိုက်တာပါ။ Vite ကို အသုံးပြုစီမံနေတဲ့ ဒီပရောဂျက်မှာ Environment Variable တွေဟာ ရှေ့က `VITE_` နဲ့စပေးရပါတယ်။ ဒါက Vite ကသတ်မှတ်ထားတာပါ။ အဲ့ဒီလိုစပေးမှ ပြန်ယူသုံးလို့ရပါတယ်။ ဒီလို Environment Variable အနေနဲ့ ကြိုတင်သတ်မှတ်ထားလို့ လိုတဲ့နေရာက ယူသုံးလို့ရသွားတဲ့အပြင် နောင်လိုအပ်လို့ ပြင်ချင်ရင်လည်း တစ်နေရာထဲမှာပဲ ပြင်ဖို့လိုအပ်မှာ ဖြစ်ပါတယ်။

ဆက်လက်ပြီး `pages/Home.jsx` မှာ အခုလိုရေးပေးပါ။

React - src/pages/Home.jsx

```
import { useState, useEffect } from "react";

// other imports

export default function Home() {
  ...

  const [data, setData] = useState([]);

  useEffect(() => {
    const api = import.meta.env.VITE_API;
```

```
    fetch(`${api}/content/posts`).then(async res => {
      setData(await res.json());
    });
  }, []);

  ...
}
```

State ရဲ့ Default Data တွေ ဖယ်ထုတ်လိုက်ပါပြီ။ မလိုအပ်တော့ပါဘူး။ သူ့အစား `useEffect()` ထဲမှာ `fetch()` နဲ့ API ကို လှမ်းခေါ်ထားပါတယ်။ `fetch()` က Response Object ကို Promise အနေနဲ့ ပြန်ပေးပါတယ်။ ဒါကြောင့် `then` Statement နဲ့ Response ပြန်ရလာတဲ့အခါ ရလာတဲ့ Response ကို JSON ပြောင်းခိုင်းထားပါတယ်။ API က ပြန်ပေးကတည်းက JSON အနေနဲ့ ပြန်ပေးပြီးသားပါ။ ဒါပေမယ့် JSON String ဖြစ်ပါတယ်။ အဲ့ဒါကို JSON Object ပြောင်းပေးဖို့လိုတာပါ။

`res.json()` က Response JSON String ကို JSON Object ပြောင်းပေးမယ်ဆိုတဲ့ Promise ပဲ ပြန်ပေးပါတယ်။ ဒါကြောင့် `then` Statement နဲ့ ပြောင်းပြီးရင် ဘာဆက်လုပ်ရမလဲ ဆက်လက်သတ်မှတ်ပေးရမှာပါ။ နမူနာမှာတော့ `then` ကိုမသုံးတော့ဘဲ `async / await` နဲ့ ပြန်ရလာတဲ့ JSON ကို `setData()` နဲ့ State Data ထဲ ထည့်ပေးလိုက်ပါတယ်။ ဒီကုဒ်ကို ရေးမယ်ဆိုရင် အခုလိုရေးရမှာပါ။

JavaScript

```
// method 1
fetch(api)
  .then(res => res.json())
  .then(json => setData(json));
```

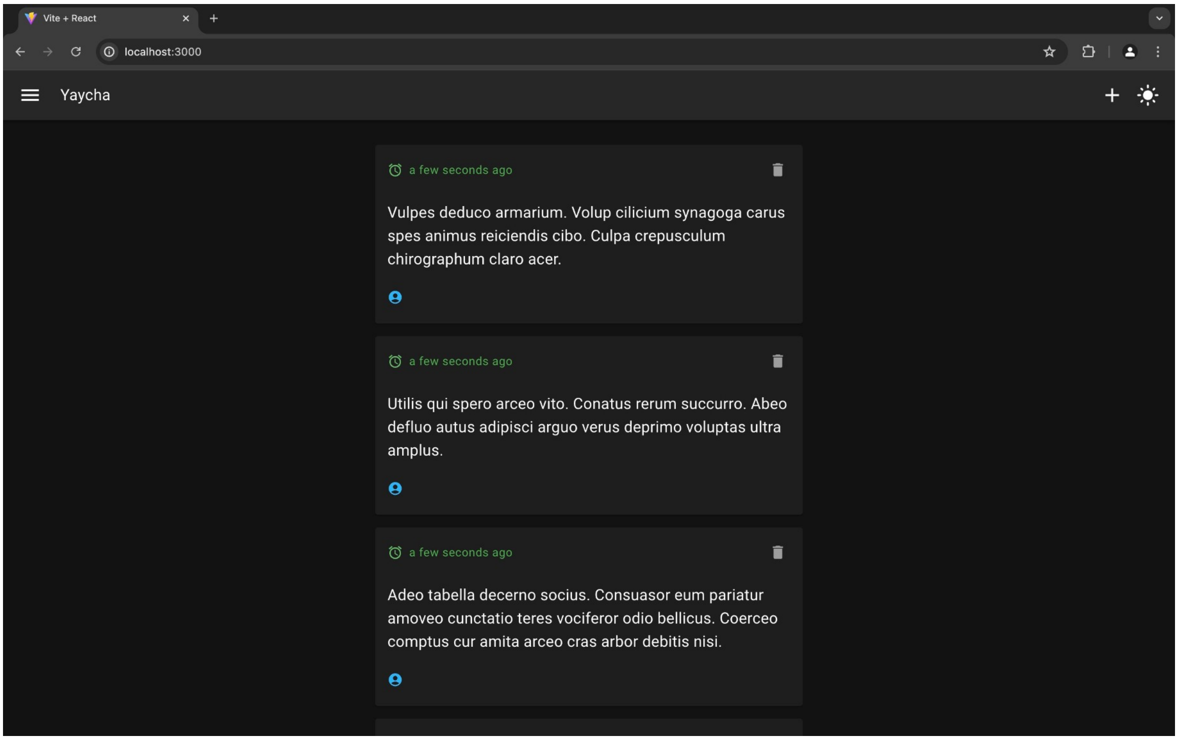
JavaScript

```
// method 2
async function fetchPosts() {
  const res = await fetch(api);
  const json = await res.json();
```

```
  setData(json);
}

fetchPosts();
```

ပထမရေးနည်းက ရိုးရိုး `then()` Statement တွေချည်းပဲ သုံးလိုက်တာဖြစ်လို့ ကုဒ် Flow က သိပ်ဖတ်လို့ မကောင်းပါဘူး။ ဒုတိယရေးနည်းက `async / await` နဲ့ ကုဒ် Flow က Step-by-Step ဖြစ်သွားလို့ ဖတ်လို့ကောင်းပေမယ့် async function တစ်ခု သပ်သပ်ရေးနေရပါတယ်။ ဒါကြောင့် လက်တွေ့ရေးဖို့ပေးထားတဲ့ နမူနာမှာ နှစ်မျိုးရောသုံးလိုက်တာပါ။ မိမိနှစ်သက်ရာရေးနည်းကို အသုံးပြုနိုင်ပါတယ်။

စမ်းကြည့်လိုက်ရင် အခုလိုရပါလိမ့်မယ်။



API ကပြန်ပေးတဲ့ Post အခု (၂၀) ကို ပြပေးနေပါပြီ။ စုံတော့မစုံသေးပါဘူး။ `Item` Component ကို နည်းနည်းပြင်ရပါဦးမယ်။ အရင်ဆုံး လိုအပ်တဲ့ Package တစ်ခုကို အခုလို Install လုပ်လိုက်ပါ။

```
npm i date-fns
```

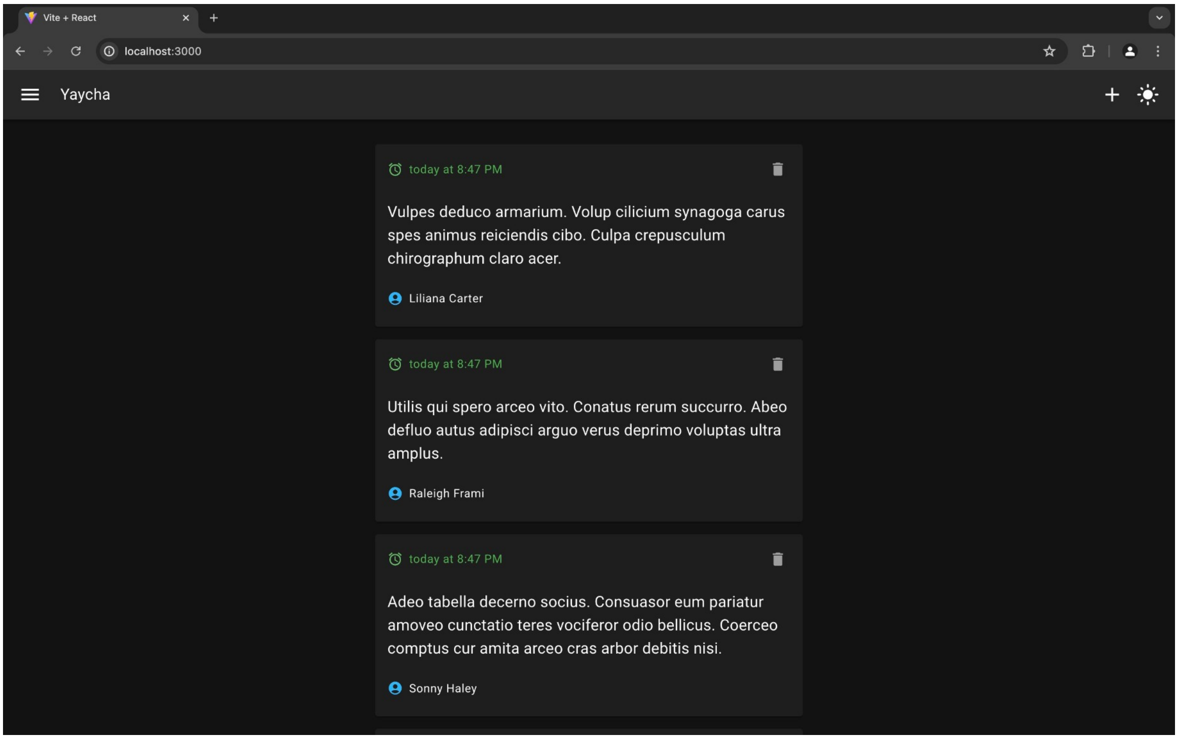
Data တွေ Time တွေကို Manage လုပ်ပေးနိုင်တဲ့ Package ပါ။ ပြီးတဲ့အခါ `components/Item.jsx` ကို အခုလို ပြင်ပေးပါ။

React - src/components/Item.jsx

```
// other imports

import { formatRelative } from "date-fns";

export default function Item({ ... }) {
  ...

  return (
    <Card sx={{ mb: 2 }}>
      ...

      <CardContent ...>
        <Box
          sx={{ ... }}>
          <Box
            sx={{ ... }}>
            <TimeIcon ... />
            <Typography ... }}>
              {formatRelative(item.created, new Date())}
            </Typography>
          </Box>
          ...
        </Box>

        <Typography sx={{ my: 3 }}>{item.content}</Typography>
```

```
        <Box
          sx={{ ... }}>
          <UserIcon ... />
          <Typography variant="caption">{item.user.name}</Typography>
        </Box>
      </CardContent>
    </Card>
  );
}
```

`formatRelative` ကို `date-fns` ကနေ Import လုပ်ယူပြီး `created` ရဲ့ မူလ DateTime ကို Relative Time ဖြစ်အောင် ပြောင်းပြီး ပြခိုင်းထားပါတယ်။ ပြီးတဲ့အခါ `item.name` အစား `item.user.name` နဲ့ မှန်အောင် ပြင်ပေးထားပါတယ်။



ဒီနေရာမှာ ဖြည့်စွက်စဉ်းစားစရာတွေ ရှိပါတယ်။ အခုလက်ရှိ ကိုယ့်စက်ထဲမှာပဲ စမ်းနေလို့ API Data ကို ချက်ချင်းလိုလိုရပါတယ်။ လက်တွေ့မှာ API Server ကနေ Client App ဆီ Data ရောက်ဖို့ အချိန်ကာလ

တစ်ခု စောင့်ရတာမျိုး ရှိနိုင်ပါတယ်။ ပြီးတော့ API Server က ဒေါင်းနေတာတို့၊ ဆက်သွယ်မှု ကြားထဲမှာ တစ်ခုခုအဆင်မပြေတာတို့လည်း ဖြစ်နိုင်ပါတယ်။ ဒါကိုနမူနာစမ်းကြည့်လို့ရအောင် **yaycha-api** ရဲ့ `routers/content.js` မှာ အခုလိုပြင်ကြည့်ပါ။

JavaScript - API - routers/content.jsx

```
router.get("/posts", async (req, res) => {
  try {
    const data = await prisma.post.findMany({
      include: {
        user: true,
        comments: true,
      },
      orderBy: { id: "desc" },
      take: 20,
    });

    setTimeout(() => {
      res.json(data);
    }, 2000);

  } catch (e) {
    res.status(500).json({ error: e });
  }
});
```

Response ကို ချက်ခြင်းပြန်မပို့ဘဲ `setTimeout` နဲ့ ၂ စက္ကန့်လောက်စောင့်ပြီးမှ ပြန်ပို့ထားတာပါ။ ဒါကြောင့် React App ဘက်မှာ ပြန်စမ်းကြည့်ရင် Post တွေက ချက်ခြင်းမပေါ်ဘဲ ခဏနေမှ ပေါ်လာတာမကို တွေ့ရမှာ ဖြစ်ပါတယ်။ ချက်ခြင်းမပေါ်တာထက် ပိုဆိုးတဲ့ ပြဿနာလည်း ရှိပါသေးတယ်။ တချို့နေရာတွေမှာ Data မရောက်သေးဘဲ Component ကို အသုံးပြုမိရင် Error တက်နိုင်ပါတယ်။ ဒါကြောင့် ဒီပြဿနာကို ထည့်ဖြေရှင်းဖို့လိုပါတယ်။ ဒါကြောင့် `Home.jsx` ကို အခုလိုပြင်လိုက်ပါ။

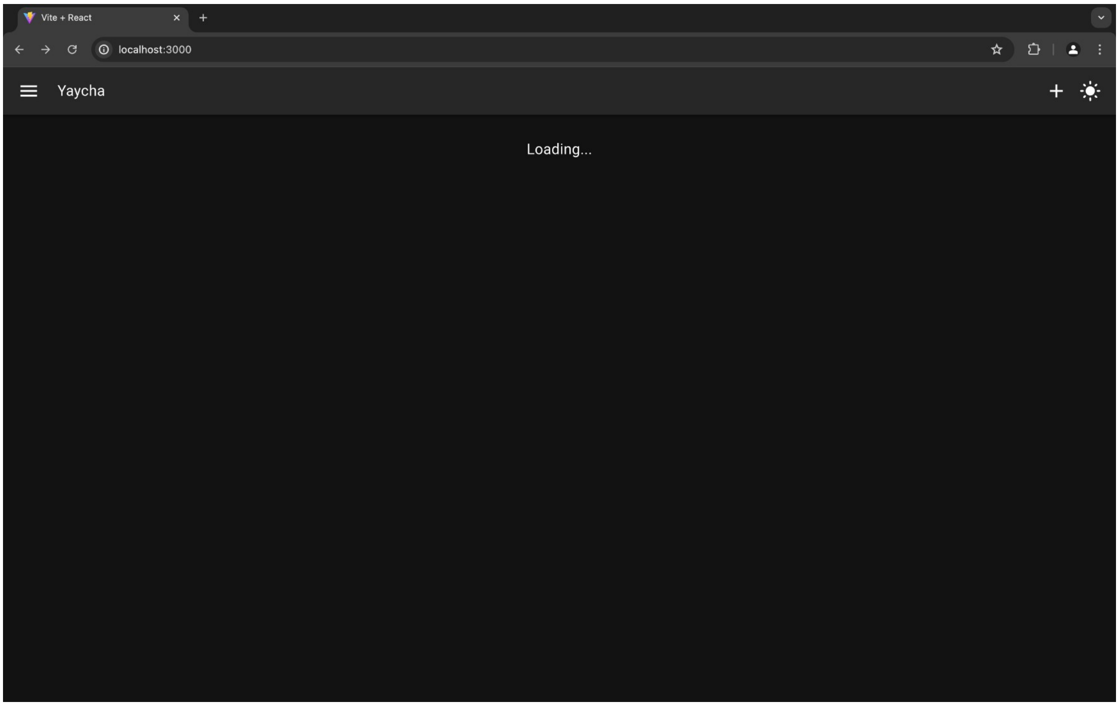
React - src/pages/Home.jsx

```
import { Alert, Box } from "@mui/material";

// other imports

export default function Home() {
  ...

  const [data, setData] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(false);

  useEffect(() => {
    const api = import.meta.env.VITE_API;
    fetch(`${api}/content/posts`)
      .then(async res => {
        if (res.ok) {
          setData(await res.json());
          setLoading(false);
        } else {
          setError(true);
        }
      })
      .catch(() => {
        setError(true);
      });
  }, []);

  ...

  if (error) {
    return (
      <Box>
        <Alert severity="warning">Cannot fetch data</Alert>
      </Box>
```

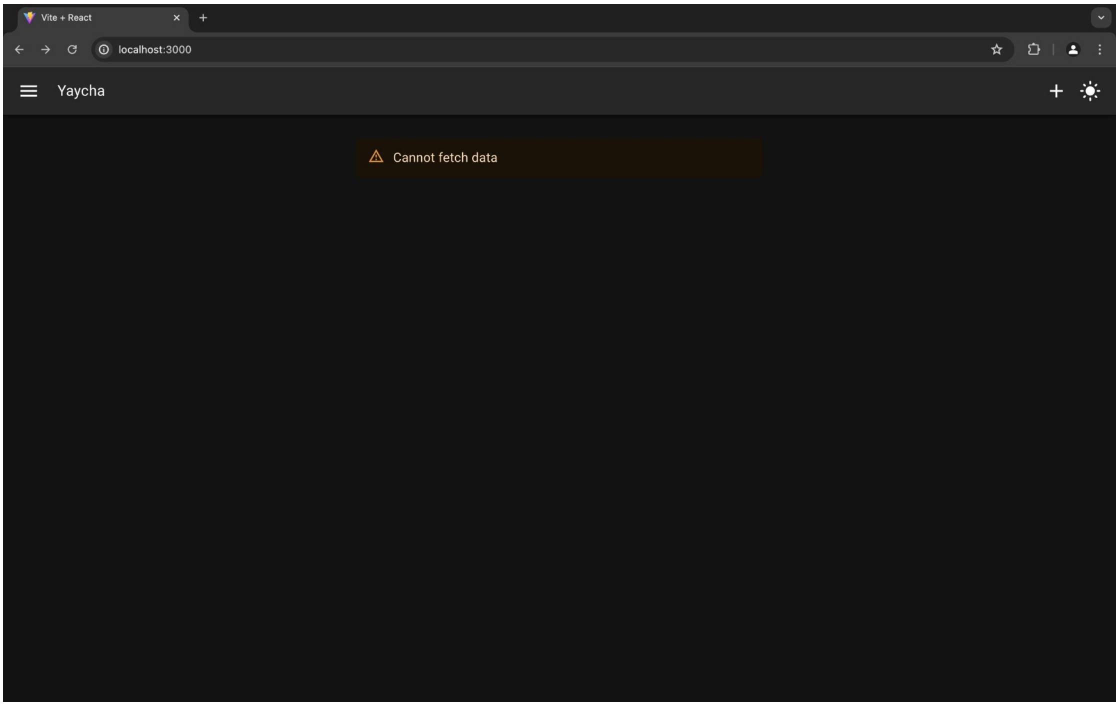
```
    );
  }

  if (loading) {
    return <Box sx={{ textAlign: "center" }}>Loading...</Box>;
  }

  return (
    <Box>
      ...
    </Box>
  );
}
```

`loading` နဲ့ `error` ဆိုတဲ့ State နှစ်ခု ကြေညာထားပါတယ်။ `loading` က `true` ဖြစ်ပြီး `error` က `false` ပါ။ `fetch()` နဲ့ API ကို ဆက်သွယ်တဲ့နေရာမှာ `catch()` Statement ပါသွားပါပြီ။ ဒါကြောင့် Network Error တစ်ခုခုရှိခဲ့ရင် `error` က `true` ဖြစ်သွားမှာပါ။ ပြီးတော့ `res.ok` ကို စစ်ပြီး ပြန်ရတဲ့ API Response မှာ တစ်ခုခုအဆင်မပြေခဲ့ရင်လည်း `error` က `true` ဖြစ်မှာပါ။ အားလုံး အဆင်ပြေလို့ `setData()` နဲ့ လိုအပ်တဲ့ Data ရပြီဆိုမှ `loading` ကို `false` လုပ်ထားပါတယ်။

ဆက်လက်ပြီး `error` ရှိနေရင် ကျန်တာတွေဆက်မလုပ်ဘဲ `<Alert>` နဲ့ Error Message တစ်ခုပြခိုင်းထားပါတယ်။ အလားတူပဲ `loading` လုပ်နေဆဲဆိုရင် Loading... ဆိုတာ စာလေးပြခိုင်းထားပါတယ်။ စမ်းကြည့်ရင် အခုလို Loading... လေးကိုအရင်ပြပြီး Data ရတော့မှ သက်ဆိုင်ရာ Data တွေကို ပြပေးမှာပါ။



API Server ကို ခဏဒေါင်းပြီး စမ်းကြည့်ရင်တော့ အခုလို Error Message ကို တွေ့မြင်ရမှာပါ။



ဒီကိစ္စတွေက အမြဲလိုလို ထည့်သွင်းဖြေရှင်းရမယ့် ကိစ္စတွေဖြစ်ပါတယ်။ ဒါကြောင့် Data Fetching လုပ်ဆောင်ချက်ကို အခုလိုကိုယ့်ဘာသာ လုပ်မယ့်အစား Data Fetching Library တစ်ခုရဲ့ အကူအညီကို ယူလိုက်ရင်တော့ ပိုအဆင်ပြေသွားမှာ ဖြစ်ပါတယ်။ ဒါကြောင့် ရေးလက်စကုဒ်ကို React Query နဲ့ ပြောင်းသုံးကြပါမယ်။ ပထမဦးဆုံး React Query ကို အခုလို Install လုပ်ပေးပါ။

```
npm i react-query
```

ပြီးတဲ့အခါ `ThemedApp.jsx` ကို အခုလို ပြင်လိုက်ပါ။

React - src/ThemedApp.jsx

```
...

import { QueryClientProvider, QueryClient } from "react-query";

...

export const queryClient = new QueryClient();

export default function ThemedApp() {
  ...

  return (
    <ThemeProvider theme={theme}>
      <AppContext.Provider
        value={{
          ...
        }}>

        <QueryClientProvider client={queryClient}>
          <RouterProvider router={router} />
        </QueryClientProvider>
```

```
          <CssBaseline />
        </AppContext.Provider>
      </ThemeProvider>
    );
}
```

`QueryClient` နဲ့ `QueryClientProvider` တို့ကို `react-query` ကနေ Import လုပ်ယူထားပါတယ်။ ဆက်လက်ပြီး `queryClient` Object တစ်ခုတည်ဆောက်ပြီး လိုအပ်ရင် ပြန်သုံးလို့ရအောင် Export ပေးထားပါတယ်။ ပြီးတဲ့အခါ App တစ်ခုလုံးကို `QueryClientProvider` ထဲမှာ ထည့်ပေးလိုက်ပါတယ်။ ဒီလိုထည့်ပေးမှ လိုအပ်တဲ့နေရာကနေ React Query ရဲ့ လုပ်ဆောင်ချက်တွေကို ယူသုံးလို့ ရမှာပါ။

`Home.jsx` မှာ ရေးထားတဲ့ Data Fetching လုပ်ဆောင်ချက်ကို React Query နဲ့ အခုလို ပြင်ရေးလိုက်လို့ရပါပြီ။ ကုဒ်အပြည့်အစုံ ပြန်ပြထားပါတယ်။

React - src/pages/Home.jsx

```
import { Alert, Box } from "@mui/material";

import Form from "../components/Form";
import Item from "../components/Item";

import { useApp } from "../ThemedApp";
import { useQuery } from "react-query";

const api = import.meta.env.VITE_API;

export default function Home() {
  const { showForm, setGlobalMsg } = useApp();
  const { isLoading, isError, error, data } = useQuery("posts",
```

```
    async () => {
      const res = await fetch(`${api}/content/posts`);
      return res.json();
    });

  const remove = id => {
    setData(data.filter(item => item.id !== id));
    setGlobalMsg("An item deleted");
  };

  const add = (content, name) => {
    const id = data[0].id + 1;
    setData([{ id, content, name }, ...data]);
    setGlobalMsg("An item added");
  };

  if (isError) {
    return (
      <Box>
        <Alert severity="warning">{error.message}</Alert>
      </Box>
    );
  }

  if (isLoading) {
    return <Box sx={{ textAlign: "center" }}>Loading...</Box>;
  }

  return (
    <Box>
      {showForm && <Form add={add} />}

      {data.map(item => {
        return (
```

```
          <Item
            key={item.id}
            item={item}
            remove={remove}
          />
        );
      })}
    </Box>
  );
}
```

useState တွေ useEffect တွေ မလိုအပ်တော့လို့ ထည့် Import မလုပ်ထားပါဘူး။ useQuery ကို တော့ react-query ကနေ Import လုပ်ယူထားပါတယ်။

Component Function ထဲမှာ useQuery ကပြန်ပေးတဲ့ Query Object ကနေ လိုချင်တဲ့ isLoading, isError, error နဲ့ data တို့ကို လက်ခံယူထားပါတယ်။ useQuery အတွက် ပထမ Parameter အနေနဲ့ Query Key ကို ပေးရပါတယ်။ နှစ်သက်ရာအမည်ကို ပေးလို့ရပါတယ်။ နောင်ဒီ Key နဲ့ Query Cache တွေကို Manage လုပ်ရမှာပါ။ ဒါကြောင့် အဓိပ္ပါယ်ရှိပြီး မှတ်ရလွယ်တာကို ပေးရပါမယ်။ ဒုတိယ Parameter အနေနဲ့ Data Fetching Function ကို ပေးရပါတယ်။ fetch() ကိုပဲ ပြန်သုံးထားပါတယ်။ Data Fetching Function က Promise ဖြစ်ရပါတယ်။ ဒါကြောင့် return res.json() နဲ့ Promise ကို ပြန်ပေးထားပါတယ်။ return await res.json() မဟုတ်ပါဘူး။ <u>အဲ့ဒီလိုရေးရင် Promise Resolve ဖြစ်ပြီးသား ရလဒ်ကို ပြန်ပေးသလို ဖြစ်သွားမှာပါ</u>။

Error Handle လုပ်တာတွေ Loading Handle လုပ်တာတွေ React Query က လုပ်သွားပါလိမ့်မယ်။ ကိုယ်ဘာသာ လိုက်စစ်နေစရာ မလိုတော့ပါဘူး။ တစ်ခုတော့ ရှိပါတယ်။ Error ဆိုတဲ့နေရာမှာ Network Error ကိုပဲ Handle လုပ်ပေးမှာပါ။ API Error ကိုတော့ Handle လုပ်ပေးမှာ မဟုတ်ပါဘူး။ အဲ့ဒါကြောင့် React Query ကြောင့်မဟုတ်ပါဘူး။ fetch() ကြောင့်ပါ။ <u>fetch() က 400, 500 စသည်ဖြင့် Error Code တွေရှိနေရင်တောင် Reject မလုပ်ဘဲ Resolve ပဲ လုပ်ပေးမှာပါ</u>။

Error ရှိတဲ့အခါမှာလည်း `error.message` နဲ့ မူလ Error Message ကိုပဲ ပြန်ပြထားတာ သတိပြုပါ။ ကျန်အလုပ်လုပ်ပုံတွေက မူလအတိုင်း အတူတူပဲ ဆက်လက်အလုပ်လုပ်နေတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။

React Query က Data Fetching မှာ Error တွေ Loading တွေကို Handle လုပ်ပေးယုံမကပါဘူး။ တခြား လုပ်ပေးတာတွေ အများကြီးရှိပါသေးတယ်။

- Network Timeout ဖြစ်နေရင် Retry (၃) ကြိမ်လုပ်ပေးပါတယ်။ အကြိမ်အရေအတွက်ကို ကိုယ့်ဘာသာ ပြင်ဆင်သတ်မှတ်လို့လည်း ရပါတယ်။
- Fetch လုပ်ထားတဲ့ ရလဒ်ကို Cache လုပ်ပြီး သိမ်းပေးပါတယ်။ ဒါကြောင့် နောင် Fetch ထပ်လုပ်ဖို့ လိုအပ်လာတဲ့အခါ စောင့်စရာမလိုဘဲ သိမ်းထားပေးတဲ့ Data တွေကို ယာယီအသုံးပြုပေးနိုင်ပါတယ်။ Update ရပြီဆိုတော့မှ Update နဲ့ အစားထိုးပေးလိုက်မှာပါ။
- App ကို Focus ဖြစ်သွားတဲ့အခါ အလိုအလျှောက် Revalidate လုပ်ပေးပါတယ်။ အလိုအလျှောက် Data Fetch ပြန်လုပ်ပေးတယ်လို့ ဆိုလိုတာပါ။ ဒါကြောင့် App ကို ဖွင့်ထားပြီး တခြားနေရာတွေ လျှောက်သွားနေရာပြီးမှ ပြန်လာသုံးတဲ့အချိန်မှာ Data တွေက Outdate ဖြစ်နေတာမျိုး မဖြစ်တော့ပါဘူး။
- Network Restore ဖြစ်တဲ့အခါမှာလည်း အလိုအလျှောက် Revalidate လုပ်ပေးပါတယ်။ ဒါကြောင့် လိုင်းခဏကျနေပြီး ပြန်တက်လာချိန်မှာ Update ကို ရသွားမှာပါ။

စသည်ဖြင့် နောက်ကွယ်ကနေ အလိုအလျှောက်လုပ်ပေးသွားတာတွေ ရှိနေပါတယ်။ ဒါတွေကို ကိုယ့်ဘာသာ အကုန်လိုက်လုပ်ရရင် မလွယ်ပါဘူး။

ဒါက Data Fetching ပဲ ရှိပါသေးတယ်။ Add တို့ Delete တို့လို Data ကိုပြောင်းလဲစေတဲ့ လုပ်ဆောင်ချက်တွေ ဘယ်လိုလုပ်ရမလဲ ဆက်ကြည့်ကြပါမယ်။ `Home.jsx` မှာ `useMutation` နဲ့ `queryClient` တို့ကို အခုလိုထည့် Import လုပ်ပေးပါ။

React - src/pages/Home.jsx

```
import { useQuery, useMutation } from "react-query";
import { queryClient } from "../ThemedApp";
```

ပြီးရင် အရင်ရေးလက်စ `remove` Function အခုလို ပြင်ရေးရပါမယ်။

React - src/pages/Home.jsx

```
const remove = useMutation(
  async id => {
    await fetch(`${api}/content/posts/${id}`, {
      method: "DELETE",
    });
  },
  {
    onMutate: id => {
      queryClient.cancelQueries("posts");
      queryClient.setQueryData("posts", old =>
        old.filter(item => item.id !== id)
      );
      setGlobalMsg("A post deleted");
    },
  }
);
```

`useMutation` အတွက် Parameter နှစ်ခုပေးထားပါတယ်။ ပထမတစ်ခုက Data Fetch Function ပါ။ Data Fetch ဆိုပေမယ့် အသစ်ထည့်တာ၊ ပြင်တာ၊ ဖျက်တာလို အပြောင်းအလဲတစ်ခုခုကို ဖြစ်စေတဲ့ Fetch လုပ်ဆောင်ချက်တွေပါ။ API ကို `DELETE` Method နဲ့ Request လုပ်ပြီး Post ကို ဖျက်ထားပါတယ်။

ဒုတိယ Parameter က Side Effect Option တွေကို ပေးရတာပါ။ Data Fetch Success ဖြစ်တဲ့အခါ လုပ်စေချင်တဲ့ အလုပ်တွေရှိရင် `onSuccess` နဲ့ ရေးပေးလို့ ရပါတယ်။ နမူနာမှာ Success ဖြစ်အောင် စောင့်မနေဘဲ ချက်ခြင်းလုပ်စေချင်လို့ `onMutate` နဲ့ ရေးထားပါတယ်။ `onMutate` က Data Fetch Function ရဲ့ Variable တွေကို လက်ဆင့်ကမ်းယူလို့ရတဲ့အတွက် `id` ကို ပြန်ယူသုံးထားပါတယ်။

`queryClient` ကိုသုံးပြီး `posts` Query Key အတွက် လက်ရှိလုပ်လက်စ Query တွေရှိရင် ရပ်ခိုင်း

ထားပါတယ်။ ပြီးတဲ့အခါ setQueryData နဲ့ လက်ရှိ သိမ်းထားတဲ့ Data ကနေ ဖယ်ထုတ်ချင်တဲ့ Item ကို filter နဲ့ ဖယ်ထုတ်ပြီး Update လုပ်ပေးထားပါတယ်။ ဒါကြောင့် Data ကို အသုံးပြုဖော်ပြနေတဲ့ Component လည်း မလိုအပ်တဲ့ Item ကို ဖယ်ထုတ်ပြီး အလိုအလျှောက် ပြပေးသွားမှာ ဖြစ်ပါတယ်။ စမ်းကြည့်လို့ရဖို့အတွက် <Item> အတွက် remove ကို Pass လုပ်ပုံလုပ်နည်း အခုလို ပြောင်းပေးဖို့ လိုပါသေးတယ်။

React - src/pages/Home.jsx

```
<Item
  key={item.id}
  item={item}
  remove={remove.mutate}
/>
```

useMutation() က mutation Object ကိုပြန်ပေးမှာပါ။ ဒါကြောင့် ဒီအတိုင်းသုံးလို့ မရပါဘူး။ ပြန်သုံးတဲ့အခါ သူ့ရဲ့ mutate() Method ကို သုံးရပါတယ်။ ဒါကြောင့် remove အတွက် remove.mutate ကို Pass လုပ်ပေးလိုက်ရမှာပါ။ စမ်းကြည့်လို့ရပါပြီ။

ဒါဟာ Server-side မှာ အပြောင်းအလဲတစ်ခုခုကို ဖြစ်ပေါ်စေတဲ့ API Request တွေကို React Query နဲ့ ပြုလုပ်ပုံပြုလုပ်နည်းပဲဖြစ်ပါတယ်။ Post အသစ်တွေထည့်လို့ရတဲ့ လုပ်ဆောင်ချက်ကိုလည်း အလားတူ ရေးပေးဖို့လိုပေမယ့် အသင့်မဖြစ်သေးပါဘူး။ နောက်မှ ဆက်ရေးကြပါမယ်။ အခု အလားတူ လုပ်ဆောင်ချက်မျိုးတွေ Comments.jsx မှာ သွားရေးကြပါမယ်။ ဒီလိုပါ –

React - src/pages/Comments.jsx

```
import { Box, Button, TextField, Alert } from "@mui/material";

import Item from "../components/Item";

import { useParams, useNavigate } from "react-router-dom";
import { useQuery, useMutation } from "react-query";
import { queryClient } from "../ThemedApp";
```

```
import { useApp } from "../ThemedApp";

const api = import.meta.env.VITE_API;

export default function Comments() {
  const { id } = useParams();
  const navigate = useNavigate();

  const { setGlobalMsg } = useApp();

  const { isLoading, isError, error, data } = useQuery(
    "comments",
    async () => {
      const res = await fetch(`${api}/content/posts/${id}`);
      return res.json();
    }
  );

  const removePost = useMutation(async id => {
    await fetch(`${api}/content/posts/${id}`, {
      method: "DELETE",
    });

    navigate("/");
    setGlobalMsg("A post deleted");
  });

  const removeComment = useMutation(
    async id => {
      await fetch(`${api}/content/comments/${id}`, {
        method: "DELETE",
      });
    },
    {
      onMutate: id => {
```

```
        queryClient.cancelQueries("comments");
        queryClient.setQueryData("comments", old => {
          old.comments = old.comments.filter(
            comment => comment.id !== id
          );
          return { ...old };
        });
        setGlobalMsg("A comment deleted");
      },
    }
  );

  if (isError) {
    return (
      <Box>
        <Alert severity="warning">{error.message}</Alert>
      </Box>
    );
  }

  if (isLoading) {
    return <Box sx={{ textAlign: "center" }}>Loading...</Box>;
  }

  return (
    <Box>
      <Item
        primary
        item={data}
        remove={removePost.mutate}
      />
      {data.comments.map(comment => {
        return (
          <Item
            comment
```

```
            key={comment.id}
            item={comment}
            remove={removeComment.mutate}
          />
        );
      })}

      <form>
        <Box
          sx={{
            display: "flex",
            flexDirection: "column",
            gap: 1,
            mt: 3,
          }}>
          <TextField
            multiline
            placeholder="Your Comment"
          />
          <Button
            type="submit"
            variant="contained">
            Reply
          </Button>
        </Box>
      </form>
    </Box>
  );
}
```

တစ်ပိုင်းချင်း ခွဲမပြောတော့ဘဲ ရေးရမယ့်ကုဒ်တွေ အကုန်ထည့်ပေးထားပါတယ်။ `useQuery` နဲ့ Post တစ်ခုကို ယူလိုက်တဲ့အခါ သက်ဆိုင်ရာ Comment တွေပါ ပါလာမှာပါ။ ဒါကြောင့် ပထမဆုံး `<Item>` တစ်ခုကို `primary` အနေနဲ့ပြလိုက်ပါတယ်။ ပြီးတဲ့အခါ `post.comments` တွေကို `map()` လုပ်ပြီး `comment <Item>` တွေအနေနဲ့ ဆက်ပြထားပါတယ်။

useMutation နဲ့ removePost နဲ့ removeComment နှစ်ခုရေးထားပါတယ်။ removePost မှာ API ခေါ်ပြီး ဘာမှဆက်မလုပ်တော့ဘဲ Home ကိုပြန်သွားပါတယ်။ removeComment မှာ တော့ Home.jsx မှာ ရေးခဲ့တဲ့နည်းအတိုင်းပဲ queryClient အကူအညီနဲ့ Data ကို Update လုပ်ပေးလိုက်တာပါ။ ကျန်ကုဒ်တွေကိုတော့ ကုဒ်အတိုင်းပဲ လိုက်ဖတ်ကြည့်လိုက်ရင် ရမယ်ထင်ပါတယ်။

စမ်းကြည့်လို့ရဖို့အတွက် Item.jsx ကို နည်းနည်းပြင်ရပါဦးမယ်။

React - src/components/Item.jsx

```
// other imports

import { useNavigate } from "react-router-dom";

export default function Item({ item, remove, primary, comment }) {
  const navigate = useNavigate();

  return (
  <Card sx={{ mb: 2 }}>
    ...

    <CardContent
      onClick={() => {
          if(comment) return false;
          navigate(`/comments/${item.id}`);
        }}
      sx={{ cursor: "pointer" }}>
      ...
    </CardContent>
  </Card>
  );
}
```

`comment` Props ကို ထပ်တိုးပြီး လက်ခံထားပါတယ်။ ပြီးတဲ့အခါ `CardContent` ရဲ့ `onClick` မှာ `navigate` နဲ့ Comment Page ကို သက်ဆိုင်ရာ `id` နဲ့ သွားထားပါတယ်။ စမ်းကြည့်လို့ရပါပြီ။

ဒီလောက်ဆိုရင် React Query ရဲ့အခြေခံအသုံးပြုပုံရသွားပါပြီ။ Post အသစ်၊ Comment အသစ်ထည့်တာတွေ အပါအဝင် ကျန်နေတဲ့ လုပ်ဆောင်ချက်တွေကို နောက်အခန်းတွေမှာ ဆက်ကြည့်ကြပါမယ်။

React Query မှာ React Query Devtools ခေါ် အသုံးဝင်တဲ့ လုပ်ဆောင်ချက်တစ်ခုလည်း ရှိပါသေးတယ်။ လက်ရှိ Run လက်စ Query တွေရဲ့အခြေအနေ၊ Cache Data တွေရဲ့အခြေအနေတွေကို စီစစ်လေ့လာလို့ရပါတယ်။ အသုံးပြုနည်းက လွယ်ပါတယ်။ ကိုယ့်ဘာသာ ဒီလိပ်စာမှာ ဆက်လက်လေ့လာ စမ်းသပ်ကြည့်လိုက်ပါ။

https://tanstack.com/query/latest/docs/framework/react/devtools

အခန်း (၁၃) - React Native - Expo, Expo Router

React အကြောင်း သိသင့်တာလေးတွေ စုံသလောက် ဖြစ်သွားပါပြီ။ လုပ်လက်စပရောဂျက်လေး မပြီးသေးပေမယ့် နောက်မှဆက်လုပ်ကြပါမယ်။ အခုကြားဖြတ်ပြီး React Native အကြောင်း လေ့လာကြပါမယ်။ React Native ဟာ React ကို အသုံးပြုပြီး Cross-Platform Mobile App တွေ ရေးသားဖို့ အသုံးပြုနိုင်တဲ့ နည်းပညာဖြစ်ပါတယ်။ ပုံမှန်အားဖြင့် Mobile App တွေ ရေးသားဖန်တီးတဲ့အခါ သက်ဆိုင်ရာ Mobile Platform ရဲ့ SDK တွေကို အသုံးပြုဖန်တီးရမှာပါ။ ဥပမာ – Android SDK, iOS SDK စသည်ဖြင့်။

React Native မတိုင်ခင်က Appcelerator, Cordova စသည်ဖြင့် Web Technology တွေကိုအသုံးပြုပြီး Mobile App တွေ ဖန်တီးလို့ရတဲ့ Hybrid နည်းပညာတွေ ပေါ်ခဲ့ကြပါတယ်။ အသင့်အတင့်လောက်ပဲ အဆင်ပြေပြီး ရေရှည်မှာ အဆင်မပြေခဲ့ကြပါဘူး။ ဒီနည်းပညာတွေက ရေးတဲ့အခါ Web Technology တွေကို အသုံးပြုရေးသားသလို၊ အလုပ်လုပ်တဲ့အခါမှာလည်း Web App တစ်ခုကဲ့သို့ပဲ အလုပ်လုပ်ပါတယ်။ Browser တော့ မလိုပါဘူး။ ရိုးရိုး App တစ်ခုလို App Store ကနေ ဒေါင်းပြီး Install လုပ်လို့ရပေမယ့် အချိန်တန်တော့ Web App တစ်ခုကဲ့သို့ပဲ အလုပ်လုပ်သွားမှာ ဖြစ်ပါတယ်။ ဒါကြောင့် အားသာချက်အနေနဲ့ Cross-Platform ခေါ် တစ်ကြိမ်ရေးပြီး Mobile Device အမျိုးမျိုးမှာ Run လို့ရနိုင်ပေမယ့်၊ အားနည်းချက်အနေနဲ့ သူ့သဘာဝ Native App တွေလောက် အသုံးပြုရတာ ချောချောမွေ့မွေ့ မရှိကြပါဘူး။

နောက်တော့ NativeScript, Xamarin, React Native, Flutter စသည်ဖြင့် အလားတူ နည်းပညာကို၊ မတူတဲ့ ချဉ်းကပ်ပုံနဲ့ မျိုးဆက်သစ် Framework တွေ ထွက်ပေါ်လာကြပါတယ်။ ဒီထဲကမှလက်ရှိ အဓိကအကျဆုံး နည်းပညာများအဖြစ် React Native နဲ့ Flutter တို့က ရပ်တည်နေကြတာဖြစ်ပါတယ်။

React Native ဟာ သူ့ရှေ့ကမျိုးဆက်လိုပဲ App တွေ ရေးသားဖန်တီးတဲ့အခါ JavaScript (React) ကို အသုံးပြုရေးသားနိုင်ပါတယ်။ ဒါပေမယ့် အလုပ်လုပ်တဲ့အခါ သက်ဆိုင်ရာ Mobile Platform ရဲ့ Native Component တွေကို အသုံးပြုပြီး အလုပ်လုပ်ပေးနိုင်ပါတယ်။ Web App က Native Mobile App တစ်ခု လို ဟန်ဆောင် အလုပ်လုပ်တာမျိုး မဟုတ်တော့ဘဲ၊ တစ်ကယ့် Native Component တွေနဲ့ အလုပ်လုပ် သွားတာပါ။ ဒါကြောင့် သူ့ရှေ့ကမျိုးဆက်လို Native App နဲ့မတူတဲ့ ပြဿနာတွေ၊ အသုံးပြုရတာ ချော ချောမွေ့မွေ့မရှိတဲ့ပြဿနာတွေ ပြေလည်သွားပါတယ်။

နောက်ဆက်တွဲ ပြဿနာတစ်ခုတော့ ပါလာပါတယ်။ အရင်မျိုးဆက်လို ၁၀၀% Cross-Platform မဖြစ်နိုင် တော့ပါဘူး။ သက်ဆိုင်ရာ Mobile Platform ရဲ့ Native Component တွေကို အသုံးပြုအလုပ်လုပ်တာဖြစ် လို့ ပလက်ဖောင်းမတူတဲ့အခါ လုပ်ဆောင်ချက်ဆင်တူတာတွေ ရှိသလို လုံးဝမတူတာတွေ ရှိလာပါတယ်။ ဥပမာ – Android မှာပဲရှိပြီး iOS မှာမရှိတဲ့ လုပ်ဆောင်ချက်ကို အသုံးပြုရေးသားထားရင် ဒီ App ဟာ iOS မှာ အလုပ်လုပ်နိုင်မှာ မဟုတ်ပါဘူး။ နှစ်ခုလုံးမှာ အလုပ်လုပ်စေချင်ရင် နှစ်ခုလုံးမှာရှိပြီး ဆင်တူတဲ့ လုပ်ဆောင်ချက်တွေကိုပဲ အသုံးပြုရပါမယ်။ ဒါဆိုရင်လည်း အကန့်အသတ်တွေ များသွားပြီး လွတ်လွတ် လပ်လပ် ကိုယ့်စိတ်ကူးရှိတဲ့အတိုင်း ဖန်တီးလို့ရဖို့ ခက်သွားပါလိမ့်မယ်။

အားသာချက် အားနည်းချက်တွေကတော့ အားလုံးမှာ ကိုယ်စီနဲ့ပါပဲ။ အရည်အသွေးနဲ့ စွမ်းဆောင်ရည် Perfect ရမှလိုချင်ရင် Native SDK တွေကို သုံးရပါလိမ့်မယ်။ ဒါပေမယ့် Platform နှစ်ခု Support လုပ်ချင် ရင် နှစ်ခါရေးရမှာဖြစ်ပါတယ်။ ရေးတဲ့အခါ ရေးပြီး ပြီးသွားတာ မဟုတ်ပါဘူး။ ဆက်လက် Maintain လုပ်ရ ဦးမှာပါ။ Code Base နှစ်ခုနဲ့ နှစ်ကြိမ်လုပ်ရပါလိမ့်မယ်။ ဒါတောင် Major Mobile Platform က နှစ်ခုပဲ ရှိ လို့ တော်သေးတာပါ။

React Native နဲ့လည်း Platform နှစ်ခု Support လုပ်ချင်ရင် နှစ်ခါရေးမှာပါပဲ။ ဒါပေမယ့် အစအဆုံး အကုန်ပြန်ရေးစရာ မလိုတော့ပါဘူး။ တော်တော်များများကို ပြန်သုံးလို့ရနိုင်ပါလိမ့်မယ်။ ဒါကြောင့် Mobile App တွေ ရေးသားဖန်တီးမှုနဲ့ပက်သက်ရင် နည်းလမ်းတစ်မျိုးထဲကိုပဲ အများစုသုံးကြတယ်ဆိုတာမျိုး မဟုတ်ဘဲ၊ Native ပဲရေးကြသူတွေ၊ React Native သုံးကြသူတွေ၊ Flutter သုံးကြသူတွေ အားလုံး အချိုးကျ သူ့နေရာနဲ့သူ ရှိနေကြတဲ့သဘော ဖြစ်နေပါတယ်။

React Native က React အခြေခံရှိပြီးသူအတွက် အသုံးပြုရ လွယ်ကူပါတယ်။ သိပ်အများကြီး ထပ်လေ့လာစရာမလိုဘဲ ကောက်ကိုင်လို့ ရနိုင်ပါတယ်။ ထူးခြားချက်လေးတွေ သတိပြုဖို့ပါပဲ။ ပထမဆုံး ထူးခြားချက်ကတော့ ရိုးရိုး React ပရောဂျက်တွေမှာ Component တွေလိုအပ်ရင် ကြိုက်တဲ့ HTML Element ကို ကောက်သုံးလိုက်လို့ ရနိုင်ပေမယ့် <u>React Native မှာ HTML Element တွေကို အသုံးပြုလို့ ရမှာမဟုတ်ပါဘူး။ သူသတ်မှတ်ပေးထားတဲ့ Component တွေကိုပဲ အသုံးပြုလို့ရမှာပါ</u>။

အခြေခံအကျဆုံး Component တွေကတော့ `<View>` နဲ့ `<Text>` ဖြစ်ပါတယ်။ စာမှန်သမျှ `<Text>` နဲ့ ပြရပါတယ်။ `<Text>` မပါဘဲ စာတွေထည့်လို့ မရပါဘူး။ `<View>` ကိုတော့ `<div>` Element နဲ့ဆင်တူတယ်လို့ သဘောထားနိုင်ပါတယ်။ လိုတဲ့နေရာတိုင်းမှာ အသုံးပြုနိုင်ပါတယ်။

နောက်ထပ်ဖြည့်စွက်မှတ်သားရမယ့် အခြေခံ Component တွေကတော့ `<Image>` နဲ့ `StyleSheet` ဖြစ်ပါတယ်။ `<Image>` ကို ပုံတွေပြဖို့ အသုံးပြုရမှာဖြစ်ပါတယ်။ React Native Component တွေကို CSS နဲ့ ဆင်တူတဲ့ရေးနည်းအတိုင်း အသွင်အပြင်ဖော်ပြပုံတွေ စိတ်တိုင်းကျ သတ်မှတ်နိုင်ပါတယ်။ ဒီအတွက် `StyleSheet` ကို အသုံးပြုနိုင်ပါတယ်။ အများအားဖြင့် CSS နဲ့ဆင်တူပေမယ့် အတိအကျတော့ တူမှာ မဟုတ်ပါဘူး။ နမူနာလေးတွေ ရေးကြည့်တဲ့အခါ တွေ့ရပါလိမ့်မယ်။ Layout တွေ ထည့်သွင်းအသုံးပြုဖို့ အတွက်လည်း CSS Flex Box ရေးနည်းနဲ့ပဲ ရေးသားစီမံနိုင်ပါတယ်။

နောက်ထပ် မှတ်သားရမယ့် Component တွေကတော့ `<ScrollView>`, `<TextInput>` နဲ့ `<Button>` တို့ပဲဖြစ်ပါတယ်။ ရိုးရိုး `<View>` က ဆန့်သလောက်ပဲ ပြမှာပါ။ ကျော်သွားရင် Scroll လုပ်ခွင့်ပေးမှာ မဟုတ်ပါဘူး။ Scroll လုပ်ဖို့လိုရင် `<ScrollView>` ကို အစားထိုး အသုံးပြုရမှာပါ။ `<TextInput>` ကတော့ User Input တွေရိုက်ထည့်စေလိုတဲ့အခါ သုံးနိုင်ပါတယ်။ `<Button>` ကတော့ Style လုပ်ရခက်လို့ သိပ်သုံးဖြစ်မှာ မဟုတ်ပါဘူး။ Native Button မို့လို့ ကောင်းတော့ ကောင်းပါတယ်။ ဒါပေမယ့် သူပြတဲ့အတိုင်းပဲ ရပါမယ်။ ကိုယ့်ဘာသာ အသေးစိတ် Style လုပ်ဖို့လိုတဲ့နေရာတွေမှာ သူ့အစား `<TouchableOpacity>` လို Component မျိုးကို အစားထိုး အသုံးပြုကြပါမယ်။

တခြား `<Switch>`, `<FlatList>` စတဲ့ အခြေခံ Component တွေလည်းရှိသေးပေမယ့် နမူနာတွေမှာ သိပ်သုံးဖြစ်မှာ မဟုတ်ပါဘူး။ ဒါကြောင့် လိုအပ်လာမှ ဆက်လေ့လာပြီး အသုံးပြုကြရမှာ ဖြစ်ပါတယ်။ React ပရောဂျက်မှာ HTML Element တွေကို ကိုယ့်ဘာသာ ထည့်သုံးမယ့်အစား MUI Component တွေ

သုံးခဲ့သလိုပဲ၊ React Native ရဲ့ အခြေခံ Component တွေကိုမသုံးဘဲ UI Framework တွေကပေးတဲ့ အသင့်သုံး Component တွေကိုလည်း အသုံးပြုနိုင်ပါတယ်။ Theme တွေ Color တွေ Manage လုပ်နိုင်ခြင်းအပါအဝင် နေရာတကာ ကိုယ့်ဘာသာ အကုန်ပြန်လုပ်စရာမလိုလို့ အသုံးဝင်နိုင်ပါတယ်။ ဒီအတွက် React Native Elements လို နည်းပညာမျိုးကို နောင်လိုအပ်ရင် လေ့လာအသုံးပြုနိုင်ပါတယ်။

React Native Elements – https://reactnativeelements.com

ဒီသင်ခန်းစာမှာတော့ အခြေခံ Component တွေကို ကိုယ့်စိတ်တိုင်းကျ Style လုပ်ပြီး ဆက်လေ့လာသွားကြမှာဖြစ်ပါတယ်။ ဒါပေမယ့် တစ်ခုမကျန် အကုန်ကိုယ့်ဘာသာလုပ်မှာတော့ မဟုတ်ပါဘူး။ လုပ်စရာလည်း မလိုပါဘူး။ Expo လို့ခေါ်တဲ့ အလွန်အသုံးဝင်တဲ့ နည်းပညာတစ်ခု ရှိနေပါတယ်။

Expo ဟာ UI Framework မဟုတ်ပါဘူး။ ဒါပေမယ့် React Native ပရောဂျက်တစ်ခု စတင်ရေးသားခြင်း၊ စမ်းသပ်ခြင်း၊ ပြီးတဲ့အခါ Publish လုပ်ခြင်း၊ Update လုပ်ခြင်းတွေထိ Process တစ်ခုလုံးကို အစအဆုံး Manage လုပ်ပေးနိုင်တဲ့ နည်းပညာ ဖြစ်ပါတယ်။ အခြေခံလိုအပ်ချက်တွေ ဖြစ်ကြတဲ့ Routing တို့ Icons တို့လို လုပ်ဆောင်ချက်တွေ အသင့် စီစဉ်ထည့်သွင်းပေးထားသလို၊ Splash Screen, Sensor, Camera, Gesture, Animation စသည်ဖြင့် App တစ်ခုမှာလိုအပ်နိုင်တဲ့ လုပ်ဆောင်ချက်တွေအများကြီးကို အသင့် ပံ့ပိုးပေးထားလို့ တော်တော်အသုံးဝင်တဲ့ နည်းပညာတစ်ခုဖြစ်ပါတယ်။ ကနေ့အချိန်မှာ React Native ပရောဂျက်တစ်ခုကို Expo မပါဘဲ ကိုယ့်ဘာသာ Manual Manage လုပ်ရင် သိပ်အဓိပ္ပါယ်မရှိတော့ပါဘူး။ ဒါကြောင့် Expo ကို အသုံးပြုပြီး ဆက်လက်လေ့လာသွားကြမှာ ဖြစ်ပါတယ်။

Expo နဲ့ React Native Project တစ်ခုကို အခုလို ဖန်တီးနိုင်ပါတယ်။

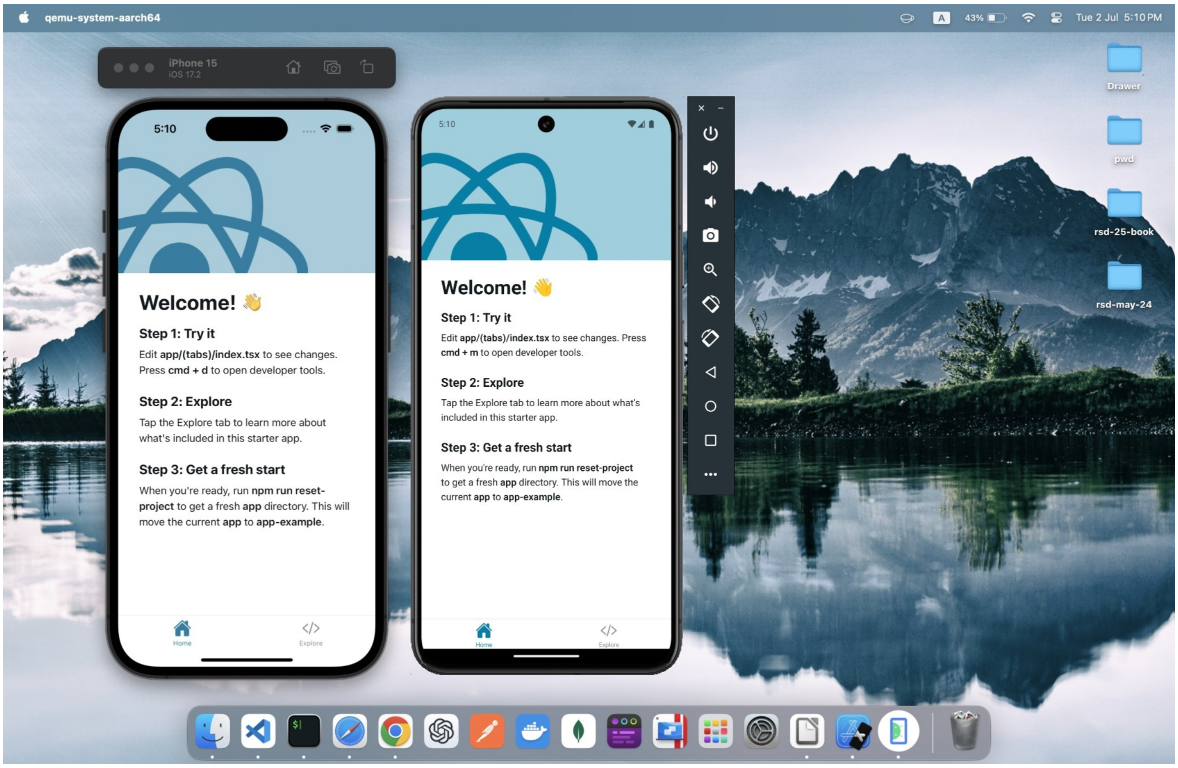
```
npx create-expo-app HelloNative

✔ Downloaded and extracted project files.
> npm install
...
```

ဒါဟာ `HelloNative` အမည်နဲ့ React Native ပရောဂျက်တစ်ခု ဖန်တီးလိုက်ခြင်း ဖြစ်ပါတယ်။ ပရောဂျက်ကို Run နည်းအမျိုးမျိုး ရှိပါတယ်။

ကိုယ့်စက်ထဲမှာ iOS Simulator တွေ Android Emulator တွေရှိထားရင်တော့ အဆင်အပြေဆုံးပါပဲ။ ဒါကြောင့် macOS အသုံးပြုနေသူများက Xcode နဲ့ Android Studio တို့ကို Install လုပ်ထားသင့်ပါတယ်။ Windows တို့ Linux တို့ကို အသုံးပြုနေသူများက Android Studio ကို Install လုပ်ထားသင့်ပါတယ်။ Xcode နဲ့အတူ iOS Simulator ကိုပါ ထည့်သွင်း Install လုပ်ရမှာဖြစ်ပြီး Android Studio နဲ့အတူ Android Emulator ကို ထည့်သွင်း Install လုပ်ရမှာပါ။ လိုအပ်တာတွေ Install လုပ်ပြီးသွားရင် `HelloNative` ဖိုဒါထဲမှာ ပရောဂျက်ကို အခုလို Run နိုင်ပါတယ်။

```
npx expo
```

ဒီလို Run လိုက်တဲ့အခါ လိုအပ်တဲ့ Instruction တွေကျလာပါလိမ့်မယ်။ Terminal ထဲမှာပဲ a ကိုနှိပ်လိုက်ရင် App ကို Android Emulator ထဲမှာ Run ပေးသွားပါလိမ့်မယ်။ Emulator ကို ကိုယ့်ဘာသာ ကြိုဖွင့်ပေးထားဖို့ လိုနိုင်ပါတယ်။ Terminal ထဲမှာပဲ `i` ကိုနှိပ်လိုက်ရင်တော့ iOS Simulator ထဲမှာ App ကို Run ပေးသွားပါလိမ့်မယ်။



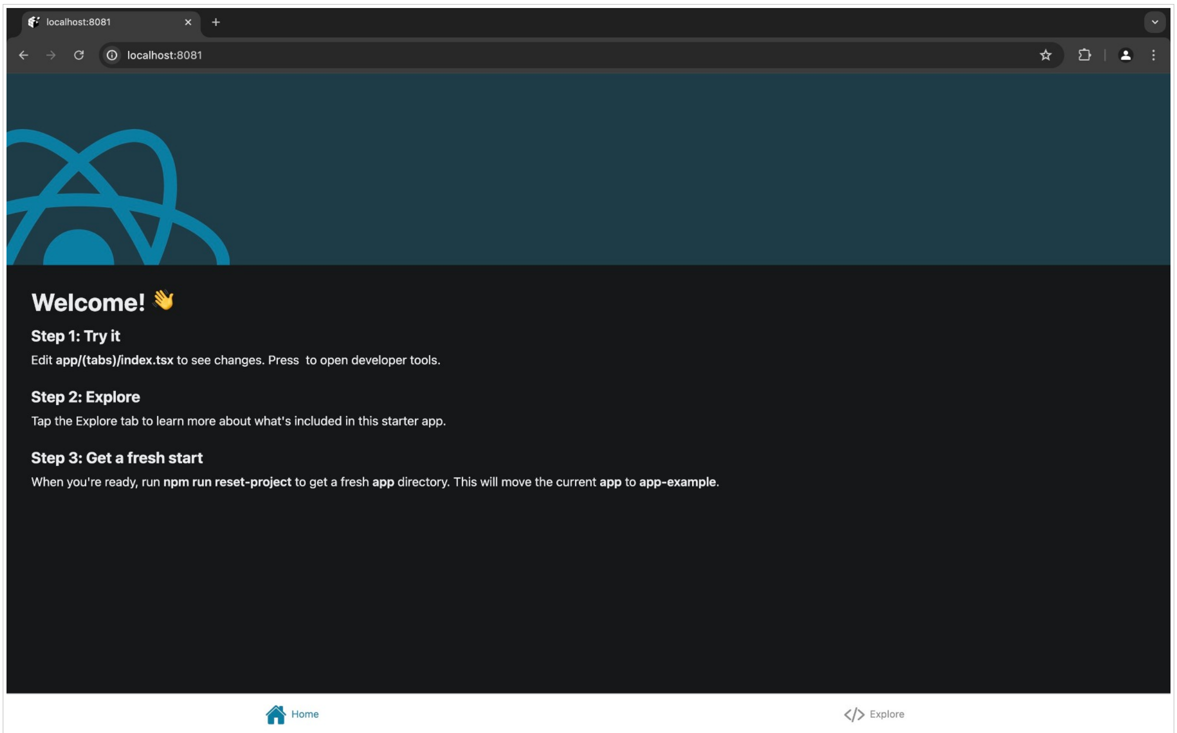
နောက်တစ်နည်းကတော့ App ကို ကိုယ့်ဖုန်းထဲမှာ Run တဲ့နည်းဖြစ်ပါတယ်။ ဒီလို Run လို့ရဖို့အတွက် ဖုန်းထဲမှာ Expo Go App ကို App Store (သို့မဟုတ်) Play Store ကနေရှာပြီး Install လုပ်ထားဖို့ လိုပါတယ်။ မှားမှာစိုးရင် ဒီလင့်ကနေ သွားဒေါင်းလို့လည်း ရပါတယ်။

Expo Go – https://expo.dev/go

Expo Go ကို ဒေါင်းပြီးရင် စောစောက ပရောဂျက်ကို Run ထားတဲ့ Terminal မှာ ကြည့်ပါ။ အခုလို QR Code တစ်ခု ရှိနေတာကို တွေ့ရပါလိမ့်မယ်။

Camera App (iOS) သို့မဟုတ် Expo Go App (Android) နဲ့ အဲ့ဒီ QR Code ကို Scan လုပ်လိုက်ပါ။ ဒါဆိုရင် App ကို ကိုယ့်ဖုန်းထဲမှာ Run ပြပေးသွားမှာ ဖြစ်ပါတယ်။ အဲ့ဒီလို Run နိုင်ဖို့အတွက် ပရောဂျက်ရှိနေတဲ့ ကွန်ပျူတာနဲ့ စမ်းချင်တဲ့ဖုန်းက Wifi Network တစ်ခုထဲမှာ အတူတူရှိနေရမှာ ဖြစ်ပါတယ်။ ပြီးတော့ ကွန်ပျူတာမှာရော၊ ဖုန်းမှာပါ VPN တွေရှိရင် ပိတ်ထားပေးဖို့ လိုနိုင်ပါတယ်။ ဒီနည်းက နည်းနည်းတော့ အလုပ်ရှုပ်ပါတယ်။ အရင်က ဒီနည်းကို ပိုအားပေးပေမယ့် အခုတော့ Emulator တွေ Simulator တွေနဲ့ Run တဲ့နည်းကိုပဲ ပိုအားပေးချင်ပါတယ်။

နောက်ဆုံးနည်းလမ်းတစ်ခု ရှိပါသေးတယ်။ အဲ့ဒါကတော့ Terminal မှာ w ကို နှိပ်ပြီး App ကို Browser ထဲမှာ Run လိုက်ခြင်းဖြစ်ပါတယ်။



ဒါကတော့ နောက်ဆုံးတခြားနည်းလမ်းတွေ ဘယ်လိုမှအဆင်မပြေတဲ့အခါ သုံးဖို့ပါ။ Web နဲ့ Mobile မတူတာတွေရှိလို့ အလုပ်လုပ်တဲ့ပုံစံကို တိတိကျကျစမ်းချင်ရင် Mobile နဲ့စမ်းမှပဲ ပိုအဆင်ပြေမှာမို့လို့ပါ။

React Native က iOS နဲ့ Android တင် Support လုပ်တာ မဟုတ်ပါဘူး။ Web နဲ့ Windows တို့ကိုလည်း Support လုပ်ပါတယ်။ ဆိုလိုတာက React Native နဲ့ Web App တွေ Windows App တွေလည်း ရေးလို့ရတဲ့သဘော ဖြစ်ပါတယ်။

လက်ရှိ App မှာ နမူနာအနေနဲ့ Bottom Navigation တစ်ခုပါပါတယ်။ Tab UI တစ်မျိုးပါပဲ။ နှိပ်ပြီး ဟိုဘက်ဒီဘက် Page တွေပြောင်းစမ်းကြည့်လို့ရပါတယ်။ Expo နဲ့ ပရောဂျက်တည်ဆောက်စဉ်မှာ Expo Router လို့ခေါ်တဲ့ နည်းပညာ တစ်ခါထဲပါသွားပါတယ်။ ပုံမှန်အားဖြင့် ရိုးရိုး React App တွေမှာ Multi-Page လုပ်ဆောင်ချက်အတွက် React Router ကို သုံးကြပြီး React Native App တွေမှာ React

Navigation ကို သုံးကြတာပါ။ အဲ့ဒီ React Navigation ကိုပဲ Expo က နည်းနည်းပြင်ပြီး Expo Router ဆို တဲ့အမည်နဲ့ ထည့်ပေးထားတာပါ။

ကုဒ်တချို့ ရေးစမ်းကြည့်ဖို့အတွက် နမူနာပါလာတဲ့ ကုဒ်တွေကို အရင်ရှင်းကြပါမယ်။ လက်ရှိ Run လက်စ Expo ကို ခဏရပ်ပြီးတော့ပဲဖြစ်ဖြစ်၊ နောက် Terminal တစ်ခုနဲ့ပဲဖြစ်ဖြစ် အခုလို Run ပေးပါ။

```
npm run reset-project

> hellonative@1.0.0 reset-project
> node ./scripts/reset-project.js

/app moved to /app-example.
New /app directory created.
app/index.tsx created.
app/_layout.tsx created.
```

ဒါ နမူနာကုဒ်တွေပါတဲ့ `app` ဖိုဒါကို ဖယ်လိုက်ပြီး အသစ်အလွတ် ပြန်လုပ်ပေးတာပါ။ `app` ဖိုဒါအသစ်ထဲ မှာ `index.tsx` နဲ့ `_layout.tsx` ဆိုတဲ့ ဖိုင်နှစ်ခု ပါဝင်ပါလိမ့်မယ်။ သူကတော့ TypeScript ကိုသုံးဖို့ ရည်ရွယ်ချက်နဲ့ File Extension တွေကို `tsx` နဲ့ပေးထားပါတယ်။ `jsx` နဲ့ ရေးရင်လည်း အတူတူပဲ အလုပ် လုပ်ပါတယ်။ ဒါကြောင့် `app` ဖိုဒါထဲက `index.tsx` နဲ့ `_layout.tsx` ကို ပြန်သာဖျက်ထားလိုက်ပါ။ နောက်မှ ကိုယ့်ဘာသာ ပြန်ရေးကြပါမယ်။

Page တစ်ခုချင်းစီကို `index.jsx`, `account.jsx`, `settings.jsx` စသည်ဖြင့် မိမိနှစ်သက် ရာအမည်နဲ့ ရေးနိုင်ပါတယ်။ ပုံမှန်အားဖြင့် Page Name ကို Component Name နဲ့တူအောင် စာလုံးကြီးနဲ့ စပြီး ပေးကြပေမယ့်၊ ဒီနေရာမှာတော့ Page Name ကို Route အနေနဲ့သုံးမှာဖြစ်လို့ စာလုံးသေးတွေပဲ ပေး သင့်ပါတယ်။ ဒီ Page တွေကို `_layout.jsx` ထဲမှာ ထည့်ပြမှာမို့လို့ အားလုံးမှာပါဝင်စေချင်တဲ့ လုပ်ဆောင်ချက်ရှိရင် `_layout.jsx` မှာ ရေးပေးလို့ရနိုင်ပါတယ်။ နမူနာအနေနဲ့ အသုံးများတဲ့ Bottom Navigation ကိုပဲ ပြန်ထည့်ကြည့်ကြပါမယ်။

`app` ဖိုဒါရဲ့ ဖွဲ့စည်းပုံက ဒီလိုဖြစ်သင့်ပါတယ်။

```
HelloNative
├── ...
├── app/
│   ├── (home)/
│   │   ├── _layout.jsx
│   │   ├── account.jsx
│   │   ├── index.jsx
│   │   └── settings.jsx
│   ├── _layout.jsx
│   └── about.jsx
└── ...
```

`app/_layout.jsx` က Main Layout ဖိုင်ဖြစ်ပြီး `app/(home)/_layout.jsx` က Sub-Layout ဖိုင်လို့ ဆိုနိုင်ပါတယ်။ `app/_layout.jsx` မှာ အခုလိုရေးပေးပါ။

React Native - app/_layout.jsx

```
import { Stack } from "expo-router";

export default function RootLayout() {
  return (
    <Stack>
      <Stack.Screen
        name="(home)"
        options={{ headerShown: false, title: "Home" }}
      />
      <Stack.Screen
        name="about"
        options={{ title: "About" }}
      />
    </Stack>
  );
}
```

Screen နှစ်ခုပါတဲ့ Layout Stack တစ်ခုသတ်မှတ်လိုက်တာပါ။ ပထမ Screen ရဲ့ name အတွက်

(home) ကို ပေးထားလို့ (home) ဖိုဒါထဲကကုဒ်တွေ အလုပ်လုပ်ပါလိမ့်မယ်။ ဒုတိယ Screen ရဲ့ name ကိုတော့ about လို့ပေးထားတဲ့အတွက် about.jsx ကို အလုပ်လုပ်ပါလိမ့်မယ်။

(home) ဖိုဒါကို ကွင်းစကွင်းပိတ် ထည့်ပေးတာ အကြောင်းရှိပါတယ်။ Route Group လုပ်လိုက်တာပါ။ အဲ့ဒီလို မပေးဘဲ ရိုးရိုးရေးရင်လည်းရတော့ရပါတယ်။ ဥပမာ - home/account.jsx ဖိုင်ရှိမယ်ဆိုရင် သူ့ရဲ့ Route Path က home/account ဖြစ်ပါတယ်။ (home)/account.jsx လို့ပေးထားတဲ့ အတွက် သူ့ရဲ့ Route Path က /account ဖြစ်ပါတယ်။ တနည်းအားဖြင့် ကွင်းစကွင်းပိတ်ထဲက Route Group ဖိုဒါရဲ့အမည်ကို Route Path ထဲမှာ ထည့်စရာမလိုခြင်း ဖြစ်ပါတယ်။ ဒါကြောင့် အခုဒီ ဖိုဒါ Structure ရဲ့ Route Path အပြည့်အစုံက ဒီလိုဖြစ်မှာပါ။

- / → (home)/index.jsx
- /account → (home)/account.jsx
- /settings → (home)/settings.jsx
- /about → about.jsx

Screen Options တွေ ထည့်ပေးထားတာကို သတိပြုပါ။ (home) အတွက် Screen Options မှာ headerShown ကို false လို့ပြောထားပါတယ်။ ဘာကြောင့်လဲ ပြောနေရင် နားရှုပ်ပါတယ်။ ခဏနေ တော့ ကိုယ့်ဘာသာ true လို့ပြောင်းပြီး စမ်းကြည့်လိုက်ပါ။

ဆက်လက်ပြီး about.jsx ဖိုင်မှာ အခုလိုရေးပေးပါ။

React Native - app/about.jsx

```
import { View, Text } from "react-native";

export default function about() {
  return (
    <View
      style={{
        height: "100%",
        justifyContent: "center",
        alignItems: "center",
```

```
      }}>
      <Text style={{ fontWeight: "bold" }}>About Us</Text>
    </View>
  );
}
```

View နဲ့ Text ကို React Native ကနေ Import လုပ်ယူပြီး သုံးထားပါတယ်။ ဘာလုပ်ဆောင်ချက်မှ ထူးထူးခြားခြားမပါပါဘူး။ `<View>` ထဲမှာ `<Text>` ကို ထည့်ပြလိုက်တာပါပဲ။ ထူးခြားချက်အနေနဲ့ `<View>` တွေဟာ မူလကတည်းက Flex Box တွေဖြစ်ကြပါတယ်။ ဒါကြောင့် `<View>` ရဲ့ `style` မှာ `alignItems` တို့ `justifyContent` တို့လို Flex Box Property တွေ ထည့်သုံးထားပါတယ်။ <u>`color` လို Text နဲ့ပက်သက်တဲ့ Property တွေကို View မှာပေးလို့မရတာ သတိပြုရပါမယ်</u>။ ရိုးရိုး HTML Elements တွေမှာဆိုရင် ပေးလို့ရပါတယ်။ `<div>` အတွက် color ပေးလိုက်ရင် အထဲက အဆင့်ဆင့်ရှိနေတဲ့ စာတွေမှာ အကုန်အသက်ဝင်သွားမှာပါပဲ။ React Native Component တွေမှာ Text နဲ့သက်ဆိုင်တဲ့ Property တွေကို `<Text>` မှာပဲ ရေးပေးရပါတယ်။ နမူနာမှာ `fontWeight` ကို ထည့်သုံးထားပါတယ်။

ဆက်လက်ပြီး `(home)/_layout.jsx` မှာ အခုလိုရေးပေးပါ။

React Native - app/(home)/_layout.jsx

```
import { Tabs } from "expo-router";
import { MaterialIcons } from "@expo/vector-icons";

export default function TabLayout() {
  return (
    <Tabs>
      <Tabs.Screen
        name="index"
        options={{
          title: "Home",
          tabBarIcon: () => (
            <MaterialIcons
              name="home"
```

```
              size={24}
            />
          ),
        }}
      />
      <Tabs.Screen
        name="account"
        options={{
          title: "Account",
          tabBarIcon: () => (
            <MaterialIcons
              name="person"
              size={24}
            />
          ),
        }}
      />
      <Tabs.Screen
        name="settings"
        options={{
          title: "Settings",
          tabBarIcon: () => (
            <MaterialIcons
              name="settings"
              size={24}
            />
          ),
        }}
      />
    </Tabs>
  );
}
```

ကုဒ်တွေများတယ်ထင်ရပေမယ့် သေချာလေ့လာကြည့်ရင် Tab Layout တစ်ခုကို Screen (၃) ခုနဲ့

သတ်မှတ်ပေးထားတာကို တွေ့ရပါလိမ့်မယ်။ Tab ခလုပ်တွေကို Icon လေးတွေနဲ့ ပြစေချင်တဲ့အတွက် `@expo/vector-icons` ကနေ `MaterialIcons` တွေကို Import လုပ်ထားပါတယ်။ တခြား `Ionicons` တို့ `Fontawesome` တို့အပါအဝင် အသင့်ထည့်သုံးလို့ရတဲ့ Icon တွေမှ အများကြီး ပါဝင်ပါတယ်။ အပြည့်အစုံကို အောက်ကလင့်မှာ ကြည့်နိုင်ပါတယ်။ Search Box မှာ လိုချင်တဲ့ Icon ကိုရှာပြီး သူပေးထားတဲ့ ကုဒ်နမူနာအတိုင်း ယူထည့်လိုက်ယုံပါပဲ။

Expo Vector Icons – https://icons.expo.fyi/Index

နမူနာမှာတော့ `MaterialIcons` ကိုထည့်သုံးထားပါတယ်။ MUI Icon တွေနဲ့ ပါတဲ့ Icon ချင်းဆင်တူပေမယ့် သုံးနည်းတော့ မတူပါဘူး။ MUI Icon တွေမှာ သုံးချင်တဲ့ Icon တစ်ခုချင်းစီကို Import လုပ်ရပြီး Expo Icon တွေကတော့ Icon Pack တစ်ခုကိုပဲ Import လုပ်ရပါတယ်။ သုံးတဲ့အချိန်ကျမှာ သုံးချင်တဲ့ Icon ရဲ့ name ကိုပေးရတာပါ။ Tab Screen တစ်ခုချင်းစီအတွက် မတူတဲ့ Icon တွေ ကိုယ်စီသတ်မှတ်ထားပါတယ်။ ဆက်လက်ပြီး `(home)/account.jsx` မှာ အခုလိုရေးပေးပါ။

```
React Native - app/(home)/account.jsx
```

```
import { View, Text } from "react-native";

export default function account() {
  return (
    <View
      style={{
        height: "100%",
        justifyContent: "center",
        alignItems: "center",
      }}>
      <Text style={{ fontWeight: "bold" }}>Account</Text>
    </View>
  );
}
```

ဘာလုပ်ဆောင်ချက်မှ မပါပါဘူး။ Account ဆိုတဲ့စာကို ပြလိုက်တာပါပဲ။ ဆက်လက်ပြီး `(home)/settings.jsx` မှာလည်း အလားတူ ရေးပေးပါ။

React Native - app/(home)/settings.jsx

```
import { View, Text } from "react-native";

export default function settings() {
  return (
    <View
      style={{
        height: "100%",
        justifyContent: "center",
        alignItems: "center",
      }}>
      <Text style={{ fontWeight: "bold" }}>Settings</Text>
    </View>
  );
}
```

Page တွေ Layout တွေ စုံသလောက်ဖြစ်သွားပါပြီ။ တစ်ခုပဲ ကျန်ပါတော့တယ်။ `(home)/index.jsx` မှာ အခုလိုရေးပေးပါ။

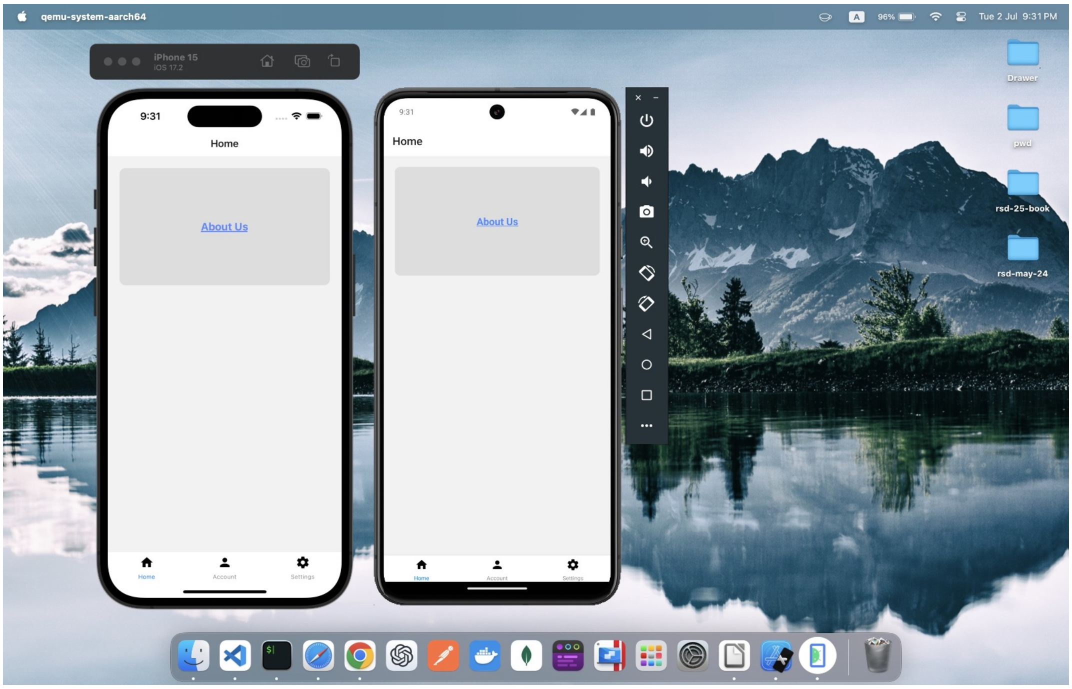
React Native - app/(home)/index.jsx

```
import { View, Text, StyleSheet } from "react-native";
import { router } from "expo-router";

export default function index() {
  return (
    <View style={styles.banner}>
      <Text
        style={styles.link}
        onPress={() => {
```

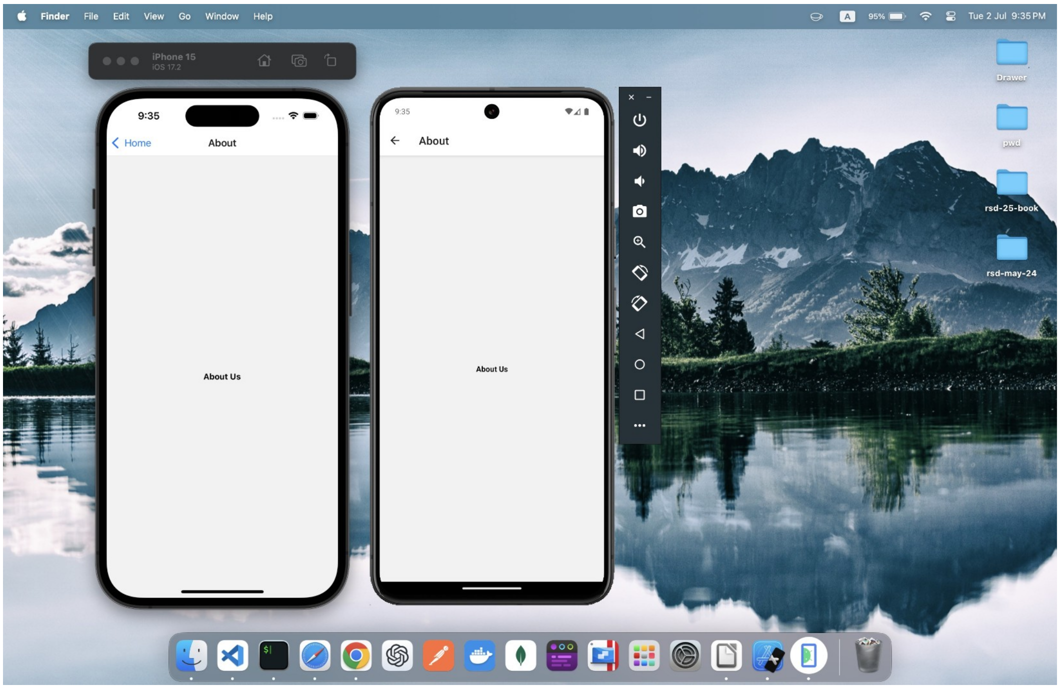
```
          router.navigate("/about");
        }}>
        About Us
      </Text>
    </View>
  );
}

const styles = StyleSheet.create({
  banner: {
    borderRadius: 10,
    margin: 20,
    backgroundColor: "#ddd",
    height: 200,
    alignItems: "center",
    justifyContent: "center",
  },
  link: {
    color: "#58f",
    fontSize: 18,
    fontWeight: "bold",
    textDecorationLine: "underline",
  },
});
```

ဒီတစ်ခါတော့ ထူးခြားချက်အနေနဲ့ `StyleSheet` ကို Import လုပ်ပြီး Style တွေကို သပ်သပ်ခွဲရေးထားပါတယ်။ ပြီးမှ လိုတဲ့နေရာမှာ ပြန်ချိတ်သုံးပါတယ်။ ပြီးတော့ `expo-router` ကနေ `router` ကို Import လုပ်ယူထားပါတယ်။ Text ကိုနှိပ်လိုက်ရင် `"/about"` ကိုသွားခိုင်းထားပါတယ်။ ထူးခြားချက်အနေနဲ့ နှိပ်လို့ရစေချင်ရင် `onClick` နဲ့ မရေးရဘဲ `onPress` နဲ့ရေးပေးရပါတယ်။

အားလုံးစုံပါပြီ။ Run ပြီးစမ်းကြည့်လိုက်ပါ။ အခုလို Tab (၃) ခုပါတဲ့ လုပ်ဆောင်ချက်ကို Home အနေနဲ့ ပြပေးသွားတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။



အောက်နား Bottom Navigation က Tab ခလုပ်တွေကိုနှိပ်ပြီး သက်ဆိုင်ရာ Page ကို ကြည့်လို့ရပါတယ်။



Header Bar တစ်ခုလည်း အပေါ်မှာ အလိုအလျှောက်ပါပြီး သက်ဆိုင်ရာ Page Title ကို ပြပေးပါလိမ့်မယ်။ Home က About Us စာကို နှိပ်ကြည့်လိုက်ရင်တော့ သီးခြား Page တစ်ခုကို တွေ့မြင်ရမှာပါ။

ဒီတစ်ခါတော့ Header Bar ထဲမှာ Back Button လည်း အလိုအလျှောက်ပါသွားတာကို တွေ့ရပါလိမ့်မယ်။ ဒါတင်မက Page Transaction Animation ကလည်း သက်ဆိုင်ရာ Mobile Platform မှာ သတ်မှတ်ထားတဲ့အတိုင်း အလုပ်လုပ်ပေးတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။ React Native ကို Expo Router နဲ့တွဲပြီး ရေးထားပေမယ့် လက်တွေ့အလုပ်လုပ်တဲ့အခါ Native Navigation နဲ့ အလုပ်လုပ်သွားတဲ့အတွက်ကြောင့်ပဲဖြစ်ပါတယ်။

အခန်း (၁၄) - React Native - Sample App UI

React နဲ့ React Native တို့ဟာ အလုပ်လုပ်တဲ့ ပလက်ဖောင်းမတူပေမယ့် ရေးနည်းက အများအားဖြင့် အတူတူပဲ ဖြစ်ပါတယ်။ Component တွေ Props တွေ State တွေ Manage လုပ်ပုံ၊ Context တွေ Provider တွေသုံးပုံ၊ Data Fetch လုပ်ပုံစသည်ဖြင့် အားလုံးအတူတူပါပဲ။ သူ့အကြောင်း ပိုသိသွားအောင် နမူနာလေးတစ်ခု လုပ်ကြည့်ကြပါမယ်။ iOS မှာရော Android မှာပါ အလုပ်လုပ်တဲ့ Cross-platform Pay App UI လေးတစ်ခု လုပ်ကြည့်ကြမှာပါ။

```
npx create-expo-app ypay
```

ypay အမည်နဲ့ ပရောဂျက်သစ်တစ်ခု စလိုက်တာပါ။ ပြီးတဲ့အခါ ပရောဂျက်ဖိုဒါထဲကို ဝင်ပြီး reset လုပ်လိုက်ပါ။

```
npm run reset-project
```

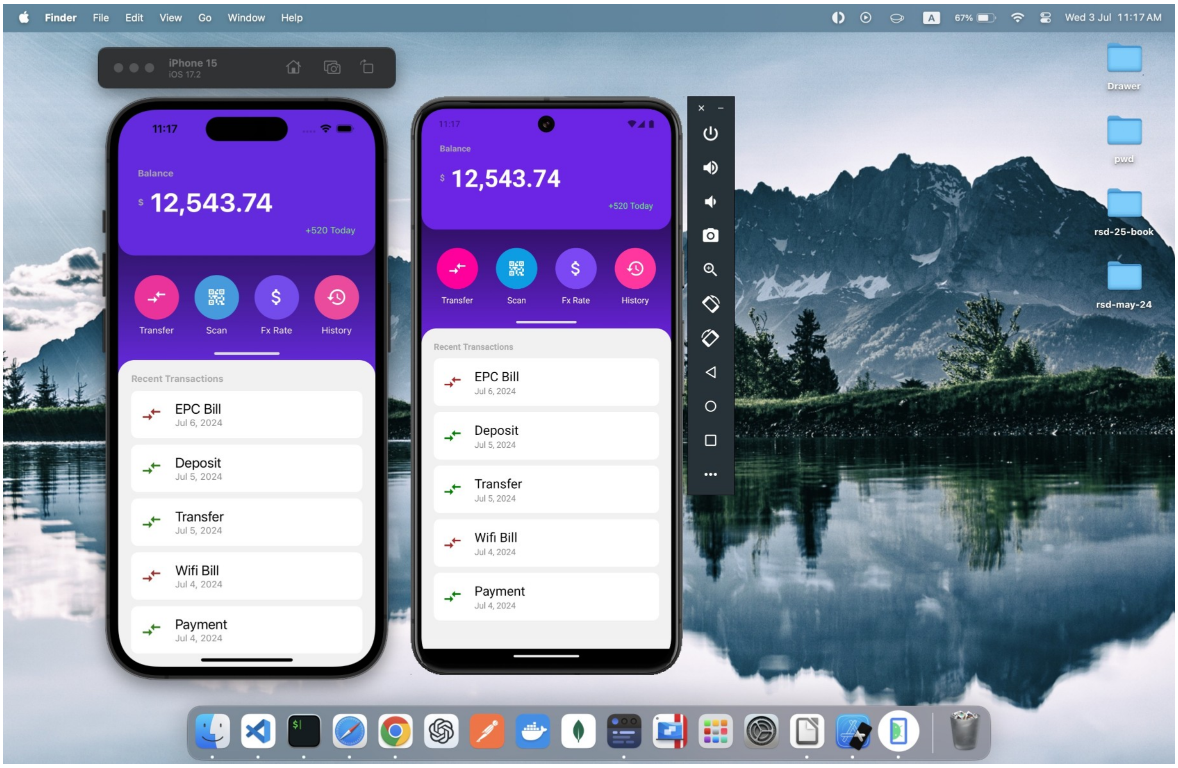
မူလနမူနာတွေ ရှင်းလိုက်တာပါ။ တစ်လက်စထဲ React Query ကို Install လုပ်လိုက်ပါ။ Data Fetching အတွက် React Query ကိုပဲဆက်သုံးမှာမို့လို့ပါ။

```
npm i react-query
```

နောက်ထပ်လိုအပ်မယ့် Package နှစ်ခုလည်း ထပ်ထည့်ပေးပါဦး။

```
npm i expo-camera expo-linear-gradient
```

`expo-camera` နဲ့ Device Camera ကိုသုံးပြီး QR Code Scan လုပ်တဲ့ လုပ်ဆောင်ချက်လေး ထည့်စမ်းမှာဖြစ်ပါတယ်။ CSS `linear-gradient` လုပ်ဆောင်ချက်ကို React Native မှာ မရနိုင်လို့ လိုအပ်တဲ့နေရာမှာ ထည့်သုံးလို့ရအောင် `expo-linear-gradient` ကိုလည်း တစ်ခါထဲ ကြိုထည့်ထားပါတယ်။ ဘယ်လို App မျိုး လုပ်မှာလဲ ကြိုသိရအောင် နောက်ဆုံးရမယ့် ရလဒ်လေး ကြိုပြထားချင်ပါတယ်။



တွေနေကြ Pay App တစ်ခုရဲ့ UI ဖြစ်ပြီး အပေါ်ဆုံးမှာ လက်ရှိ Balance ကို ပြထားပါတယ်။ အောက်နားမှာ လက်တလောလုပ်ထားတဲ့ Transaction တွေကိုပြထားပါတယ်။ အလယ်မှာ နှိပ်စမ်းကြည့်လို့ရမယ့် Transfer, Scan, Fx Rate, History စတဲ့ လုပ်ဆောင်ချက် (၄) ခုပါဝင်ပါတယ်။ အားလုံးတစ်ခုမကျန် အကုန်စုံအောင်တော့ မလုပ်ပါဘူး။ UI Prototype လောက်ပဲရအောင် လုပ်ကြမှာဖြစ်ပါတယ်။

ရေးစရာရှိတာ ဆက်ရေးကြပါမယ်။ app ဖိုဒါရဲ့ Structure ကို အခုလိုရအောင် လုပ်ပေးလိုက်ပါ။

```
ypay
├── ...
├── app/
│   ├── _layout.jsx
│   ├── components/
│   │   ├── ActionButton.jsx
│   │   ├── BalanceCard.jsx
│   │   └── Transactions.jsx
│   ├── fxrate.jsx
│   ├── history.jsx
│   ├── index.jsx
│   ├── scan.jsx
│   └── transfer.jsx
└── ...
```

`index`, `transfer`, `scan`, `fxrate`, `history` ဆိုတဲ့ Page (၅) ခုပါဝင်ပါတယ်။ ရိုးရိုး Stack Screen တွေပါပဲ။ Tab တွေဘာတွေ မသုံးထားပါဘူး။ ပင်မ `_layout.jsx` ဖိုင်တစ်ခုထဲပဲ ပါဝင်ပါတယ်။ အဲ့ဒီဖိုင်ထဲမှာ ဒီကုဒ်ကို ရေးပေးပါ။

React Native - app/_layout.jsx

```
import { Stack } from "expo-router";

import { QueryClient, QueryClientProvider } from "react-query";

export const queryClient = new QueryClient();

export default function RootLayout() {
  return (
    <QueryClientProvider client={queryClient}>
      <Stack>
        <Stack.Screen
          name="index"
```

```
          options={{ headerShown: false, title: "Home" }}
        />
        <Stack.Screen
          name="transfer"
          options={{ title: "Transfer" }}
        />
        <Stack.Screen
          name="scan"
          options={{ title: "Scan" }}
        />
        <Stack.Screen
          name="fxrate"
          options={{ title: "Exchange Rate" }}
        />
        <Stack.Screen
          name="history"
          options={{ title: "History" }}
        />
      </Stack>
    </QueryClientProvider>
  );
}
```

React Query ရဲ့ `QueryClienProvider` ထဲမှာ Screen (၅) ခုပါတဲ့ Stack Layout တစ်ခု သတ်မှတ်ထားတာပါ။ ကုဒ်တွေများတယ် ထင်ရပေမယ့် အသစ်အဆန်းတွေတော့ မပါတာကို တွေ့ရပါလိမ့်မယ်။ ဆက်လက်ပြီး `components` တွေ အရင်ရေးကြပါမယ်။ `components/ActionButton.jsx` မှာ အခုလိုရေးပေးပါ။

React Native - app/components/ActionButton.jsx

```
import { Text, View, StyleSheet, TouchableOpacity } from "react-native";

import { MaterialIcons } from "@expo/vector-icons";
```

```
import { router } from "expo-router";

export default function ActionButton({ color, icon, label, path }) {
  return (
    <TouchableOpacity
      style={styles.action}
      onPress={() => {
        if (path) {
          router.navigate(path);
        }
      }}>
      <View
        style={[
          styles.actionButton,
          { backgroundColor: color ? color : "#7b48f4" },
        ]}>
        <MaterialIcons
          name={icon ? icon : "attach-money"}
          size={32}
          color="white"
        />
      </View>
      <Text style={styles.text.actionText}>
        {label ? label : "Label"}
      </Text>
    </TouchableOpacity>
  );
}

const styles = StyleSheet.create({
  action: {
    justifyContent: "center",
    alignItems: "center",
    gap: 8,
  },
```

```
  actionButton: {
    width: 68,
    height: 68,
    borderRadius: 68,
    justifyContent: "center",
    alignItems: "center",
  },
  text: {
    actionText: {
      color: "#fff",
    },
  },
});
```

ဒါက App ရဲ့ အလယ်နားက နှိပ်လို့ရတဲ့ ခလုပ် (၄) ခုအတွက်ပြန်ခေါ်သုံးလို့ရအောင် Button Component တစ်ခု ကိုယ့်ဘာသာ ဖန်တီးလိုက်တဲ့သဘောပါပဲ။ `props` အနေနဲ့ `color`, `icon`, `label` နဲ့ `path` တို့ကို လက်ခံထားပါတယ်။

ထူးခြားချက်အနေနဲ့ `TouchableOpacity` Component ကိုသုံးထားတာ သတိပြုပါ။ နှိပ်လို့ရတဲ့ လုပ်ဆောင်ချက်တွေအတွက် `TouchableOpacity` ကို `Button` အစား သုံးနိုင်ပါတယ်။ ဒါကြောင့် ပြန်ပေးထားတာလည်း အဲ့ဒီ နှိပ်လို့ရတဲ့ `TouchableOpacity` ကို ပြန်ပေးထားပါတယ်။ `onPress` မှာ နှိပ်လိုက်ရင် `props` အနေနဲ့ ပါလာတဲ့ `path` ကို သွားထားပါတယ်။

`StyleSheet` နဲ့ `styles` တွေကို အောက်နားမှာ ခွဲရေးထားပါတယ်။ သတိပြုရမှာကတော့ `<View>` ရဲ့ style မှာ Array တစ်ခုပေးထားခြင်းဖြစ်ပါတယ်။ ဒီနည်းနဲ့ ကြိုရေးထားတဲ့ Styles တွေကို ပေါင်းစပ် အသုံးပြုလို့ ရပါတယ်။ နမူနာမှာ ကြိုရေးထားတဲ့ Style အပြင် Prop color ကိုပါ ထပ်တိုးပြီး ထည့်ပေးထားတာပါ။

ဆက်လက်ပြီး `components/BalanceCard.jsx` မှာ အခုလိုရေးပေးပါ။

React Native - app/components/BalanceCard.jsx

```
import { Text, View, StyleSheet } from "react-native";

export default function BalanceCard() {
  return (
    <View style={styles.banner}>
      <Text style={styles.text.label}>Balance</Text>
      <View style={styles.balance}>
        <Text style={styles.text.label}>$</Text>
        <Text style={styles.text.amount}>12,543.74</Text>
      </View>
      <View style={styles.balanceGrowth}>
        <Text style={styles.text.growth}>+520 Today</Text>
      </View>
    </View>
  );
}

const styles = StyleSheet.create({
  banner: {
    padding: 30,
    borderBottomLeftRadius: 25,
    borderBottomRightRadius: 25,
    backgroundColor: "#6d25e5",
  },
  balance: {
    marginTop: 12,
    flexDirection: "row",
    gap: 10,
    alignItems: "center",
  },
  balanceGrowth: {
    alignItems: "flex-end",
    paddingTop: 10,
  },
```

```
  text: {
    label: {
      fontWeight: "bold",
      color: "#aaa",
    },
    amount: {
      fontWeight: "bold",
      fontSize: 40,
      color: "#fff",
    },
    growth: {
      color: "#6f6",
    },
  },
});
```

ဒီမှာတော့ လိုချင်တဲ့ UI ရအောင် <View> တွေ <Text> တွေ Style တွေကို ပေါင်းစပ်အသုံးပြုထားပေမယ့် ထူးဆန်းတဲ့လုပ်ဆောင်ချက်တွေ မပါတော့ပါဘူး။ ဆက်လက်ပြီး components/Transactions.jsx မှာ အခုလို ရေးပေးပါ။

React Native - app/components/Transactions.jsx

```
import {
  Text,
  View,
  ScrollView,
  StyleSheet,
} from "react-native";

import { MaterialIcons } from "@expo/vector-icons";
import { useState } from "react";

export default function Transactions() {
  const [transactions, setTransactions] = useState([
```

```
    { id: 1, msg: "EPC Bill", type: "out", time: "Jul 6, 2024" },
    { id: 2, msg: "Deposit", type: "in", time: "Jul 5, 2024" },
    { id: 3, msg: "Transfer", type: "in", time: "Jul 5, 2024" },
    { id: 4, msg: "Wifi Bill", type: "out", time: "Jul 4, 2024" },
    { id: 5, msg: "Payment", type: "in", time: "Jul 4, 2024" },
  ]);

  return (
    <ScrollView>
      {transactions.map(transaction => {
        return (
          <View
            style={styles.transaction}
            key={transaction.id}>
            <MaterialIcons
              name="compare-arrows"
              size={32}
              color={
                transaction.type === "in" ? "green" : "brown"
              }
            />
            <View>
              <Text style={styles.text.payment}>
                {transaction.msg}
              </Text>
              <Text style={styles.text.muted}>
                {transaction.time}
              </Text>
            </View>
          </View>
        );
      })}
    </ScrollView>
  );
}
```

```
const styles = StyleSheet.create({
  transaction: {
    flexDirection: "row",
    gap: 20,
    alignItems: "center",
    padding: 15,
    backgroundColor: "#fff",
    borderRadius: 10,
    marginBottom: 10,
  },
  text: {
    payment: {
      fontSize: 21,
    },
    muted: {
      color: "#888",
    },
  },
});
```

ဒီနေရာမှာတော့ `useState` နဲ့ Default Data တချို့ ပေးထားပါတယ်။ အဲ့ဒီ State Data ကို `map()` လုပ်ပြီး `<View>` တွေကိုပဲ `List` တစ်ခုအနေနဲ့ ပြပေးထားတာပါ။ State အပြောင်းအလဲ လုပ်ဆောင်ချက်တွေ မပါလို့ `useState` တောင်သုံးမနေဘဲ ရိုးရိုး Variable နဲ့ရေးရင်လည်း ရနိုင်ပါတယ်။

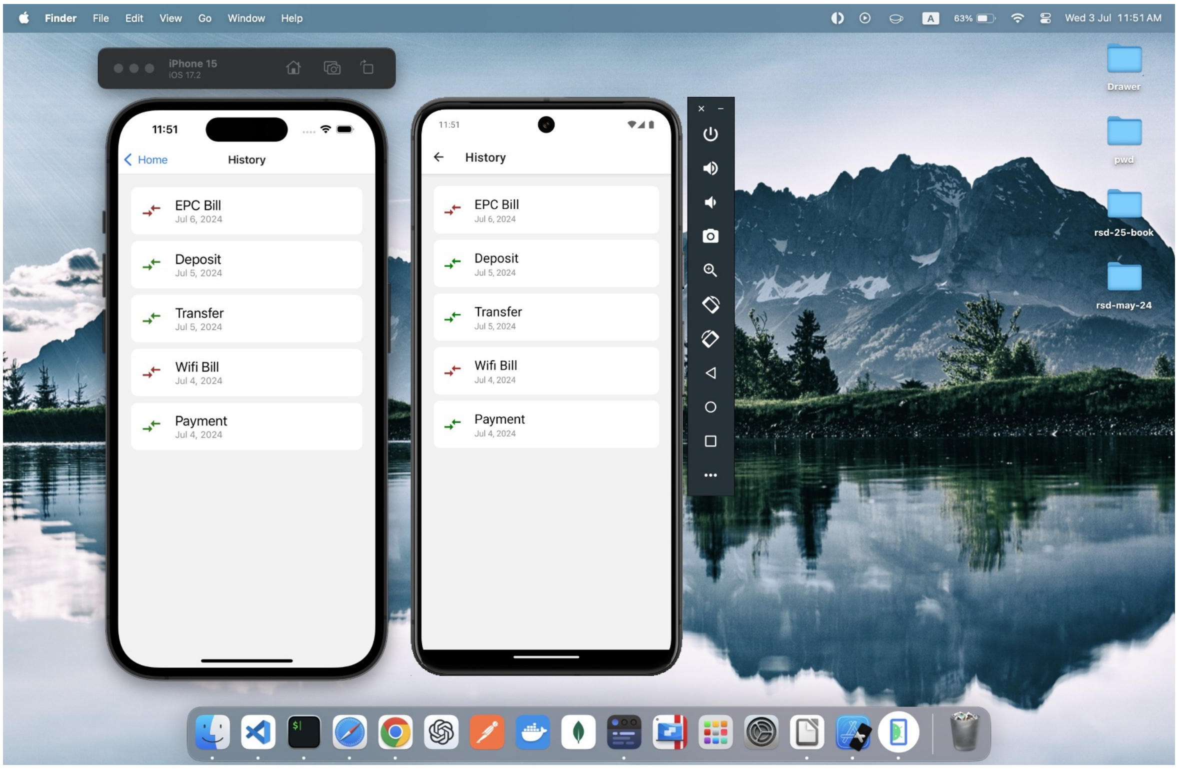
List တွေပြဖို့အတွက် Item တွေများရင် ပိုသင့်တော်တာကတော့ `FlatList` Component ဖြစ်ပါတယ်။ ပိုမြန်ပါလိမ့်မယ်။ ဒီနေရာမှာတော့ သိပ်မများတဲ့အတွက် ရိုးရိုး View တွေကိုပဲ List Item ပုံစံဖြစ်အောင် Style နဲ့လုပ်ပြီး ပြထားတာပါ။

လိုတဲ့နေရာကနေ ခေါ်သုံးဖို့ ရေးထားတဲ့ Component တွေတော့ စုံပါပြီ။ Page တွေရေးကြပါမယ်။ အရှင်ဆုံးဖြစ်တဲ့ `history.jsx` ကနေ စရေးကြပါမယ်။ ဒီလိုရေးပေးပါ။

React Native - app/history.jsx

```
import { View } from "react-native";

import Transactions from "./components/Transactions";

export default function History() {
  return <View style={{ padding: 20 }}>
    <Transactions />
  </View>
}
```

ဘာမှမပါပါဘူး။ ခုဏရေးရေးလိုက်တဲ့ Transaction Component ကိုပဲ ယူသုံးပြီး ပြလိုက်တာပါ။ အခု တော့ စမ်းဖို့မပြည့်စုံသေးပါဘူး။ အားလုံးစုံလို့ စမ်းကြည့်လိုက်ရင် ဒီ Page ရဲ့ ဖော်ပြပုံက အခုလိုဖြစ်မှာပါ။



ဆက်လက်ပြီး transfer.jsx မှာ အခုလိုရေးပေးပါ။

React Native - app/transfer.jsx

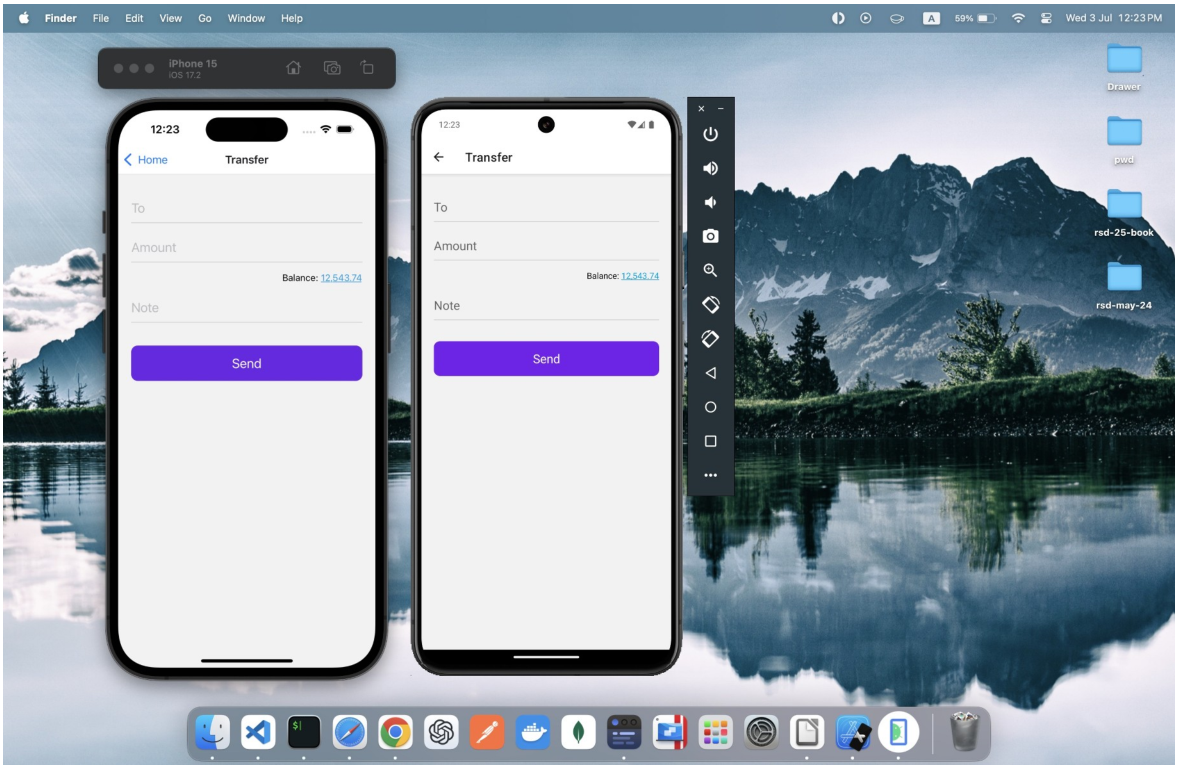
```
import {
  StyleSheet,
  View,
  Text,
  TextInput,
  TouchableOpacity,
} from "react-native";

import { useRef, useState } from "react";

export default function Transfer() {
  const toInput = useRef();
  const amountInput = useRef();
  const noteInput = useRef();

  const [available, setAvailable] = useState("12,543.74");
  const [to, setTo] = useState("");
  const [amount, setAmount] = useState(0);
  const [note, setNote] = useState("");

  return (
    <View style={styles.container}>
      <TextInput
        style={styles.input}
        placeholder="To"
        ref={toInput}
        value={to}
        onChangeText={setTo}
      />
      <TextInput
        style={styles.input}
        placeholder="Amount"
        ref={amountInput}
        value={amount}
```

```
        onChangeText={setAmount}
      />
      <TouchableOpacity
        style={styles.inputDes}
        onPress={() => {
          setAmount(available);
        }}>
        <Text>Balance: </Text>
        <Text style={styles.balance}>{available}</Text>
      </TouchableOpacity>

      <TextInput
        style={styles.input}
        placeholder="Note"
        ref={noteInput}
        value={note}
        onChangeText={setNote}
      />

      <TouchableOpacity style={styles.button}>
        <Text style={styles.buttonText}>Send</Text>
      </TouchableOpacity>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    gap: 15,
    padding: 20,
    marginTop: 10,
  },
  input: {
    flexGrow: 1,
    fontSize: 20,
```

```
    paddingTop: 10,
    paddingBottom: 10,
    borderBottomWidth: 1,
    borderBottomColor: "#ccc",
  },
  inputDes: {
    flexDirection: "row",
    justifyContent: "flex-end",
  },
  balance: {
    color: "#0e9ce2",
    textDecorationLine: "underline",
  },
  button: {
    marginTop: 20,
    alignItems: "center",
    padding: 15,
    backgroundColor: "#6d25e5",
    borderRadius: 10,
  },
  buttonText: {
    fontSize: 20,
    color: "white",
  },
});
```

Form Input တွေ Manage လုပ်ပုံ ပါသွားပါပြီ။ `<TextInput>` တွေကိုသုံးထားပါတယ်။ `Ref` တွေကိုယ်စီချိတ်ထားပါတယ်။ ပြယုံသက်သက်ပါပဲ။ ဘာအလုပ်မှတော့ လုပ်မထားပါဘူး။ ထူးခြားချက်အနေနဲ့ `<TextInput>` တွေရဲ့ `value` ကို သက်ဆိုင်ရာ State Value နဲ့ ချိတ်ပေးထားပါတယ်။ ပြီးတဲ့အခါ `onChangeText` မှာ `setState` Function တွေကို ပေးထားပါတယ်။ ဒါကြောင့် React Native က `TextInput` မှာ အပြောင်းအလဲရှိတိုင်း `setState` Function ကိုသုံးပြီး သက်ဆိုင်ရာ State Value ကို Update လုပ်ပေးမှာဖြစ်ပါတယ်။

လက်ရှိ Balance ကိုပြတဲ့ Text ကို `TouchableOpacity` နဲ့ နှိပ်လို့ရအောင်လည်း လုပ်ပေးထားပါတယ်။ နှိပ်လိုက်ရင် လက်ရှိ Balance ကို `amount` `TextInput` မှာ အလိုအလျှောက် ထည့်ပေးသွားမှာပါ။ Input မှာ ထည့်တာတော့ မဟုတ်ပါဘူး။ `amount` State မှာ ထည့်ပေးလိုက်လို့ သူနဲ့ချိတ်ထားတဲ့ Input Value ဖြစ်သွားတာပါ။ နောင်အားလုံးစုံလို့ စမ်းကြည့်ရင် အခုလိုပုံစံရမှာပါ။



ဆက်လက်ပြီး `scan.jsx` မှာ အခုလိုရေးပေးပါ။

React Native - app/scan.jsx

```
import {
  View,
  Text,
  Button,
  StyleSheet,
  TouchableOpacity
} from "react-native";
```

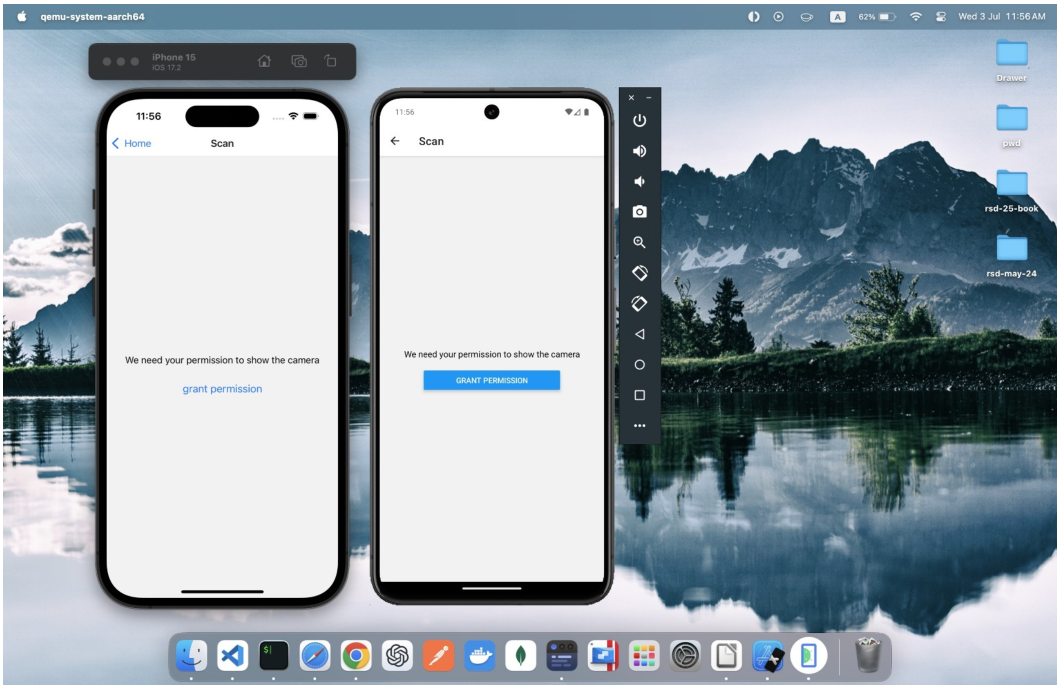
```
import { useState } from "react";
import { MaterialIcons } from "@expo/vector-icons";
import { CameraView, useCameraPermissions } from "expo-camera";

export default function Scan() {
  const [facing, setFacing] = useState("back");
  const [permission, requestPermission] = useCameraPermissions();

  if (!permission) {
    return <View />;
  }

  if (!permission.granted) {
    return (
      <View style={styles.container}>
        <Text style={{ textAlign: "center", fontSize: 16 }}>
          We need your permission to show the camera
        </Text>
        <View
          style={{ marginTop: 20, width: 250, alignSelf: "center" }}>
          <Button
            onPress={requestPermission}
            title="grant permission"
          />
        </View>
      </View>
    );
  }

  function toggleCameraFacing() {
    setFacing(current => (current === "back" ? "front" : "back"));
  }
```

```
  return (
    <View style={styles.container}>
      <CameraView
        style={styles.camera}
        facing={facing}

        onBarcodeScanned={result => {
          console.log(result.data);
        }}>
        <View style={styles.buttonContainer}>
          <TouchableOpacity
            style={styles.button}
            onPress={toggleCameraFacing}>
            <MaterialIcons
              name="cameraswitch"
              size={48}
              color="white"
            />
          </TouchableOpacity>
        </View>
      </CameraView>
    </View>
  );
}

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: "center",
  },
  camera: {
    flex: 1,
  },
  buttonContainer: {
    flex: 1,
```

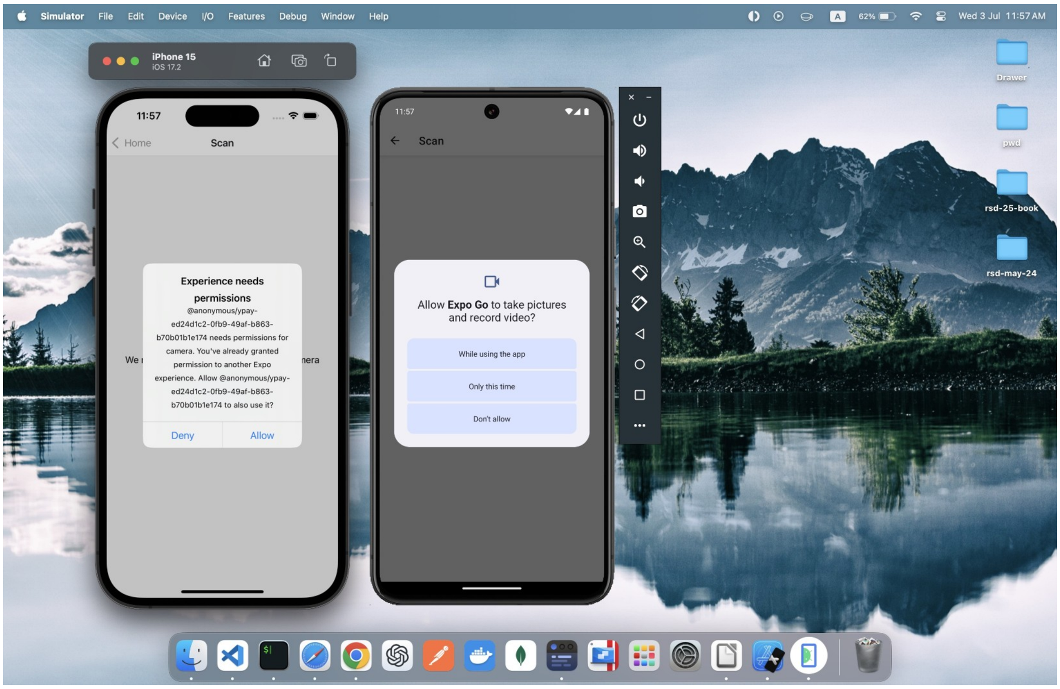
```
    flexDirection: "row",
    backgroundColor: "transparent",
    margin: 64,
  },
  button: {
    flex: 1,
    alignSelf: "flex-end",
    alignItems: "center",
  },
  text: {
    fontSize: 24,
    fontWeight: "bold",
    color: "white",
  },
});
```

ဒီကုဒ်ရဲ့ ထူးခြားချက်ကတော့ Expo Camera ကို အသုံးပြုထားခြင်းဖြစ်ပါတယ်။ `facing` နဲ့ `permission` ဆိုတဲ့ State Data နှစ်ခုလည်း ထည့်သုံးထားပါတယ်။ `facing` က ရှေ့ရိုးကင်မရာလား၊ Selfie ကင်မရာလား ဆိုတဲ့ State ကို သိမ်းဖို့ပါ။ Default အနေနဲ့ `back` ကိုပေးထားပါတယ်။ Permission ကတော့ User ဆီက Camera Permission ကိုတောင်းပြီး သိမ်းဖို့ ဖြစ်ပါတယ်။ `useState` မသုံးပါဘူး `useCameraPermissions` ကို သုံးပါတယ်။

`permission` မရှိရင် ဆက်အလုပ်မလုပ်အောင် `<View>` အလွတ်တစ်ခု ပြန်ပေးထားပါတယ်။ ပြီးတဲ့အခါ Permission တောင်းတဲ့ UI ကို ဆက်ရေးထားပါတယ်။ `useCameraPermissions` ကနေရတဲ့ `requestPermission` ကို နှိပ်လိုက်ရင် သက်ဆိုင်ရာ Mobile Device ရဲ့ Camera Permission တောင်းတဲ့လုပ်ဆောင်ချက် အလုပ်လုပ်သွားပါလိမ့်မယ်။ နောင်အားလုံးစုံလို့ စမ်းကြည့်တဲ့အခါ ရလဒ်က အခုလိုပုံစံ ဖြစ်မှာပါ။



`grant permission` ခလုပ်ကိုနှိပ်လိုက်ရင်တော့ အခုလိုတွေ့ရမှာပါ။



Expo မှာ အဲ့ဒီလိုမျိုး Device Feature တွေကို စီမံပေးနိုင်တဲ့ လုပ်ဆောင်ချက်တွေအစုံပါဝင်ပါတယ်။ ရေးထားတဲ့ကုဒ်ကို ပြန်လေ့လာကြည့်ပါ။ `<CameraView>` Component နဲ့ ကင်မရာရဲ့ လုပ်ဆောင်ချက်ကို ပြထားပါတယ်။ သူ့ထဲမှာ ကင်မရာရှေ့နောက်ပြောင်းလို့ရတဲ့ ခလုပ်တစ်ခုထည့်ထားပါတယ်။ `facing` ကို `font` နဲ့ `back` လိုသလို ပြောင်းပေးလိုက်တာပါပဲ။ အဲ့ဒီ `facing` Value ကို `<CameraView>` ရဲ့ Prop မှာ ထည့်ပေးထားပါတယ်။

အဓိကအကျဆုံးကတော့ `onBarcodeScanned` လုပ်ဆောင်ချက်ဖြစ်ပါတယ်။ ဒါကြောင့် Scan Page ကိုဖွင့်လိုက်ရင် `CameraView` ပေါ်လာပါလိမ့်မယ်။ အဲ့ဒီ Camera နဲ့ QR Code အပါအဝင် Barcode တွေကို ဖတ်လိုက်တဲ့အခါ `onBarcodeScanned` အလိုအလျှောက် အလုပ်လုပ်သွားပါလိမ့်မယ်။ နမူနာအနေနဲ့ `data` ကို Console မှာ ရိုက်ထုတ်ပြခိုင်းထားပါတယ်။

`<CameraView>` ရဲ့ ပြဿနာက Emulator တွေ Simulator တွေနဲ့ စမ်းလို့ အဆင်မပြေတာပါပဲ။ ဒါက `CameraView` ကြောင့် မဟုတ်ပါဘူး။ Emulator တွေ Simulator တွေမှာကိုက မရတာပါ။ Emulator မှာ ရအောင် လုပ်ရင်ရနိုင်ပေမယ့် တော်တော်လေးပြီး သုံးရခက်ပါတယ်။ ဒါကြောင့် ဒီလုပ်ဆောင်ချက်ကို ဖုန်းမှာစမ်းကြည့်မှပဲ အဆင်ပြေမှာဖြစ်ပါတယ်။ အခုတော့ စမ်းလို့မရသေးပါဘူး။ အားလုံးစုံမှ စမ်းကြည့်လိုက်ပါ။

Expo Camera ရဲ့လုပ်ဆောင်ချက် ပြည့်စုံအောင် Setting လည်းပြင်ပေးရပါမယ်။ `app.json` ဖိုင်ရဲ့ `plugins` Section ကို ဒီလိုပြင်ပေးပါ။

JSON - app.json

```json
"plugins": [
  "expo-router",
  [
    "expo-camera",
    {
      "cameraPermission": "Allow $(PRODUCT_NAME) to access your camera"
    }
  ]
],
```

expo-router ကတော့ မူလကတည်းက ပါပြီး ဖြစ်ပါလိမ့်မယ်။ expo-camera ထပ်ထည့်လိုက်တာပါ။ လိုအပ်ရင် Camera, Video, Recording Permission တွေတောင်းလို့ရပါတယ်။ ဒီနေရာမှာတော့ မလိုအပ်လို့ Camera Permission တစ်မျိုးပဲ တောင်းထားပါတယ်။

နောက်တစ်ဆင့်အနေနဲ့ fxrate.jsx မှာ အခုလို ရေးပေးပါ။

React Native - app/fxrate.jsx

```
import { View, TextInput, Text, StyleSheet } from "react-native";
import { useRef, useState } from "react";
import { useQuery } from "react-query";

import { MaterialIcons } from "@expo/vector-icons";

const api = "https://api.frankfurter.app/latest?from=USD";

export default function FxRate() {
  const amountInput = useRef();
  const [amount, setAmount] = useState(1);

  const { isLoading, isError, error, data } = useQuery("fxrate", async () => {
    const res = await fetch(api);
    return res.json();
  });

  if (isError) {
    return (
      <View style={{ justifyContent: "center", alignItems: "center" }}>
        <Text>{error.message}</Text>
      </View>
    );
  }
```

```
if (isLoading) {
  return (
    <View style={{ justifyContent: "center", alignItems: "center" }}>
      <Text>Loading...</Text>
    </View>
  );
}

  const convert = code => {
      const usd = amount > 0 ? amount : 1;
      if(!data.rates[code]) return 0;

      return usd * data.rates[code];
  }

return (
  <View style={{ padding: 20 }}>
    <View style={styles.item}>
      <MaterialIcons
        name="attach-money"
        color="#0e9ce2"
        size={32}
      />
      <TextInput
        style={styles.input}
        onChangeText={setAmount}
        value={amount}
        ref={amountInput}
        placeholder="1 USD"
      />
    </View>
    <View style={styles.item}>
      <MaterialIcons
        name="euro"
        color="#0e9ce2"
```

```
          size={24}
        />
        <Text style={styles.result}>{convert("EUR")} EUR</Text>
      </View>
      <View style={styles.item}>
        <MaterialIcons
          name="currency-pound"
          color="#0e9ce2"
          size={24}
        />
        <Text style={styles.result}>{convert("GBP")} GBP</Text>
      </View>
      <View style={styles.item}>
        <MaterialIcons
          name="currency-yen"
          color="#0e9ce2"
          size={24}
        />
        <Text style={styles.result}>{convert("JPY")} JPY</Text>
      </View>
      <View style={styles.item}>
        <MaterialIcons
          name="currency-yuan"
          color="#0e9ce2"
          size={24}
        />
        <Text style={styles.result}>{convert("CNY")} CNY</Text>
      </View>
      <View style={styles.item}>
        <MaterialIcons
          name="currency-rupee"
          color="#0e9ce2"
          size={24}
        />
        <Text style={styles.result}>{convert("INR")} INR</Text>
```

```
      </View>
      <View style={styles.item}>
        <MaterialIcons
          name="currency-exchange"
          color="#0e9ce2"
          size={24}
        />
        <Text style={styles.result}>{convert("CAD")} CAD</Text>
      </View>
      <View style={styles.item}>
        <MaterialIcons
          name="currency-exchange"
          color="#0e9ce2"
          size={24}
        />
        <Text style={styles.result}>{convert("AUD")} AUD</Text>
      </View>
      <View style={styles.item}>
        <MaterialIcons
          name="currency-exchange"
          color="#0e9ce2"
          size={24}
        />
        <Text style={styles.result}>{convert("SGD")} SGD</Text>
      </View>
      <View style={styles.item}>
        <MaterialIcons
          name="currency-exchange"
          color="#0e9ce2"
          size={24}
        />
        <Text style={styles.result}>{convert("THB")} THB</Text>
      </View>
    </View>
  );
```

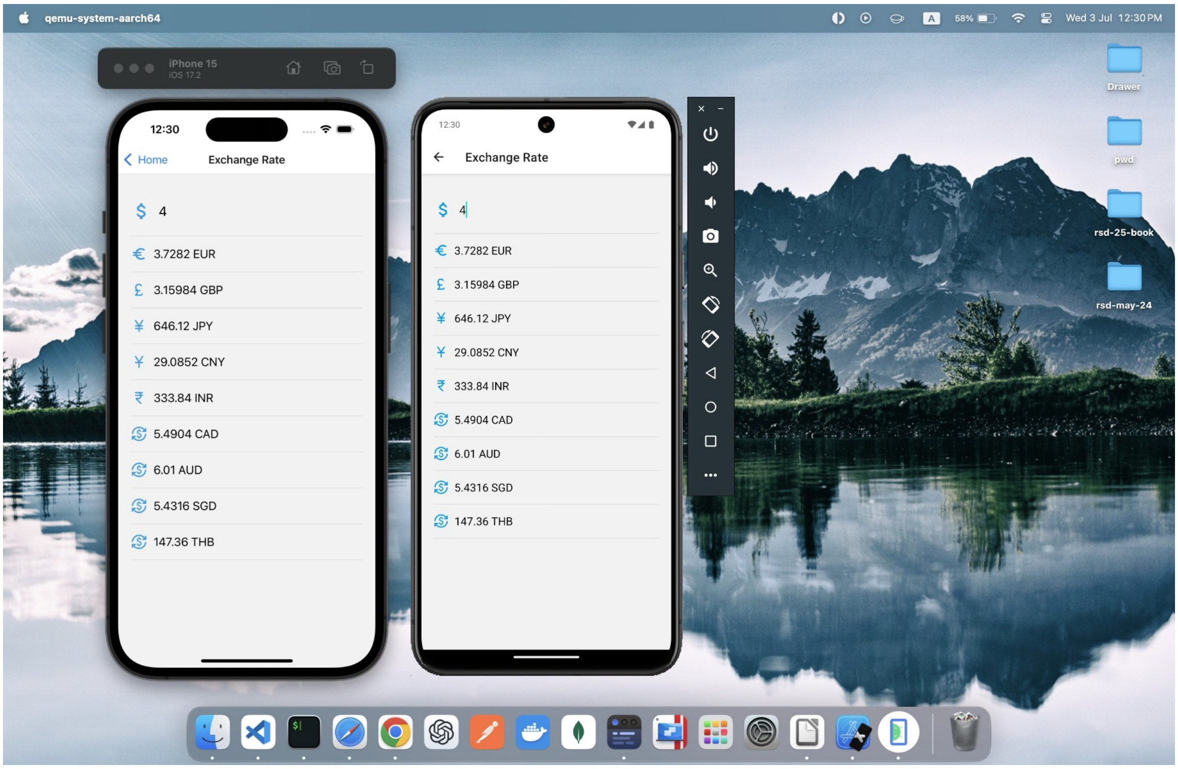
```
}

const styles = StyleSheet.create({
  input: {
    flexGrow: 1,
    fontSize: 20,
    paddingTop: 10,
    paddingBottom: 10,
  },
    item: {
        flexDirection: "row",
        gap: 10,
        alignItems: "center",
        paddingTop: 15,
        paddingBottom: 15,
        borderBottomWidth: 1,
        borderBottomColor: "#ddd",
    },
    result: {
        fontSize: 18,
    }
});
```

ပါတဲ့ကုဒ်များပေမယ့် လိုရင်းကတော့ React Query နဲ့ Exchange Rate API ကိုလှမ်းခေါ်ပြီး ပြန်ရတဲ့ Value တွေကို List အနေနဲ့ ပြထားတာပါ။ နမူနာသုံးပြထားတဲ့ API ဟာ ဥရောပဗဟိုဘဏ်က ထုတ်ပြန်တဲ့ ငွေလဲနှုန်းတွေကို ပြန်ပေးတဲ့ Free API ဖြစ်ပါတယ်။ သူပြန်ပေးတဲ့ Response Data ရဲ့ ဖွဲ့စည်းပုံက ဒီလို ဖြစ်ပါလိမ့်မယ်။

```
{
  "amount": 1.0,
  "base": "USD",
  "date": "2024-07-02",
  "rates": {
    "AUD": 1.5025,
    "BGN": 1.8229,
    "BRL": 5.637,
    "CAD": 1.3726,
    "CHF": 0.90381,
    "CNY": 7.2713,
    "CZK": 23.474,
    "DKK": 6.9522,
    "EUR": 0.93205,
    "GBP": 0.78996,
    "HKD": 7.8133,
    ...
  }
}
```

1 Dollar ကို အခြေခံတဲ့ ငွေလဲနှုန်းကိုပြန်ပေးတာပါ။

`TextInput` တစ်ခုလည်း ထည့်ထားပါတယ်။ `convert()` Function တစ်ခုလည်း ရေးထားပါတယ်။ `TextInput` မှာ ရိုက်ထည့်ထားတဲ့တန်ဖိုးနဲ့ API ကရတဲ့ ငွေလဲနှုန်းကို မြှောက်ပြီး ပြန်ပေးပါတယ်။ ဒါကြောင့် `TextInput` မှာ ဒေါ်လာ Amount ကိုရိုက်ထည့်လိုက်ရင် အဲ့ဒီ Amount နဲ့ကိုက်ညီတဲ့ ငွေလဲနှုန်းတွေကို ရရှိမှာ ဖြစ်ပါတယ်။ ရလဒ်က အခုလိုပုံစံဖြစ်ပါလိမ့်မယ်။



အခုဆိုရင်လိုအပ်တဲ့ Page တွေစုံသွားပါပြီ။ နောက်ဆုံးတစ်ခုအနေနဲ့ `index.jsx` မှာ အခုလိုရေးပေးပါ။

React Native - app/index.jsx

```
import {
  Text,
  View,
  StyleSheet,
  SafeAreaView,
  Platform,
} from "react-native";

import { LinearGradient } from "expo-linear-gradient";

import Transactions from "./components/Transactions";
import ActionButton from "./components/ActionButton";
import BalanceCard from "./components/BalanceCard";
```

```
export default function Home() {
  return (
    <SafeAreaView style={styles.container}>
      <LinearGradient
        colors={["rgba(0,0,0,0.8)", "transparent"]}
        style={styles.background}
      />
      <View>
        <BalanceCard />
        <View style={styles.actions}>
          <ActionButton
            color="#ff009d"
            icon="compare-arrows"
            label="Transfer"
            path="/transfer"
          />
          <ActionButton
            color="#0e9ce2"
            icon="qr-code-2"
            label="Scan"
            path="/scan"
          />
          <ActionButton
            color="#7b48f4"
            icon="attach-money"
            label="Fx Rate"
            path="/fxrate"
          />
          <ActionButton
            color="#ff379e"
            icon="history"
            label="History"
            path="/history"
          />
```

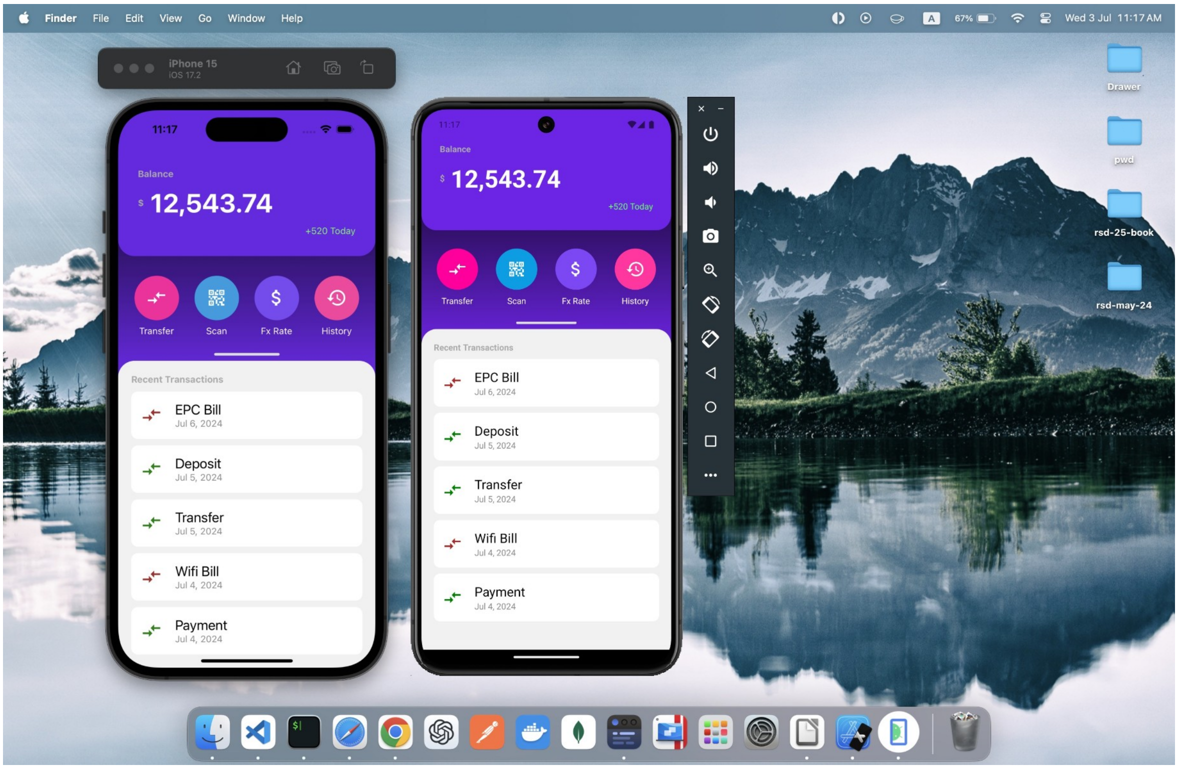
```
        </View>

        <View style={styles.moreActionBar}></View>
        <View style={styles.transactions}>
          <Text style={styles.text.label}>Recent Transactions</Text>
          <Transactions />
        </View>
      </View>
    </SafeAreaView>
  );
}

const styles = StyleSheet.create({
  background: {
    position: "absolute",
    left: 0,
    right: 0,
    top: 100,
    height: 300,
  },
  container: {
    paddingTop: Platform.OS === "android" ? 25 : 0,
    backgroundColor: "#6d25e5",
  },
  actions: {
    paddingTop: 30,
    paddingLeft: 25,
    paddingRight: 25,
    paddingBottom: 25,
    flexDirection: "row",
    justifyContent: "space-between",
  },
  moreActionBar: {
    width: 100,
    height: 4,
```

```
    backgroundColor: "#ddd",
    borderRadius: 50,
    alignSelf: "center",
    marginBottom: 8,
  },
  transactions: {
    padding: 20,
    gap: 10,
    borderTopLeftRadius: 25,
    borderTopRightRadius: 25,
    backgroundColor: "#f1f1f1",
  },
  text: {
    label: {
      fontWeight: "bold",
      color: "#aaa",
    },
  },
});
```

ထူးခြားချက်တွေ ရွေးကြည့်ကြပါမယ်။ `SafeAreaView` Component ကိုထည့်သုံးထားပါတယ်။ ကိုယ့် App ရဲ့ Content တွေကို Notch တွေ Dynamic Island တွေနဲ့လွတ်ကင်းအောင် ပြစေချင်ရင် `SafeAreaView` နဲ့ ပြရပါတယ်။ အသုံးဝင်ပေမယ့် ဒီ Component က iOS အတွက်ပဲ အလုပ်လုပ်ပါတယ်။ Android မှာအလုပ်မလုပ်ပါဘူး။ ဒါကြောင့် Platform ကိုသုံးပြီး Style မှာ စစ်ထားတာကို သတိပြုပါ။ `Platform.OS` က Android ဖြစ်မယ်ဆိုရင် paddingTop ကိုပိုပေးထားပါတယ်။ Android Device တွေမှာ ပါလေ့ရှိတဲ့ Camera Notch တွေ Punch Hole တွေနဲ့ လွတ်အောင် ပြပေးစေချင်လို့ ဖြစ်ပါတယ်။

`<LinearGradient>` ကိုသုံးပြီး Gradient Background Color ရအောင် လုပ်ထားပါတယ်။ မူလ Background ပေါ်မှာ Transparent Layer တစ်ထပ်ထပ်အုပ်ပြီး Gradient Color ပုံစံရအောင် ထည့်ထားတာပါ။ `styles.background` မှာ `position: absolute` နဲ့ `top: 100` လောက်နေ စပြီခိုင်းထားတာကို သတိပြုပါ။ `top: 0` ကနေပြလည်းရပါတယ်။ ကိုယ့်ဘာသာပြောင်းပြီး စမ်းကြည့်လို့ရပါ

တယ်။ `top: 0` ကနေ စပြရင် Status Bar ပေါ်က စာလုံးတွေနဲ့ အရောင်ရောပြီး အဆင်မပြေလို့ `top: 100` ကနေ စပြခိုင်းထားတာပါ။ Status Bar ကို Expo ရဲ့ `<Statusbar>` Component နဲ့ Manage လုပ်လို့ရပေမယ့် ထည့်သုံးပြမထားပါဘူး။ လေ့လာပြီးထည့်သုံးကြည့်လိုက်ပါ။ သုံးရလွယ်ပါတယ်။ ကျန်တာတွေကတော့ ကုဒ်များပေမယ့် ထူးခြားချက်သိပ်မရှိဘဲ စောစောက ကြိုရေးထားတဲ့ Component တွေကို လိုအပ်သလို ခေါ်သုံးလိုက်တာပါပဲ။ ဒီထိရေးပြီးရင်တော့ အားလုံးစုံသွားပါပြီ။ စမ်းကြည့်လို့ ရပါပြီ။



Emulator တွေ Simulator တွေထဲမှာ စမ်းကြည့်သင့်သလို၊ ကိုယ့်ဖုန်းထဲမှာလည်း Expo Go နဲ့ စမ်းကြည့်သင့်ပါတယ်။ Scan လုပ်ဆောင်ချက်မျိုးဆိုရင် ဖုန်းနဲ့စမ်းမှ အဆင်ပြေမှာမို့လို့ပါ။

မူလအစီအစဉ်ကတော့ ရှေ့မှာ ရိုးရိုး React နဲ့ ရေးခဲ့တဲ့ **yaycha** App ကို React Native နဲ့ ပြန်ရေးပြမလို့ပါ။ နှစ်ခါပြန်လုပ်ရင် ပျင်းစရာဖြစ်နေမှာစိုးလို့ အခုလို ပုံစံမတူတဲ့ Pay App UI တစ်ခု လုပ်ပြလိုက်တာပါ။ ကိုယ့်ဘာသာ React Native နဲ့ **yaycha** App ပုံစံ စမ်းလုပ်ကြည့်ဖို့လည်း တိုက်တွန်းပါတယ်။ အလေ့အကျင့် ရသွားမှာဖြစ်သလို လိုအပ်ချက်ရှိလာတဲ့အခါ ဖြည့်စွက်လေ့လာဖြစ်သွားတာတွေလည်း ရှိပါလိမ့်မယ်။

အခန်း (၁၅) - API Authentication - Login, Register

ဒီအခန်းမှာ လုပ်လက်စဖြစ်တဲ့ **Yaycha** App အတွက် Authentication လုပ်ဆောင်ချက် ထည့်ကြမှာ ဖြစ်ပါတယ်။ API ဘက်မှာလည်း လိုအပ်တဲ့လုပ်ဆောင်ချက်တွေ ထပ်ဖြည့်ရမှာဖြစ်သလို၊ Client App ဘက်မှာလည်း လိုအပ်တဲ့ UI တွေ ထပ်ဖြည့်ကြရမှာပါ။

ပြည့်စုံတဲ့ Authentication စနစ်တစ်ခုရဖို့ ကိုယ်တိုင်အစအဆုံးလုပ်ရတာ မလွယ်ပါဘူး။ မလွယ်ဘူးဆိုတာ နည်းပညာပိုင်း ခက်တာကိုချည်းပဲ ပြောတာမဟုတ်ပါဘူး။ လုပ်ရတဲ့အလုပ်တွေ များလွန်းလို့ ပရောဂျက်ရဲ့ အချိန်တော်တော်များများကို သူ့မှာအချိန်ပေးရလို့ပါ။

Username နဲ့ Password ကိုသုံးပြီး Login ဝင်ခိုင်းမယ်ဆိုယုံလောက်တဲ့ မပြီးပါဘူး။ Email သုံးရင် Email Activation ကိစ္စတွေလုပ်ရပါမယ်။ ဖုန်းနံပါတ်သုံးရင် Phone Activation ကိစ္စတွေ လုပ်ရပါမယ်။ Password မေ့သွားရင် ပြန်ယူလို့ရအောင် Account Recovery အတွက် လိုအပ်တာတွေလုပ်ရပါမယ်။ အဲ့ဒီ Account Recovery ကို Abuse လုပ်လို့မရအောင် လုပ်ရပါမယ်။ Social Media အကောင့်တွေနဲ့ Login ဝင်လို့ရတာမျိုးလည်း လိုနိုင်ပါတယ်။ ပိုပြည့်စုံချင်ရင် SMS OTP သို့မဟုတ် အခြား Two Factors Authentication စနစ်တစ်ခုခုကို သုံးပေးရပါမယ်။ ဒီထက်ပိုလုံခြုံချင်ရင်တော့ Fingerprint လို Biometric ကိစ္စတွေပါလုပ်ရပါမယ်။ တစ်ကယ့်ကို မလွယ်တဲ့ကိစ္စကြီးပါ။

ပြီးတော့ တစ်ခါလည်းမဟုတ် နှစ်ခါလည်းမဟုတ်၊ ပရောဂျက်တိုင်းမှာ လိုအပ်တဲ့ကိစ္စဖြစ်လို့ ဒီလောက်များတဲ့ အလုပ်တွေကို ပရောဂျက်တစ်ခါစတိုင်း တစ်ခါပြန်လုပ်ရရင်လည်း မဟုတ်သေးပါဘူး။ <u>Authentication ကို ဖြေရှင်းပြီးစီးပြီးသားကိစ္စ (Solved Problem) လို့ပြောကြပါတယ်</u>။ ဖြေရှင်းပြီးသားကို ထပ်ဖြေရှင်းနေ

မယ့်အစား ရှိပြီးသားကို အသုံးချလိုက်ပါလို့ ဆိုကြပါတယ်။ ဒါကြောင့် ကိုယ့်ဘာသာအစအဆုံး ရေးတာကို သိပ်အားမပေးကြဘဲ Auth0 တို့ Clerk တို့လို လိုအပ်တဲ့ လုပ်ဆောင်ချက်တွေ ပြည့်စုံအောင် ပါဝင်တဲ့ Third Party နည်းပညာတွေကို အသုံးပြုဖို့ တိုက်တွန်းကြပါတယ်။

ဒီအခန်းမှာ Authentication အကြောင်းလုပ်ကြတဲ့အခါ အဲ့ဒီလောက် ပြည့်စုံအောင် လုပ်နိုင်မှာမဟုတ်ပါဘူး။ Username, Password နဲ့ Login ဝင်လို့ရတဲ့စနစ်လောက်ပဲ လုပ်နိုင်မှာပါ။ လက်တွေ့အသုံးပြုဖို့ထက် Practice ရစေဖို့၊ လုပ်ရင်းနဲ့ လေ့လာသင့်တာတွေ ဖြည့်စွက်လေ့လာသွားဖို့ကိုသာ ရည်ရွယ်ပါတယ်။ စလိုက်ကြရအောင်။

API Authentication

ပထမဆုံးအနေနဲ့ API Authentication အတွက် Token Based Authentication စနစ်ကို အသုံးပြုကြပါမယ်။ သူ့ရဲ့အလုပ်လုပ်ပုံက ဒီလိုပါ။

1. Client က ပထမဆုံးတစ်ကြိမ် Username နဲ့ Password ကို မှန်အောင်ပေးရပါမယ်။
2. မှန်ရင် API က Access Token ကို ပြန်ပေးပါမယ်။
3. Client က Access Token ကို သိမ်းထားရပါမယ်။
4. API က တချို့လုပ်ဆောင်ချက်တွေကို Access Token ပါမှ ခွင့်ပြုမယ်လို့ ကန့်သတ်ပါမယ်။
5. Client က ကန့်သတ်ထားတဲ့ လုပ်ဆောင်ချက်တွေကို လိုချင်ရင် Access Token ပေးရပါမယ်။

ဒီနည်းနဲ့ အလုပ်လုပ်တဲ့ စနစ်ပါ။ Access Token အဖြစ် Random String Token တွေ သုံးမယ်ဆိုရင် ရနိုင်ပါတယ်။ သုံးလည်းသုံးကြပါတယ်။ ဒီနေရာမှာတော့ JSON Web Token (JWT) လို့ခေါ်တဲ့ နည်းပညာကို သုံးပါတယ်။ JWT ကိုသုံးပြီး Access Token တွေ ဖန်တီးလို့ရပါတယ်။ ထူးခြားချက်အနေနဲ့ JWT က ပေးလိုက်တဲ့ Data ပေါ်အခြေခံပြီး Token ကိုဖန်တီးမှာပါ။ Random မဟုတ်ပါဘူး။ နောင်လိုအပ်ရင် Token ကို Decode လုပ်ပြီး မူလ Data ကို ပြန်ထုတ်ယူလို့ရနိုင်ပါတယ်။

`yaycha-api` ပရောဂျက်ထဲမှာ အခုလို JWT ကို အခုလို Install လုပ်ပေးပါ။

```
npm i jsonwebtoken
```

သူ့ရဲ့အလုပ်လုပ်ပုံက ဒီလိုပါ။

JavaScript

```
const jwt = require("jsonwebtoken");

const secret = "s!c#1$G9";
const data = { name: 'Alice', bio: 'jwt bio' };

const token = jwt.sign(data, secret);

console.log(token);
```

JWT နဲ့ data ကို `sign()` လုပ်ပြီး token ထုတ်ယူထားတာပါ။ အဲ့ဒီလို `sign` လုပ်တဲ့အခါ Secret Key တစ်ခု ပေးရပါတယ်။ အဲ့ဒီ Secret Key နဲ့ Decode ပြန်လုပ်လို့ရတဲ့အတွက် Strong ဖြစ်တဲ့ Key ဖြစ်ဖို့လို ပြီး၊ လုံခြုံအောင်သိမ်းထားဖို့လည်း လိုပါတယ်။ ဒီကုဒ်ကို Run ကြည့်လိုက်ရင် JWT Token ကို အခုလို Output အနေနဲ့ ရရှိမှာဖြစ်ပါတယ်။

Output

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1lIjoiQWxpY2UiLCJiaW8iOiJqd
3QgYmlvIiwiaWF0IjoxNzIwMTAxNjgxfQ.h09_x1CbU4xT1Bj2OJPB-
PY_TEJpQHF5uXUAnLfM49A
```

Token မှန်မမှန် ပြန်စစ်ရင် Token ထဲက Data ကို အခုလို ပြန်ထုတ်ယူလို့ရပါတယ်။

JavaScript

```
const jwt = require("jsonwebtoken");

const secret = "s!c#1$G9";
// const data = { name: 'Alice', bio: 'jwt bio' };
// const token = jwt.sign(data, secret);
```

```
const token =
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1lIjoiQWxpY2UiLCJiaW8iOiJq
d3QgYmlvIiwiaWF0IjoxNzIwMTAxNjgxfQ.h09_x1CbU4xT1Bj2OJPB-
PY_TEJpQHF5uXUAnLfM49A";

const decode = jwt.verify(token, secret);
console.log(decode);
```

`verify()` နဲ့ Token ကိုပေးပြီး Decode လုပ်ထားတာပါ။ Decode လုပ်ချိန်မှာ ထည့်ပေးတဲ့ Secret Key လည်းမှန်ဖို့လိုပါတယ်။ ဒီကုဒ်ကို Run ကြည့်ရင် ရလဒ်က အခုလိုဖြစ်မှာပါ။

Output

```
{ name: 'Alice', bio: 'jwt bio', iat: 1720101681 }
```

မူလ Data ကို ပြန်ရမှာဖြစ်ပြီး နောက်ဆုံးက Token Created လုပ်ခဲ့ချိန် Timestamp လည်း ပါလာတာကို တွေ့ရမှာပါ။ Token Create လုပ်ချိန်မှာ Expire Time ထည့်ပေးမယ်ဆိုရင်လည်း ရနိုင်ပါတယ်။

ဒီနည်းပညာကို သုံးထားတဲ့ Login ကုဒ်ကို လုပ်လက်စ **`yaycha-api`** ရဲ့ `routers/user.js` မှာ အခုလိုရေးပေးပါ။

JavaScript - API - routers/user.js

```
router.post("/login", async (req, res) => {
  const { username, password } = req.body;

  if (!username || !password) {
    return res
      .status(400)
      .json({ msg: "username and password required" });
  }
```

```
  const user = await prisma.user.findUnique({
    where: { username },
  });

  if (user) {
    if (bcrypt.compare(password, user.password)) {
      const token = jwt.sign(user, process.env.JWT_SECRET);
      return res.json({ token, user });
    }
  }

  res.status(401).json({ msg: "incorrect username or password" });
});
```

POST ကို သုံးထားတာ သတိပြုပါ။ Request Body ကနေ username နဲ့ password ကို ယူထားပါတယ်။ မပါရင် 400 Bad Request ကုဒ်ကိုပြန်ပေးထားပါတယ်။ ပြီးတဲ့အခါ username နဲ့ Data ကို ထုတ်ယူပါတယ်။ ရှေ့ပိုင်းက နမူနာတွေမှာလို `findFirst()` ကို သုံးရင်လည်း ရပါတယ်။ နမူနာမှာ `findUnique()` ကိုသုံးထားပါတယ်။ သုံးနည်းက အတူတူပါပဲ။

User Data တွေထည့်တဲ့အခါ `password` တွေကို `bcrypt` နဲ့ Hash လုပ်ပြီးသိမ်းခဲ့တာ မှတ်မိပါလိမ့်မယ်။ ဒါကြောင့် နောက်တစ်ဆင့်မှာ `bcrypt.compare()` နဲ့ ပေးလာတဲ့ `password` ကို သိမ်းထားတဲ့ Hash နဲ့တူမတူ တိုက်စစ်ပါတယ်။ User လည်းရှိတယ်၊ `password` လည်းမှန်တယ်ဆိုရင် Login Success ဖြစ်သွားတာပါပဲ။ ဒါကြောင့် JWT နဲ့ User Data ကိုပေးပြီး Token ထုတ်ယူပါတယ်။ Secret Key ကိုတော့ ကုဒ်ထဲမှာ ထည့်မရေးပါဘူး။ `.env` ဖိုင်ထဲမှာ အခုလို ထည့်ထားပါတယ်။

.env

```
DATABASE_URL="file:./dev.db"
JWT_SECRET="F93U56G40jWrShL"
```

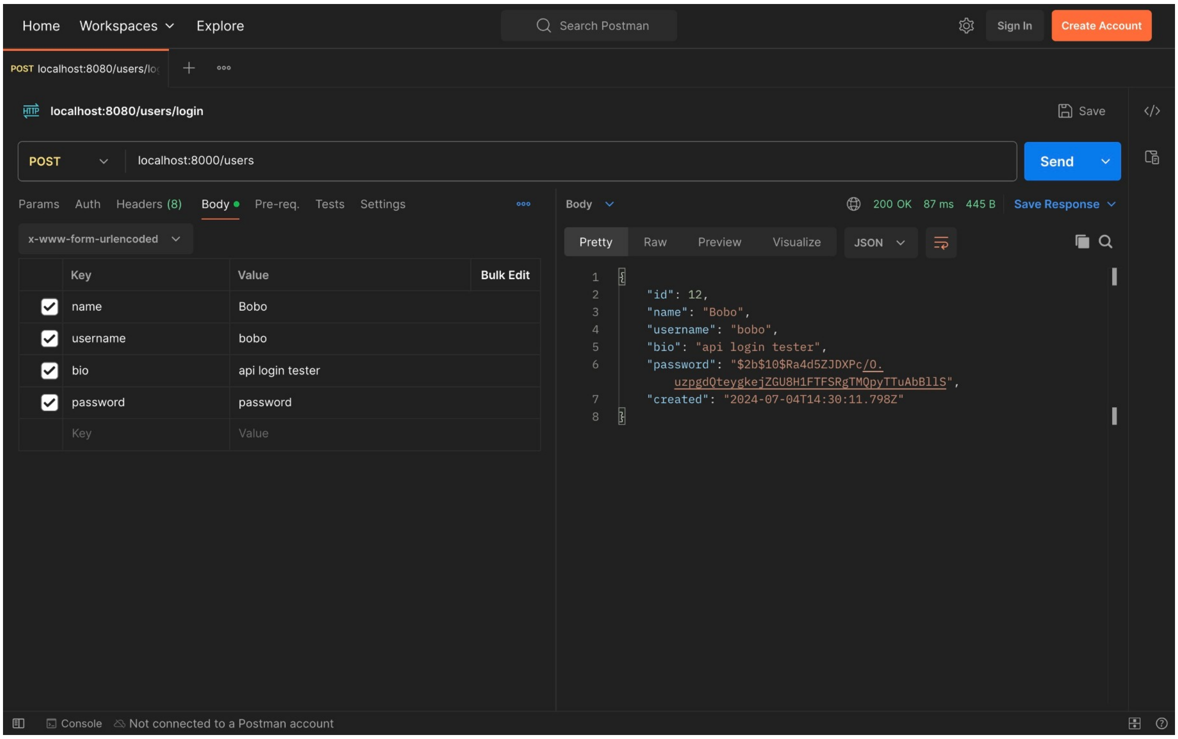
`DATABASE_URL` က မူလကတည်းက ရှိနေတာဖြစ်ပြီး `JWT_SECRET` ထပ်တိုးပေးလိုက်တာပါ။ အခုလို

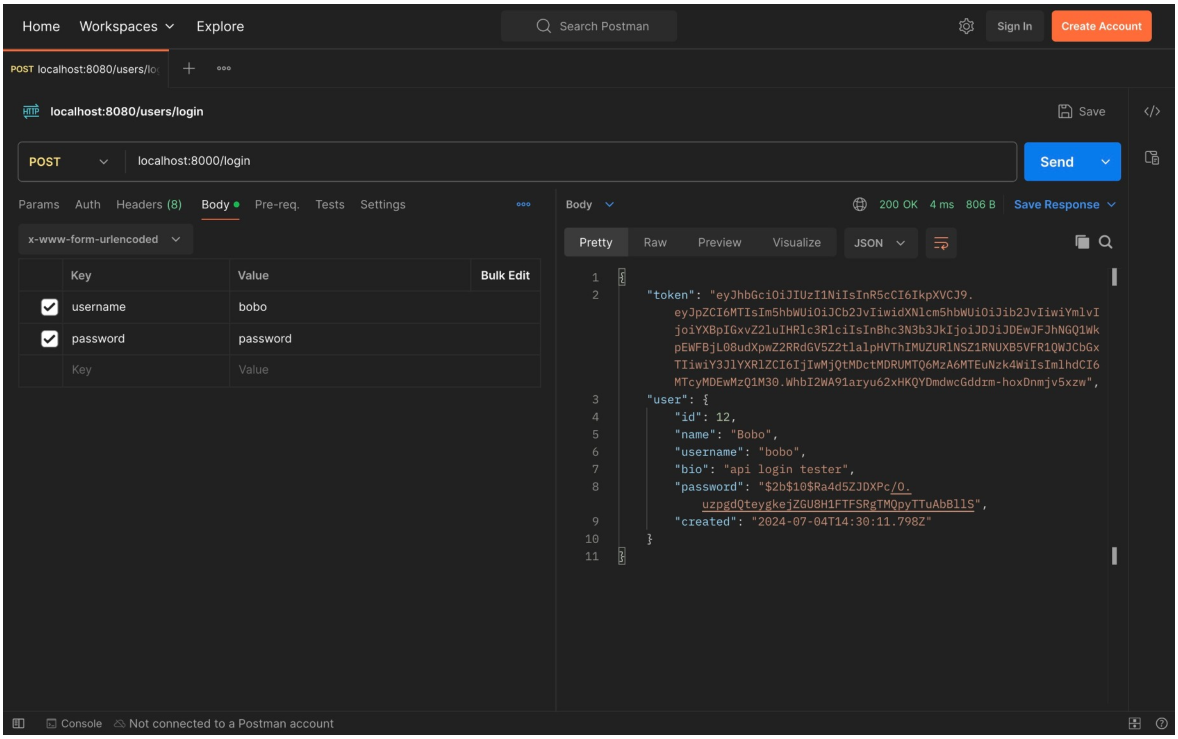
`.env` ဖိုင်က Variable တွေကို ချိတ်သုံးချင်ရင် `dotenv` လို့ခေါ်တဲ့ နည်းပညာကို သုံးရပါတယ်။ ဒါကြောင့် အခုလို Install လုပ်ပေးဖို့ လိုပါတယ်။

```
npm i dotenv
```

Install လုပ်ပြီးရင်ရပါပြီ။ `process.env` ကနေတစ်ဆင့် `.env` ထဲက Variable တွေကို ယူသုံးလို့ ရသွားပါပြီ။ Sign လုပ်ပြီးလို့ Token ရလာတဲ့အခါ Token ရော User Data ရော နှစ်ခုလုံးကို Response Data အဖြစ်ပြန်ပေးထားပါတယ်။ အကယ်၍ Username, Password မမှန်လို့ အပေါ်ကအလုပ်တွေ မလုပ်ဖြစ်ခဲ့ရင်တော့ 401 Unauthorized Code ကို ပြန်ပို့ထားပါတယ်။

Postman နဲ့ စမ်းကြည့်လို့ရပါပြီ။ Seed လုပ်တုံးက နမူနာထည့်ထားတဲ့ User Data တွေကိုသုံးလို့လည်း ရပါတယ်။ User အသစ်ထည့်ပေးနိုင်တဲ့ API ကုဒ်လည်း ရေးပြီးသားမို့လို့ ကိုယ့်ဘာသာ အကောင့်သစ်တစ်ခု အခုလို ထည့်လိုက်ပြီးမှ စမ်းကြည့်ရင်လည်း ရပါတယ်။





အသစ် Create လုပ်ထားတဲ့ User Account နဲ့ `/login` Route ကို စမ်းကြည့်လိုက်တဲ့အခါ ပုံမှာပြထားသလို `token` ရော User Data တွေရော ပြန်ရသွားတာ ဖြစ်ပါတယ်။ API ဘက်မှာ ဆက်ရေးစရာတွေ ကျန်သေးပေမယ့် ဒီမှာခဏထားပြီး **Client App** ဘက်မှာ Register, Login, Profile လုပ်ဆောင်ချက်တွေ အရင်သွားထည့်ကြပါမယ်။

Register

Login Page နဲ့ Register Page တွေက React Router အကြောင်းလေ့လာခဲ့စဉ်မှာ ကြိုထည့်ခဲ့ကြပြီး ဖြစ်ပါတယ်။ အခုလိုအပ်တဲ့ လုပ်ဆောင်ချက်တွေ ထပ်မထည့်ခင် `libs` အမည်နဲ့ ဖိုဒါတစ်ခုဆောက်ပြီး `fetcher.js` အမည်နဲ့ ကုဒ်ဖိုင်တစ်ခုထည့်ပေးပါ။ API နဲ့ဆက်သွယ်ရတဲ့ကုဒ်တွေကို ဒီဖိုင်ထဲမှာ စုစည်းပြီး ရေးဖို့ဖြစ်ပါတယ်။ ဒါကြောင့် **yaycha** ပရောဂျက်ပါ `src` ရဲ့ ဖွဲ့စည်းပုံက အခုလို ဖြစ်သွားပါလိမ့်မယ်။

```
yaycha
├── ...
├── src/
│   ├── Template.jsx
│   ├── ThemedApp.jsx
│   ├── assets/
│   ├── components/
│   │   ├── AppDrawer.jsx
│   │   ├── Form.jsx
│   │   ├── Header.jsx
│   │   ├── Item.jsx
│   │   └── UserList.jsx
│   ├── libs/
│   │   └── fetcher.js
│   ├── main.jsx
│   └── pages/
│       ├── Comments.jsx
│       ├── Home.jsx
│       ├── Likes.jsx
│       ├── Login.jsx
│       ├── Profile.jsx
│       └── Register.jsx
└── ...
```

`fetcher.js` ထဲမှာ အခုလို ရေးပေးပါ။

JavaScript - src/libs/fetcher.js

```javascript
const api = import.meta.env.VITE_API;

export async function postUser(data) {
  const res = await fetch(`${api}/users`, {
    method: "POST",
    body: JSON.stringify(data),
    headers: {
      "Content-Type": "application/json",
```

```
    },
  });

  if (res.ok) {
    return res.json();
  }

  throw new Error("Error: Check Network Log");
}

export async function postLogin(username, password) {
  const res = await fetch(`${api}/login`, {
    method: "POST",
    body: JSON.stringify({ username, password }),
    headers: {
      "Content-Type": "application/json",
    },
  });

  if (res.ok) {
    return res.json();
  }

  throw new Error("Incorrect username or password");
}
```

ပုံစံခပ်ဆင်ဆင် Function နှစ်ခုပါ။ `postUser()` က API ရဲ့ `/users` Route ကို POST နဲ့ fetch လုပ်ထားပါတယ်။ `res.ok` ဖြစ်ရင် `res.json()` Promise ကို Return ပြန်ပေးထားပါတယ်။ ok မဟုတ်ရင်တော့ Error ဖြစ်ကြောင်း Throw လုပ်ပေးထားပါတယ်။ အလားတူပဲ `postLogin()` က API ရဲ့ `/login` Route ကို POST နဲ့ `fetch()` လုပ်ထားပါတယ်။ React Query နဲ့ တွဲသုံးမှာမို့လို့ Data တွေ Return ပြန်မပေးဘဲ Promise တွေ Return ပြန်ပေးရတာကိုပဲ သတိပြုပေးပါ။

API လည်း Ready ဖြစ်နေပြီ။ API ကို `fetch` လုပ်ပေးနိုင်တဲ့ Function တွေလည်း ရပြီဆိုတော့ လိုအပ်

တဲ့ UI နဲ့ချိတ်ဆက်မှုတွေလုပ်ကြပါမယ်။ `pages/Register.jsx` ကုဒ်ကို အခုလို ပြင်ပေးပါ။

React - src/pages/Register.jsx

```
import {
  Box,
  Button,
  TextField,
  Typography,
  Alert,
} from "@mui/material";

import { useRef, useState } from "react";

import { useMutation } from "react-query";
import { postUser } from "../libs/fetcher";
import { useNavigate } from "react-router-dom";
import { useApp } from "../ThemedApp";

export default function Register() {
  const { setGlobalMsg } = useApp();

  const nameInput = useRef();
  const usernameInput = useRef();
  const bioInput = useRef();
  const passwordInput = useRef();

  const [error, setError] = useState(null);
  const navigate = useNavigate();

  const handleSubmit = () => {
    const name = nameInput.current.value;
    const username = usernameInput.current.value;
    const bio = bioInput.current.value;
    const password = passwordInput.current.value;
```

```
    if (!name || !username || !password) {
      setError("name, username and password required");
      return false;
    }

    create.mutate({ name, username, bio, password });
  };

  const create = useMutation(async data => postUser(data), {
    onError: async () => {
      setError("Cannot create account");
    },
    onSuccess: async user => {
      setGlobalMsg("Account Created");
      navigate("/login");
    },
  });

  return (
    <Box>
      <Typography variant="h3">Register</Typography>

      {error && (
        <Alert
          severity="warning"
          sx={{ mt: 2 }}>
          {error}
        </Alert>
      )}

      <form
        onSubmit={e => {
          e.preventDefault();
          handleSubmit();
```

```
        }}>
        <Box
          sx={{
            display: "flex",
            flexDirection: "column",
            gap: 1,
            mt: 2,
          }}>
          <TextField
            inputRef={nameInput}
            placeholder="Name"
            fullWidth
          />
          <TextField
            inputRef={usernameInput}
            placeholder="Username"
            fullWidth
          />
          <TextField
            inputRef={bioInput}
            placeholder="Bio"
            fullWidth
          />
          <TextField
            inputRef={passwordInput}
            type="password"
            placeholder="Password"
            fullWidth
          />
          <Button
            type="submit"
            variant="contained"
            fullWidth>
            Register
          </Button>
```

```
        </Box>
      </form>
    </Box>
  );
}
```

ဒီအချက်တွေ ထပ်ဖြည့်ပြင်ဆင်လိုက်တာပါ။

- `fetcher` ရဲ့ `postUser` ကို Import လုပ်ထားပါတယ်။
- `<TextField>` တွေအတွက် `ref` တွေကိုယ်စီချိတ်ပေးထားပါတယ်။
- `<form>` ရဲ့ `onSubmit` အတွက် `handleSubmit` ရေးပြီးချိတ်ပေးထားပါတယ်။
- Error ရှိရင်ပြဖို့ `error` State ကြေညာထားပါတယ်။
- `<Alert>` ကို `error` ရှိမှပြဖို့ ပြင်ထားပါတယ်။

အရေးအကြီးဆုံးကတော့ `create` Function ပါပဲ။ `useMutation` နဲ့ API Fetch လုပ်ပြီး User အသစ် ထည့်ပေးတဲ့ ကုဒ်ပါ။ `fetch` ကုဒ်ကို ထပ်မရေးတော့ဘဲ `fetcher` ကနေ Import လုပ်ယူထားတဲ့ `postUser` ကိုသုံးပါတယ်။ `onError` နဲ့ `fetch` Error ရှိခဲ့ရင် Error Message ပြဖို့သတ်မှတ်ထားပြီး၊ `onSuccess` မှာ အားလုံးအဆင်ပြေသွားပြီဖြစ်လို့ Login ဝင်လို့ရအောင် `/login` Page ကို သွားခိုင်းထားပါတယ်။

ဒီထိစုံအောင် ရေးပြီးရင် Account တွေ Register လုပ်ပြီးစမ်းလို့ရပါပြီ။ တစ်ခုခု အဆင်မပြေခဲ့ရင် API, `fetcher`, UI သုံးဆင့်တွဲအလုပ်လုပ်တာဖြစ်လို့ မှားနိုင်တာလည်း သုံးဆင့်ဖြစ်သွားပါတယ်။ အမှားရှာရ ခက်တာမျိုးဖြစ်နိုင်ပါတယ်။ နမူနာကုဒ်ကို သေချာနားလည်အောင်ဖတ်၊ သေချာတိုက်ကြည့်ပြီး ကြိုးစား လုပ်ကြည့်ပါ။

Login

Login အတွက်လည်း API တွေ fetcher တွေက အသင့်ပါပဲ။ pages/Login.jsx ကို အခုလိုပြင်ပေးပါ။

React - src/pages/Login.jsx

```
import {
  Alert,
  Box,
  Button,
  TextField,
  Typography,
} from "@mui/material";

import { useRef, useState } from "react";

import { useNavigate } from "react-router-dom";
import { useApp } from "../ThemedApp";
import { postLogin } from "../libs/fetcher";
import { useMutation } from "react-query";

export default function Login() {
  const usernameInput = useRef();
  const passwordInput = useRef();

  const [error, setError] = useState(null);

  const handleSubmit = () => {
    const username = usernameInput.current.value;
    const password = passwordInput.current.value;

    if (!username || !password) {
      setError("username and password required");
      return false;
    }
```

```
    login.mutate({ username, password });
  };

  const login = useMutation(
    async ({ username, password }) => postLogin(username, password),
    {
      onError: async () => {
        setError("Incorrect username or password");
      },
      onSuccess: async result => {
        setAuth(result.user);
        localStorage.setItem("token", result.token);
        navigate("/");
      },
    }
  );

  const navigate = useNavigate();
  const { setAuth } = useApp();

  return (
    <Box>
      <Typography variant="h3">Login</Typography>

      {error && (
        <Alert
          severity="warning"
          sx={{ mt: 2 }}>
          {error}
        </Alert>
      )}

      <form
        onSubmit={e => {
```

```
          e.preventDefault();
          handleSubmit();
        }}>
        <Box
          sx={{
            display: "flex",
            flexDirection: "column",
            gap: 1,
            mt: 2,
          }}>
          <TextField
            inputRef={usernameInput}
            placeholder="Username"
            fullWidth
          />
          <TextField
            inputRef={passwordInput}
            type="password"
            placeholder="Password"
            fullWidth
          />
          <Button
            type="submit"
            variant="contained"
            fullWidth>
            Login
          </Button>
        </Box>
      </form>
    </Box>
  );
}
```

စောစောက Register Page လိုပါပဲ။

- fetcher ရဲ့ postLogin ကို Import လုပ်ထားပါတယ်။
- <TextField> တွေအတွက် ref တွေကိုယ်စီချိတ်ပေးထားပါတယ်။
- <form> ရဲ့ onSubmit အတွက် handleSubmit ရေးပြီးချိတ်ပေးထားပါတယ်။
- Error ရှိရင်ပြဖို့ error State ကြေညာထားပါတယ်။
- <Alert> ကို error ရှိမှပြဖို့ ပြင်ထားပါတယ်။

useMutation နဲ့ရေးထားတဲ့ login Function က အရေးအကြီးဆုံးပါပဲ။ onSuccess မှာ ပြန်ရတဲ့ token ကို localStorage ထဲမှာ သိမ်းလိုက်ပြီး ပြန်ရတဲ့ user Data ကိုတော့ setAuth နဲ့ auth ထဲမှာ ထည့်သိမ်းလိုက်တာပါ။

စမ်းကြည့်ရတာ ပြည့်စုံစေဖို့အတွက် components/AppDrawer.jsx မှာ အခုလို ပြင်ပေးပါ။

React - src/components/AppDrawer.jsx

```
// imports

export default function AppDrawer() {
  ...

  return (
    <div>
      <Drawer ...>
        <Box ...>
          <Box ...>
            <Avatar .../>
            <Typography sx={{ fontWeight: "bold" }}>
              {auth ? auth.name : "Guest"}
            </Typography>
          </Box>
        </Box>
        <List onClick={() => setShowDrawer(false)}>
          ...
```

```
          {auth && (
            <>
              <ListItem>
                <ListItemButton
                  onClick={() => navigate(`/profile/${auth.id}`)}>
                  <ListItemIcon>
                    <ProfileIcon />
                  </ListItemIcon>
                  <ListItemText>Profile</ListItemText>
                </ListItemButton>
              </ListItem>
              <ListItem>
                <ListItemButton
                  onClick={() => {
                    localStorage.removeItem("token");
                    setAuth(null);
                    navigate("/");
                  }}>
                  <ListItemIcon>
                    <LogoutIcon color="error" />
                  </ListItemIcon>
                  <ListItemText>Logout</ListItemText>
                </ListItemButton>
              </ListItem>
            </>
          )}

          {!auth && ( ... )}
        </List>
      </Drawer>
    </div>
  );
}
```

`auth` ရှိရင် `auth.name` ကို ပြခိုင်းပြီး မရှိရင် Guest ကို ပြခိုင်းထားပါတယ်။ ပြီးတော့ `/profile` ကို သွားတဲ့အခါ သက်ဆိုင်ရာ `auth.id` နဲ့ချိတ်ပြီးသွားထားပါတယ်။ Logout Button မှာလည်း `localStorage` ထဲမှာ သိမ်းထားတဲ့ Token ကို ပြန်ရှင်းထားပါတယ်။

ဒါတွေရေးပြီးရင် Account တွေ Register လုပ်ကြည့်၊ Login ပြန်ဝင်စမ်းကြည့်လို့ ရသွားပါပြီ။

Profile

ဆက်လက်ပြီး Profile Page ကိုလည်း လိုသလိုပြင်ကြပါမယ်။ `libs/fetcher.js` မှာ အခုလို ထပ်ဖြည့်ပေးပါ။

JavaScript - src/libs/fetcher.js

```
export async function fetchUser(id) {
  const token = getToken();
  const res = await fetch(`${api}/users/${id}`, {
    headers: {
      Authorization: `Bearer ${token}`,
    },
  });

  return res.json();
}
```

ကြိုရေးပြီးသားဖြစ်တဲ့ `/users/:id` Route ကို `fetch` လုပ်လိုက်တာပါ။ `id` Parameter လိုအပ်တာကို သတိပြုပါ။

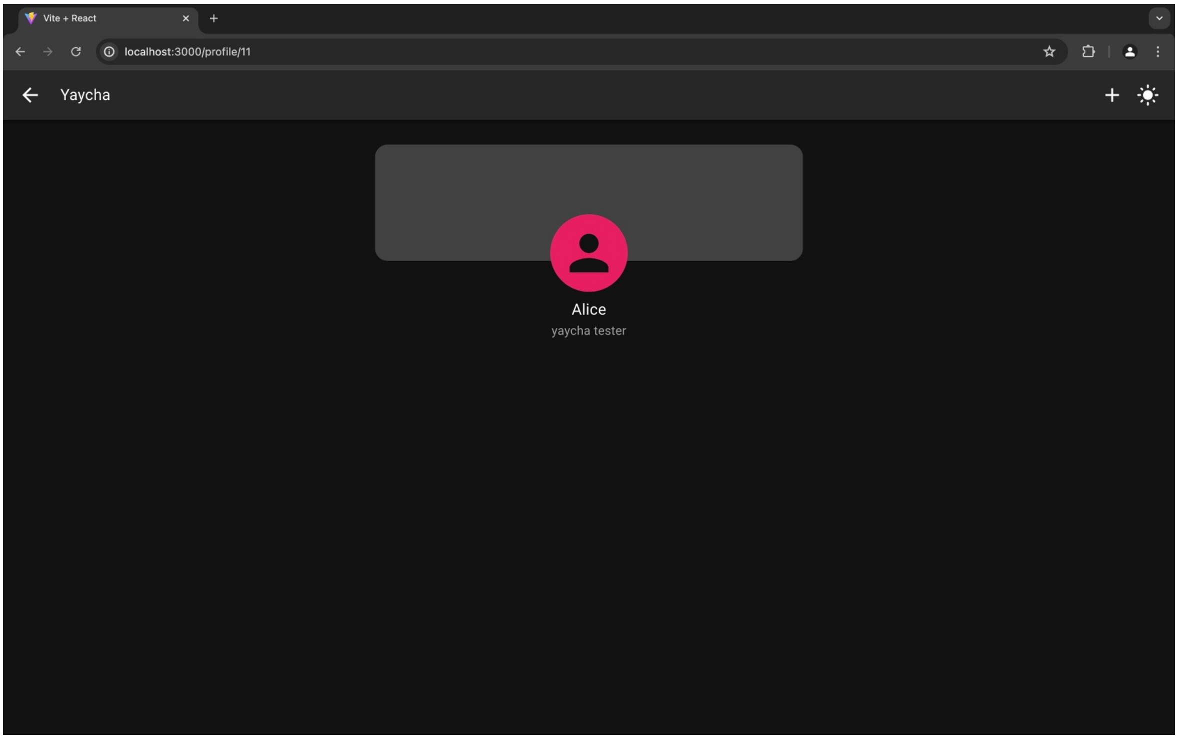
`pages/Profile.jsx` မှာ အခုလိုပြင်ပေးပါ။

React - src/pages/Profile.jsx

```
import { Avatar, Box, Typography } from "@mui/material";
import { pink } from "@mui/material/colors";

import { useParams } from "react-router-dom";
import { fetchUser } from "../libs/fetcher";
import { useQuery } from "react-query";

export default function Profile() {
  const { id } = useParams();

  const { isLoading, isError, error, data } = useQuery(
    `users/${id}`,
    async () => fetchUser(id)
  );

  if (isError) {
    return (
      <Box>
        <Alert severity="warning">{error.message}</Alert>
      </Box>
    );
  }

  if (isLoading) {
    return <Box sx={{ textAlign: "center" }}>Loading...</Box>;
  }

  return (
    <Box>
      <Box sx={{
        bgcolor: "banner",
        height: 150,
        borderRadius: 4 }}></Box>
      <Box
```

```
        sx={{
          mb: 4,
          marginTop: "-60px",
          display: "flex",
          flexDirection: "column",
          justifyContent: "center",
          alignItems: "center",
          gap: 1,
        }}>
        <Avatar sx={{
          width: 100,
          height: 100,
          bgcolor:
          pink[500] }} />
        <Box sx={{ textAlign: "center" }}>
          <Typography>{data.name}</Typography>
          <Typography sx={{ fontSize: "0.8em", color: "text.fade" }}>
            {data.bio}
          </Typography>
        </Box>
      </Box>
    </Box>
  );
}
```

`useParams` နဲ့ `id` ကို ယူလိုက်ပြီး `useQuery` နဲ့ စောစောကရေးလိုက်တဲ့ `fetchUser` ကို သုံးလိုက်တာပါ။ ပြန်ရလာတဲ့ Data တွေထဲက `name` နဲ့ `bio` ကို သူ့နေရာနဲ့သူ ထည့်ပြထားပါတယ်။ Login ဝင်လိုက်ပြီး Drawer Menu ထဲက Profile ကိုနှိပ်ပြီး စမ်းကြည့်လို့ရပါတယ်။ ရလဒ်က အခုလိုဖြစ်မှာပါ။



`components/Item.jsx` ကိုလည်း အခုလိုပြင်သင့်ပါတယ်။

React - src/components/Item.jsx

```
// imports

export default function Item({ ... }) {
  ...

  return (
    <Card sx={{ mb: 2 }}>
      ...

      <CardContent ...>
        ...

        <Box
          onClick={e => {
```

```
                navigate(`/profile/${item.user.id}`);
                e.stopPropagation();
            }}
            sx={{ ... }}>
            <UserIcon fontSize="12" color="info" />
            <Typography variant="caption">{item.user.name}</Typography>
          </Box>
        </CardContent>
      </Card>
    );
}
```

User Name ရှိတဲ့ Box မှာ onClick သွားထည့်ပေးလိုက်တာပါ။ ဒါကြောင့် User ကိုနှိပ်လိုက်ရင် သူ့ Profile ကိုရောက်သွားမှာဖြစ်ပါတယ်။

useQuery နဲ့ Fetch လုပ်လိုက်တဲ့ data ထဲမှာ User ရဲ့ posts တွေလည်းပါပါတယ်။ ဒါကြောင့် ပြချင်ရင် Profile မှာ User ရဲ့ပို့စ်တွေထည့်ပြလို့ရပါတယ်။ နမူနာမှာ ပြမထားပါဘူး။ Exercise အနေနဲ့ ကိုယ့်ဘာသာ ဖြည့်ရေးကြည့်လိုက်ပါ။

ဒါဟာ User Authentication System တစ်ခု Implement လုပ်ပေးလိုက်တာပါပဲ။ ကျန်လုပ်ဆောင်ချက်တွေ မဆက်ခင်၊ API ဘက်မှာ လိုအပ်တာတွေ ဆက်လုပ်ကြပါမယ်။

Auth Middleware

Middleware ဆိုတာ ကြားဖြတ်လုပ်တဲ့အလုပ်လို့ပဲ ပြောရပါမယ်။ ကြိုတင်ပြင်ဆင်တဲ့ကိစ္စတွေ၊ ကြိုတင်စစ်ဆေးတဲ့ကိစ္စတွေကို Middleware အနေနဲ့ ရေးထားနိုင်ပါတယ်။ Express နဲ့ ရေးသားထားတဲ့ API မှာ `cors()` တို့ `express.json()` တို့လို Middleware တွေကိုထည့်သုံးလက်စ ရှိနေပါတယ်။ `cors()` က ကျန်တဲ့အလုပ်တွေ မလုပ်ခင် Cross-Origin Resource Sharing Setting တွေကြိုထည့်ပေးတယ်။ `express.json()` က Request Body မှာပါလာတဲ့ JSON Data တွေကို Parse လုပ်ပေးပါတယ်။

အလားတူ Middleware တွေ ကိုယ့်ဘာသာလည်း အလွယ်တကူရေးလို့ရပါတယ်။ ရေးနည်းက Route Function တွေရေးနည်းနဲ့ အတူတူပါပဲ။ ကျန်တဲ့အလုပ်တွေ မလုပ်ခင် Authentication ကြိုစစ်ပေးတဲ့ Middleware တစ်ခုရေးကြပါမယ်။ **`yaycha-api`** ပရောဂျက်မှာ `middlewares` အမည်နဲ့ ဖိုဒါတစ်ခု ဆောက်ပြီး `auth.js` အမည်နဲ့ ကုဒ်ဖိုင်တစ်ခုရေးပေးပါ။ ဒါကြောင့် **`yaycha-api`** ရဲ့ ဖွဲ့စည်းပုံက အခု လိုဖြစ်သွားပါလိမ့်မယ်။

```
yaycha-api
├── node_modules/
├── index.js
├── middlewares/
│   └── auth.js
├── package-lock.json
├── package.json
├── prisma/
├── prismaClient.js
└── routers/
    ├── content.js
    └── user.js
```

`auth.js` မှာ ရေးရမယ့်ကုဒ်က ဒီလိုပါ။

JavaScript - API - middlewares/auth.js

```javascript
const express = require("express");
const jwt = require("jsonwebtoken");

/***
 * @param {express.Request} req
 * @param {express.Response} res
 * @param {express.NextFunction} next
 */
function auth(req, res, next) {
  const { authorization } = req.headers;
  const token = authorization && authorization.split(" ")[1];
```

```
  if (!token) {
    return res.status(400).json({ msg: "token required" });
  }

  const user = jwt.decode(token, process.env.JWT_SECRET);
  if (!user) {
    return res.status(401).json({ msg: "incorrect token" });
  }

  res.locals.user = user;

  next();
}

module.exports = { auth };
```

`express` နဲ့ `jsonwebtoken` တို့ကို Import လုပ်ထားပါတယ်။ `auth` Function တစ်ခုရေးပြီး `exports` လုပ်ပေးထားပါတယ်။ `auth` Function က ရိုးရိုး Express Route Function တစ်ခုလို `req` နဲ့ `res` ကိုလက်ခံအလုပ်လုပ်ပါတယ်။ ထူးခြားချက်အနေနဲ့ `next` လည်းပါသွားပါတယ်။ `next` ဟာ Function ဖြစ်ပြီး သူ့ကို Run လိုက်ရင် ဆက်လုပ်ရမယ့် Middleware တွေ၊ Route Function တွေကို လက်ဆင့်ကမ်း အလုပ်လုပ်ပေးသွားမှာပါ။

`auth` Function အပေါ်မှာ ရေးထားတဲ့ Comment ကို `JSDoc` Comment လို့ခေါ်ပါတယ်။ Code Documentation တွေကို Comment အနေနဲ့ ရေးတဲ့ရေးနည်းပါ။ Variable Data Type တွေ Function Parameter Type တွေကို Hint လုပ်ပေးဖို့ အသုံးဝင်ပါတယ်။ နမူနာမှာ `req` ဟာ `Express.Request` Object ဖြစ်ပြီး `res` ဟာ `Express.Response` Object ဖြစ်တယ်လို့ Type Hint လုပ်ပေးထားပါတယ်။ `next` ကတော့ `Express.NextFunction` ဖြစ်ပါတယ်။ ဒီလိုရေးပေးထားတဲ့အတွက် Code Editor က ဒီ Parameter တွေအကြောင်းကို နားလည်သွားပြီး Autocomplete တွေ မှန်အောင် လုပ်ပေးနိုင်သွားမှာ ဖြစ်ပါတယ်။ အသုံးဝင်တဲ့ လုပ်ဆောင်ချက်ဖြစ်လို့ တစ်ခါထဲ ထည့်ပေးလိုက်တာပါ။

`auth` Function ကုဒ်ကို လေ့လာကြည့်ပါ။ `authorization` Header ပါမပါ စစ်ထားပါတယ်။ Request နဲ့အတူ Token က အခုလို ပါလာရမှာပါ။

Request Headers

```
headers: {
  Authorization: "Bearer token"
}
```

ဒါကြောင့် `authorization` Header ပါလာခဲ့ရင် `split()` နဲ့ နှစ်ပိုင်းပိုင်းထားပါတယ်။ ရှေ့က Bearer ဆိုတဲ့ စာဖြစ်ရမှာဖြစ်ပြီး နောက်က Token အမှန်ဖြစ်ရမှာဖြစ်ပါတယ်။ နမူနာမှာ ရှေ့ကစာ Bearer ဟုတ်မဟုတ် စစ်မထားပါဘူး။ နောက်က Token မှန်မမှန်ကိုပဲ စစ်မှာဖြစ်ပါတယ်။ ဒါကြောင့် Index `[1]` ကို တန်းယူထားတာပါ။

`token` လုံးဝမပါရင် 400 Bad Request ကိုပြန်ပေးထားပါတယ်။ ပါလာရင်တော့ `jwt.decode` နဲ့ စစ်ကြည့်ပါတယ်။ `token` မမှန်လို့ Decode လုပ်လို့မရရင် 401 Unauthorized ကိုပြန်ပေးထားပါတယ်။ `token` မှန်တယ်ဆိုရင်တော့ Decode လုပ်လို့ရလာတဲ့ User Data ကို `res.locals.user` မှာ ထည့်သိမ်းပေးလိုက်ပါတယ်။ တခြား Middleware တွေ Route Function တွေကို ယူသုံးချင်ရင် သုံးလို့ရအောင် သိမ်းပေးလိုက်တာပါ။ အားလုံးမှန်တယ်၊ ရှေ့ဆက်သွားလို့ရတယ်ဆိုတဲ့ သဘောမျိုးနဲ့ `next()` ကို Run ပေးထားပါတယ်။

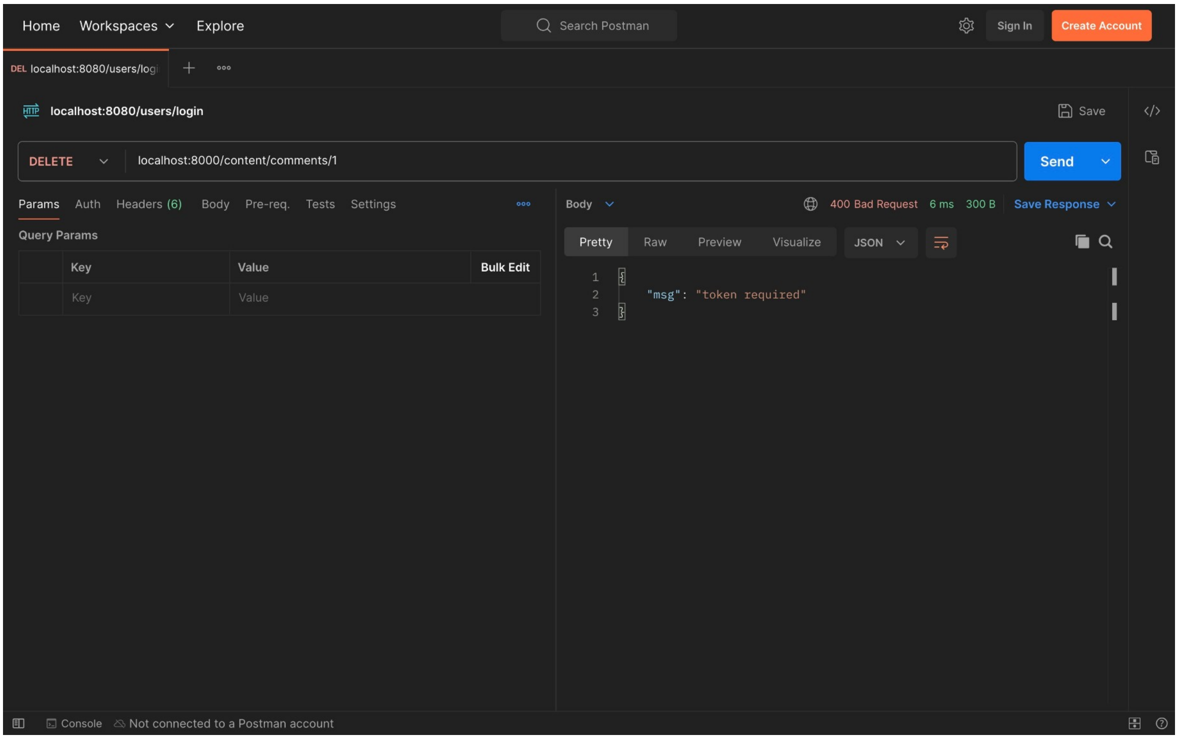
ဒါ Authentication `token` စစ်တဲ့ Middleware တစ်ခု ရေးလိုက်တာပါပဲ။ ဒီ Middleware ကို အသုံးပြုပြီး တချို့ Route တွေကို Protect လုပ်ကြပါမယ်။ `routers/content.js` မှာ ရေးထားတဲ့ `DELETE` Route နှစ်ခုကို အခုလို ပြင်ပေးလိုက်ပါ။

JavaScript - API - routers/content.js

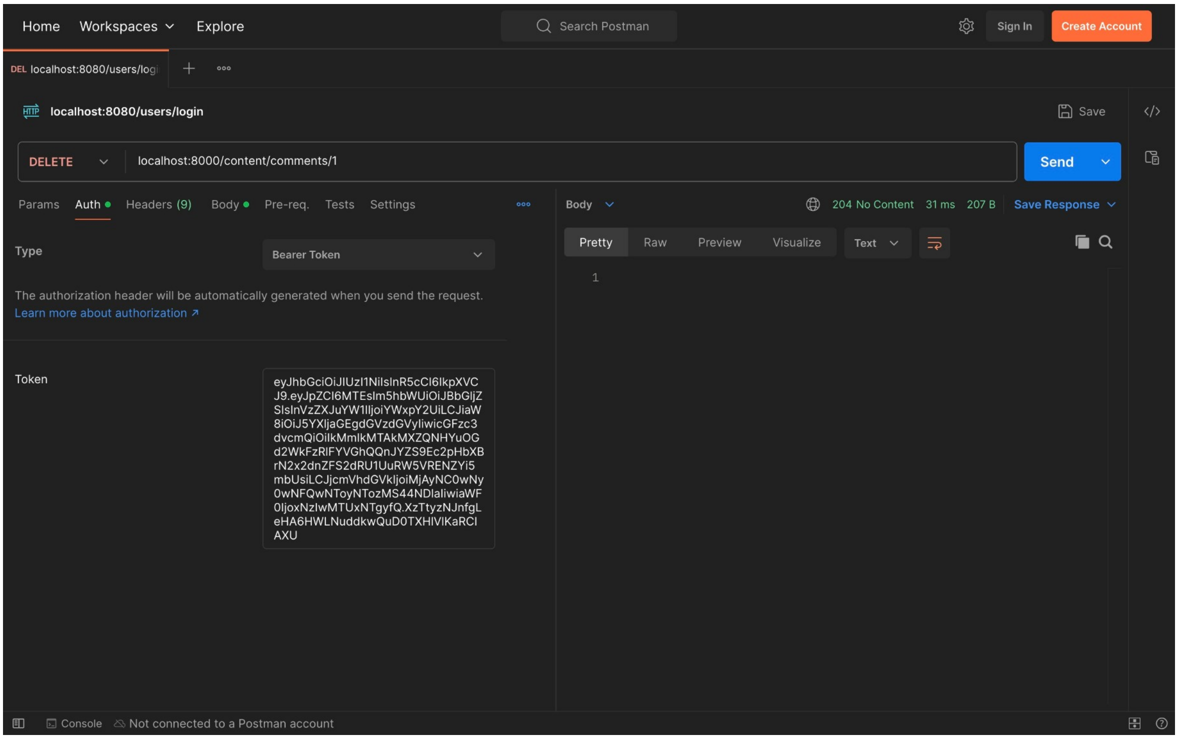
```
const { auth } = require("../middlewares/auth");

router.delete("/posts/:id", auth, async (req, res) => {
  ...
```

```
});

router.delete("/comments/:id", auth, async (req, res) => {
  ...
});
```

auth ကို Import လုပ်ပြီး Route Function ရဲ့ရှေ့မှာ auth Middleware ကို တစ်ဆင့်ခံပေးလိုက်တာပါ။ Route Function ကုဒ်တွေကို ပြင်စရာမလိုပါဘူး။ စမ်းကြည့်လိုက်ရင် အခုလို ဖြစ်မှာပါ။



/content/comments/1 ကို DELETE နဲ့သွားတဲ့အခါ token မပါလို့ 400 Bad Request ပြန်ရတာပါ။ ဒါကြောင့် စောစောကရေးထားတဲ့ /login Route ကို username, password အမှန်နဲ့ ပြန်သွားပြီး ရလာတဲ့ token ကို ကူးယူထားလိုက်ပါ။ ပြီးတဲ့အခါ ဒီလိုစမ်းကြည့်ပါ။



Postman ရဲ့ `Authorization` Section မှာ `Bearer Token` ကိုရွေးပြီး ရထားတဲ့ `Token` ကိုထည့်ပေးလိုက်တာပါ။ ဒီလိုထည့်ပေးလိုက်တဲ့အတွက် ဖျက်လို့ရသွားတယ်ဆိုတာကို တွေ့ရမှာပဲ ဖြစ်ပါတယ်။

Authorization

ပိုပြည့်စုံသွားအောင် `token` ရှိယုံနဲ့ ဖျက်လို့မရဘဲ၊ ကိုယ်ထည့်ထားတဲ့ Post တွေ Comment တွေဖြစ်မှသာ ဖျက်လို့ရအောင် Middleware တစ်ခု ထပ်ထည့်ကြပါမယ်။ `middlewares/auth.js` ကို အခုလိုဖြည့်စွက်လိုက်ပါ။

JavaScript - API - middlewares/auth.js

```
const express = require("express");
const jwt = require("jsonwebtoken");

const prisma = require("../prismaClient");
```

```
/***
 * ...
 */
function auth(req, res, next) {
  ...
}

/***
 * @param {('post'|'comment')} type
 */
function isOwner(type) {
  /***
   * @param {express.Request} req
   * @param {express.Response} res
   * @param {express.NextFunction} next
   */
  return async (req, res, next) => {
    const { id } = req.params;
    const user = res.locals.user;

    if (type == "post") {
      const post = await prisma.post.findUnique({
        where: { id: Number(id) },
      });

      if (post.userId == user.id) return next();
    }

    if (type == "comment") {
      const comment = await prisma.comment.findUnique({
        where: { id: Number(id) },
        include: {
          post: true,
        },
      });
```

```
      if (comment.userId == user.id || comment.post.userId == user.id)
        return next();
    }

    res.status(403).json({ msg: "Unauthorize to delete" });
  };
}

module.exports = { auth, isOwner };
```

`prismaClient` ကို Import လုပ်ထားတာကို သတိပြုပါ။ ပြီးတဲ့အခါ `isOwner` အမည်နဲ့ Middleware Function တစ်ခုထပ်ရေးပေးလိုက်တာပါ။ ဒီ Function ဟာ ရိုးရိုး Function မဟုတ်ဘဲ Higher Order Function ဖြစ်ပါတယ်။ သူ့ကိုခေါ်လိုက်ရင် Function တစ်ခုကို Return ပြန်ပေးမှာပါ။ တစ်ကယ့် Middleware Function က အထဲမှာဖြစ်ပါတယ်။ type Parameter ကို အရင်ကြို လက်ခံချင်လို့ ဒီလိုရေးလိုက်တာပါ။ Middleware Function မှာ အဲ့ဒီလို သူနဲ့မဆိုင်တဲ့ Parameter တွေ လက်ခံလို့မရပါဘူး။ type က post သို့မဟုတ် comment ဖြစ်ရမယ်လို့ အပေါ်မှာ `JSDoc` နဲ့ Hint လုပ်ပေးထားတာကို လည်း သတိပြုပါ။

Middleware Function ကို ဆက်ကြည့်ကြပါမယ်။ Route Parameter `id` ကို ယူထားပါတယ်။ `res.locals.user` ထဲမှာ သိမ်းထားတဲ့ လက်ရှိ Authenticated User ရဲ့ Data ကိုလည်း ယူထားပါတယ်။ ပြီးတဲ့အခါ `type` ကို ကြည့်ပြီး `prisma` နဲ့ သက်ဆိုင်ရာ Post သို့မဟုတ် Comment ကို ထုတ်ယူပါတယ်။ ထုတ်ယူပြီးတဲ့အခါ Post ရဲ့ `userId` နဲ့ လက်ရှိ Authenticated User ရဲ့ `id` တူမတူစစ်ပါတယ်။ တူရင် လက်ခံပါတယ်။

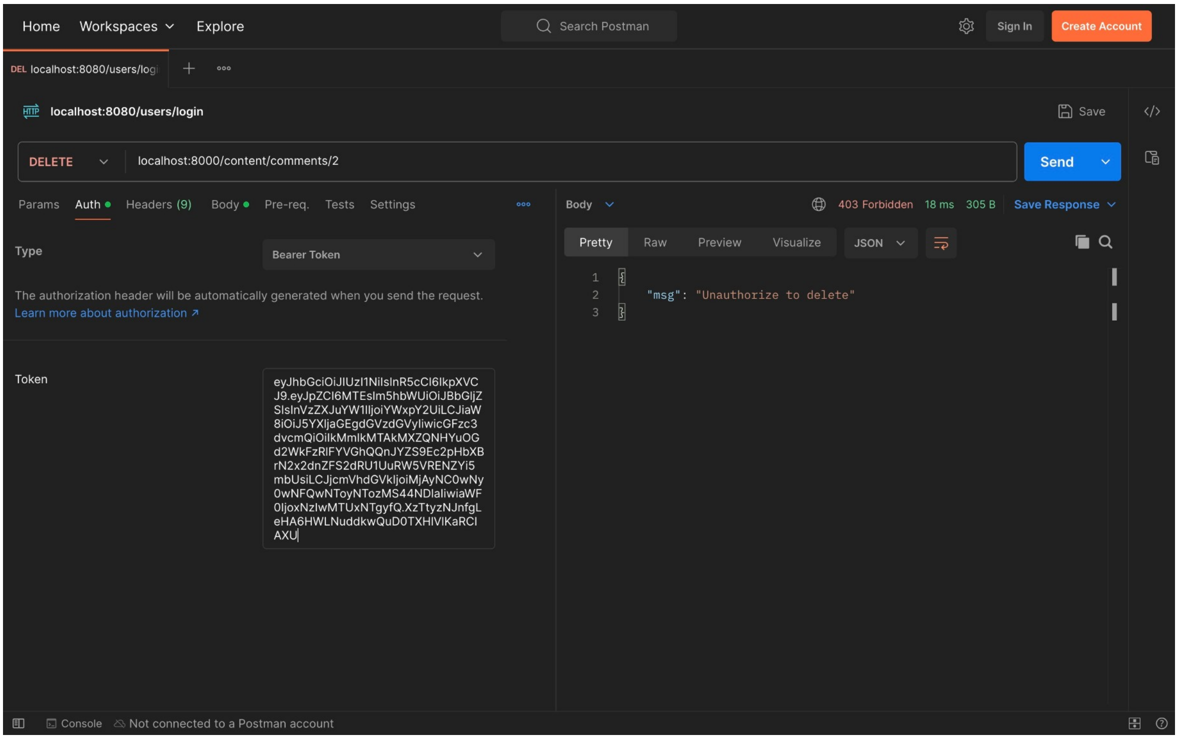
Comment မှာတော့ `userId` သာမက `comment.post.userId` ကိုလည်း စစ်ထားပါတယ်။ တူရင် လက်ခံပါတယ်။ ဆိုလိုတာ Comment ကို ဖျက်တဲ့အခါ Comment ပိုင်ရှင်ကဖျက်လို့ရသလို၊ Comment ရှိနေတဲ့ Post ရဲ့ ပိုင်ရှင်က ဖျက်ရင်လည်း ရချင်တာပါ။ ဒါကြောင့် `comment.userId` နဲ့ မတူရင်တောင် `comment.post.userId` နဲ့တူရင် လက်ခံထားပါတယ်။

တစ်ခုမှ အဆင်မပြေရင်တော့ 403 Forbidden ကို ပြန်ပေးထားပါတယ်။ ဒီ Middleware ကိုလည်း စောစောက DELETE တွေမှာ အခုလို တွဲပေးပါ။

JavaScript - API - routers/content.js

```
const { auth, isOwner } = require("../middlewares/auth");

router.delete("/posts/:id", auth, isOwner("post"), async (req, res) => {
  ...
});

router.delete("/comments/:id", auth, isOwner("comment"), async (req, res) => {
  ...
});
```

`isOwner("post")` နဲ့ လှမ်းခေါ်လိုက်တော့မှ သူပြန်ပေးတဲ့ Middleware Function ကိုရသွားတဲ့ သဘော ဖြစ်ပါတယ်။ Higher Order Function တွေဟာ အခုလို အသုံးဝင်တဲ့ ရေးနည်းတစ်ခုပါ။ အခုနေ စမ်းကြည့်ရင် ရလဒ်က ဒီလိုဖြစ်မှာပါ။



token အမှန်ပါပေမယ့် သူနဲ့ဆိုင်တဲ့ Comment မဟုတ်လို့ ဖျက်ခွင့်မရှိဘဲ 403 Forbidden ကို ပြန်လည်ရရှိတာ ဖြစ်ပါတယ်။ ဒီနည်းနဲ့ Authentication သာမက Authorization လုပ်ဆောင်ချက်ပါ နမူနာရသွားပြီဖြစ်ပါတယ်။

ဆက်လက်ပြီး အရင်က Authentication လုပ်ဆောင်ချက်မရှိသေးလို့ ထည့်မလုပ်ဘဲ ချန်ထားတဲ့ New Post, New Comment စတဲ့ API Feature တွေထည့်ကြပါမယ်။ `routers/content.js` မှာ အခုလို ဖြည့်စွက်ပေးပါ။

JavaScript - API - routers/content.js

```
router.post("/posts", auth, async (req, res) => {
  const { content } = req.body;
  if (!content) {
    return res.status(400).json({ msg: "content required" });
  }

  const user = res.locals.user;
```

```
  const post = await prisma.post.create({
    data: {
      content,
      userId: user.id,
    },
  });

  const data = await prisma.post.findUnique({
    where: { id: Number(post.id) },
    include: {
      user: true,
      comments: {
        include: { user: true },
      },
    },
  });

  res.json(data);
});

router.post("/comments", auth, async (req, res) => {
  const { content, postId } = req.body;
  if (!content || !postId) {
    return
      res.status(400).json({ msg: "content and postId required" });
  }

  const user = res.locals.user;

  const comment = await prisma.comment.create({
    data: {
      content,
      userId: Number(user.id),
      postId: Number(postId),
```

```
    },
  });

  comment.user = user;

  res.json(comment);
});
```

`/posts` အတွက် `POST` Route တစ်ခုနဲ့ `/comments` အတွက် `POST` Route တစ်ခု ထပ်ထည့်လိုက်တာပါ။ `auth` Middleware ကို နှစ်ခုလုံးမှာသုံးထားပါတယ်။ ဒါကြောင့် Authenticated User ဖြစ်မှ ဒီလုပ်ဆောင်ချက်တွေကို ရမှာပါ။ `auth` ကိုတစ်ဆင့်ခံထားတဲ့အတွက် လိုအပ်တဲ့ လက်ရှိ Authenticated User Data ကို `res.locals.user` ကနေ ယူသုံးလို့ရသွားပါတယ်။

Post Data ထည့်ဖို့အတွက် Request Body မှာ content ပါရပါမယ်။ မပါရင် 400 Bad Request ပြန်ပေးထားပါတယ်။ `content` ပါရင် `user.id` နဲ့တွဲပြီး Post Data ကို သိမ်းလိုက်ပါတယ်။ သိမ်းပြီးမှာ `findUnique()` နဲ့ ပြန်ထုတ်ယူထားပါတယ်။ သက်ဆိုင်ရာ User နဲ့ Comment Relationship Data အပြည့်အစုံ လိုချင်လို့ပါ။ ပြီးတော့မှ အခုထည့်လိုက်တဲ့ Post အသစ်ကို Relationship Data အပြည့်အစုံနဲ့အတူ Response ပြန်ပေးထားပါတယ်။

Comment မှာလည်း အလုပ်လုပ်ပုံ အတူတူပါပဲ။ Comment ထည့်ချင်ရင် `content` နဲ့ `postId` ပါရမှာဖြစ်ပါတယ်။ Comment မှာတော့ Relationship Data ပါအောင် ပြန်ထုတ်မနေတော့ပါဘူး။ ရှိပြီးသား `user` ကိုပဲ တွဲထည့်ပေးထားပါတယ်။

ဒီလောက်ဆိုရင် Content အသစ်တွေ ထပ်ထည့်လို့ရတဲ့ လုပ်ဆောင်ချက်လည်း ပါသွားပါပြီ။ နောက်ဆုံးတစ်ခုအနေနဲ့ `routers/users.js` မှာ ဒါလေးဖြည့်ပေးပါ။

JavaScript - API - routers/user.js

```
router.get("/verify", auth, async (req, res) => {
  const user = res.locals.user;
  res.json(user);
});
```

/verify Route တစ်ခု ထပ်ထည့်လိုက်တာပါ။ ဘာလုပ်ဆောင်ချက်မှတော့ ထူးထူးထွေထွေမပါပါဘူး။ auth Middleware ခံပြီး ရလာတဲ့ User Data ကို ပြန်ပေးထားတာပါ။ ဘာအတွက်လည်း ပြောပြပါမယ်။

Verification

Client App ရဲ့ လက်ရှိအနေအထားမှာ လိုအပ်ချက်တစ်ခုရှိနေပါတယ်။ Login Success ဖြစ်သွားချိန်မှာ User Data ကို auth AppContext ထဲမှာ ထည့်ပေးလိုက်ပါတယ်။ ဒါကြောင့် လက်ရှိ User ကို Authenticated User အနေနဲ့ အလုပ်လုပ်ပေးနိုင်သွားပါတယ်။ အဆင်ပြေပါတယ်။

ဒါပေမယ့် Page ကို Manual Refresh လုပ်လိုက်ရင် App ကို အစကနေ ပြန်စလိုက်သလိုဖြစ်သွားမှာပါ။ auth Value လည်း default Value ပြန်ဖြစ်သွားပြီး User လည်း Authenticated User ဟုတ်မှာ မဟုတ်တော့ပါဘူး။ Refresh လုပ်ပြီး စမ်းကြည့်လို့ရပါတယ်။

ဒီလိုမဖြစ်အောင် Page Refresh အပါအဝင် App ကို စတင်အလုပ်လုပ်ချိန်မှာ User Authentication ကို Verify လုပ်ပေးဖို့လိုပါတယ်။ **yaycha** App ရဲ့ fetcher.js မှာ အခုလို ဖြည့်ပေးရပါမယ်။

JavaScript - src/libs/fetcher.js

```
function getToken() {
  return localStorage.getItem("token");
}

export async function fetchVerify() {
  const token = getToken();
  const res = await fetch(`${api}/verify`, {
    headers: {
```

```
      Authorization: `Bearer ${token}`,
    },
  });

  if (res.ok) {
    return res.json();
  }

  return false;
}
```

အရင်ဆုံး `localStorage` ထဲကနေ `token` ကိုထုတ်ပေးတဲ့ `getToken()` ရေးလိုက်ပါတယ်။ ပြီးတော့မှ စောစောက API ဘက်မှာ ကြိုရေးခဲ့တဲ့ `/verify` Route ကို fetch လုပ်ပေးတဲ့ `fetchVerify()` Function ရေးပေးလိုက်တာပါ။ `/verify` Route ကိုသွားတဲ့အခါ `token` ပါဖို့လိုပါတယ်။ ဒါကြောင့် Authorization Header နဲ့ `token` ကို ထည့်ပေးထားတာ သတိပြုပါ။

Authorization အပါအဝင် Header Information တွေကို စာလုံးအကြီးအသေး ကြိုက်သလိုရေးလို့ရပါတယ်။ API မှာ အသေးနဲ့ရေးပြီး ဒီမှာအကြီးနဲ့ရေးတာ ဘာသဘောလည်း စဉ်းစားနေမှာစိုးလို့ပါ။ ထုံးစံလိုဖြစ်နေလို့ `fetch` လုပ်ချိန်မှာ အကြီးနဲ့စပြီးရေးလိုက်တာပါ။ API ဘက်မှာတော့ အသေးနဲ့ပဲ ရေးရပါတယ်။ ပြီးတဲ့အခါ `ThemedApp.jsx` မှာ အခုလိုဖြည့်ပေးဖို့လိုပါတယ်။

React - src/ThemedApp.jsx

```
import { useEffect } from "react";
import { fetchVerify } from "./libs/fetcher";

useEffect(() => {
  fetchVerify().then(user => {
    if (user) setAuth(user);
  });
}, []);
```

ဒီအတိုင်းကူးယူလို့ မရပါဘူး။ သူ့နေရာနဲ့သူ မှန်အောင်ရေးပေးပါ။ Import တွေကို အပေါ်မှာရေးပြီး useEffect ကို Component Function ထဲမှာ ရေးပေးပါ။ fetchVerify လုပ်ပြီး ပြန်ရလာတဲ့ user Data ကို setAuth နဲ့ auth ထဲမှာ သိမ်းပေးလိုက်တာပါ။ ဒါကြောင့် App ကို စလိုက်တာနဲ့ User က token ရှိမယ်ဆိုရင် Authenticated ဖြစ်နေပြီး ဖြစ်ပါလိမ့်မယ်။

React Query ကို မသုံးဘဲ ကိုယ့်ဘာသာ useEffect နဲ့ရေးလိုက်တာ ထူးခြားပါတယ်။ auth, setAuth State တွေကို Manual ရေးပြီး သုံးလက်စဖြစ်နေတော့ ရေးပြီးသားကုဒ်တွေ လိုက်မပြင်ချင်တော့လို့ပါ။ ဖြစ်သင့်တာက auth ဟာ React Query ကပြန်ပေးတဲ့ data ဖြစ်သင့်ပါတယ်။ React Query နဲ့ Manual ရေးထားတဲ့ State နဲ့ကိုတော့ တွဲမသုံးသင့်ပါဘူး။ Pattern ရောပြီး ပျက်သွားပါလိမ့်မယ်။ နောင် ဒုက္ခပေးတာတွေ ရှိလာနိုင်ပါတယ်။ ဒါကြောင့် ဒီတစ်ခုကို ခြွင်းချက်အနေနဲ့ React Query မသုံးတော့ဘဲ ကိုယ့်ဘာသာ useEffect နဲ့ fetch လုပ်ထားလိုက်တာပါ။

စမ်းကြည့်လို့ရပါပြီ။ Login ဝင်ထားပြီး Refresh လုပ်လိုက်ရင်လည်း User က Login ဆက်ဖြစ်နေပါလိမ့်မယ်။

Finalizing

ဆက်လက်ပြီး စောစောကရေးလိုက်တဲ့ Post အသစ်၊ Comment အသစ်ထည့်လို့ရတဲ့ API Feature တွေကို သုံးပြီး Client App ဘက်မှာလည်း ဒီလုပ်ဆောင်ချက်တွေ ထည့်ကြပါမယ်။ အရင်ဆုံး fetcher.js မှာ ဒီလိုဖြည့်စွက်ပေးပါ။

JavaScript - src/libs/fetcher.js

```javascript
export async function postPost(content) {
  const token = getToken();
  const res = await fetch(`${api}/content/posts`, {
    method: "POST",
    body: JSON.stringify({ content }),
    headers: {
      "Content-Type": "application/json",
      Authorization: `Bearer ${token}`,
    },
```

```
  });

  if (res.ok) {
    return res.json();
  }

  throw new Error("Error: Check Network Log");
}

export async function postComment(content, postId) {
  const token = getToken();
  const res = await fetch(`${api}/content/comments`, {
    method: "POST",
    body: JSON.stringify({ content, postId }),
    headers: {
      "Content-Type": "application/json",
      Authorization: `Bearer ${token}`,
    },
  });

  if (res.ok) {
    return res.json();
  }

  throw new Error("Error: Check Network Log");
}
```

ထူးခြားချက်တွေ မပါပါဘူး။ `/content/posts` နဲ့ `/content/comments` Route တွေကို `POST` လုပ်ပေးတဲ့ Function တွေ။ ပြီးရင် `pages/Home.jsx` မှာရှိတဲ့ `<form>` ကို အခုလိုပြင်လိုက်ပါ။

React - src/pages/Home.jsx

```
{showForm && auth && <Form add={add} />}
```

`<Form>` ကို `auth` Data ရှိမှပြဖို့ ပြောလိုက်တာပါ။ ပြီးတဲ့အခါ `useMutation` ကိုသုံးပြီး `create` Function တစ်ခု အခုလို ရေးပေးလိုက်ပါ။

React - src/pages/Home.jsx

```
const add = useMutation(async content => postPost(content), {
  onSuccess: async post => {
    await queryClient.cancelQueries("posts");
    await queryClient.setQueryData("posts", old => [post, ...old]);
    setGlobalMsg("A post added");
  }
});
```

`postPost()` Fetcher ကို `useMutation` နဲ့ခေါ်ပြီး `onSuccess` မှာ Query Data ကို Update လုပ်ပေးလိုက်တာပါ။ `postPost` ကို ထည့်ပြီး Import လုပ်ပြမထားပါဘူး။ ကိုယ့်ဘာသာ Import မှန်အောင် ထည့်လုပ်လိုက်ပါ။ စမ်းကြည့်လို့ရဖို့အတွက် `components/Form.jsx` မှာ ဒီတစ်ခုပြင်ပေးဖို့ လိုပါတယ်။

React - src/components/Form.jsx

```
// add(content);
add.mutate(content);
```

မူလက `add()` ကိုတိုက်ရိုက်ခေါ်ထားပေမယ့် အခုမရတော့ပါဘူး။ `add.mutate()` ကို ခေါ်ရမှာပါ။ ဒီထိရေးပြီးရင်တော့ Post အသစ်တွေထည့်ပြီး စမ်းကြည့်လို့ရပါပြီ။

ဆက်လက်ပြီး Comment အသစ်တွေ ထည့်လို့ရတဲ့ လုပ်ဆောင်ချက် ရဖို့အတွက် `pages/Comments.jsx` မှာ `postComment` ကို Import လုပ်ပြီး ပုံစံတူ အခုလို Mutation Function တစ်ခု ရေးပေးပါ။

React - src/pages/Comments.jsx

```
const addComment = useMutation(content => postComment(content, id), {
  onSuccess: async comment => {
    await queryClient.cancelQueries("comments");
    await queryClient.setQueryData("comments", old => {
      old.comments = [...old.comments, comment];
      return { ...old };
    });
    setGlobalMsg("A comment added");
  },
});
```

Function Name ကို `addComment` လို့ပေးထားတာပဲ ရှိပါတယ်။ ကျန်အလုပ်လုပ်ပုံတွေ ခပ်ဆင်ဆင်ပါပဲ။ ပြီးရင် Form Input နဲ့ချိတ်ပေးဖို့ `ref` တစ်ခု ထည့်လိုက်ပါ။

React - src/pages/Comments.jsx

```
const contentInput = useRef();
```

ပြီးတဲ့အခါ `<form>` ကိုအခုလိုပုံစံဖြစ်အောင် ပြင်ရေးပေးပါ။

React - src/pages/Comments.jsx

```
{auth && (
  <form
    onSubmit={e => {
      e.preventDefault();
      const content = contentInput.current.value;
      if (!content) return false;

      addComment.mutate(content);

      e.currentTarget.reset();
    }}>
```

```
    <Box
      sx={{
        display: "flex",
        flexDirection: "column",
        gap: 1,
        mt: 3,
      }}>
      <TextField
        inputRef={contentInput}
        multiline
        placeholder="Your Comment"
      />
      <Button
        type="submit"
        variant="contained">
        Reply
      </Button>
    </Box>
  </form>
)}
```

auth Data ရှိမှ Form ကိုပြခိုင်းထားပါတယ်။ auth ကို Import ကိုယ့်ဘာသာထည့်လုပ်ပေးလိုက်ပါ။ TextField မှာ ref နဲ့ ချိတ်ပေးထားပါတယ်။ onSubmit မှာတော့ စောစောကရေးလိုက်တဲ့ addComment ရဲ့ mutate ကိုခေါ်ပြီး Comment ထည့်ပေးလိုက်မှာ ဖြစ်ပါတယ်။

ဒီထိစုံပြီဆိုရင် Comment အသစ်တွေလည်း ထည့်လို့ရသွားပါပြီ။ နဂိုအလုပ်လုပ်နေတဲ့ Delete လုပ်ဆောင်ချက်တွေကတော့ အလုပ်လုပ်တော့မှာ မဟုတ်ပါဘူး။ token ထည့်ပေးမထားလို့ပါ။ ဒါကို လည်း Exercise အနေနဲ့ ပုံမှန်အတိုင်း အလုပ်ပြန်လုပ်သွားအောင် ကိုယ့်ဘာသာ ဖြည့်စွက်ကြည့်လိုက်ပါ။ လိုအပ်ရင်တော့ စာအုပ်နဲ့အတူ တွဲထည့်ပေးထားတဲ့ Source Code တွေထဲမှာ ကြည့်လိုက်ပါ။

Authentication နဲ့ Authorization ရသွားပြီဖြစ်သလို Content နဲ့ပက်သက်တဲ့ အခြေခံ Feature တွေ လည်း စုံသလောက် ဖြစ်သွားပါပြီ။

အခန်း (၁၆) - Project - Social Media Features

ဒီအခန်းမှာတော့ လုပ်လက်စ ပရောဂျက်မှာ Social Media Features တွေ ထပ်ထည့်ကြပါမယ်။ API ဘက်မှာလည်း ထပ်ဖြည့်စရာတွေ ရှိမှာဖြစ်သလို Client App ဘက်မှာလည်း ထပ်ဖြည့်စရာတွေ ထည့်ကြပါမယ်။ ရေးရမယ့်ကုဒ်တွေများပြီး Front-end, Back-end နေရာစုံမှာရေးရမှာဖြစ်လို့ အဓိကကျတဲ့ ကုဒ်တွေကို ဦးစားပေး ဖော်ပြသွားမှာဖြစ်ပြီး Import လုပ်ဖို့တို့၊ Style တွေ ညှိဖို့တို့လို အသေးအဖွဲ့တွေကိုတော့ အကုန်ထည့်ပြောနိုင်မှာ မဟုတ်တော့ပါဘူး။ တချို့ လိုအပ်တာတွေကို ကိုယ်ဘာသာ ကြိုးစားပြီး ဖြည့်စွက်ပေးရပါလိမ့်မယ်။ ရေးပြီးသားကို ပြင်ရတဲ့အခါမှာလည်း ကုဒ်အဟောင်းအသစ် အကုန်အတွဲလိုက် ပြနိုင်မှာ မဟုတ်ဘဲ အသစ်ကိုပဲ ပြနိုင်မှာဖြစ်ပါတယ်။ ကိုယ့်ဘာသာ ကြိုးစားပေါင်းစပ်ပေးရမှာ ဖြစ်ပါတယ်။

Like တို့ Follow တို့လိုလုပ်ဆောင်ချက်အသစ်တွေ ထပ်ဖြည့်မှာပါ။ ဒါကြောင့် လိုအပ်တဲ့ Data တွေသိမ်းဖို့အတွက် Schema ကနေ စပြင်ရပါမယ်။ API ဘက်ခြမ်း၊ `prisma.schema` မှာ အခုလို ထပ်ဖြည့်ပေးပါ။

Prisma - prisma/schema.prisma

```
model PostLike {
  id       Int    @id @default(autoincrement())
  post     Post   @relation(fields: [postId], references: [id],
                  onDelete: Cascade)
  postId   Int
  user     User   @relation(fields: [userId], references: [id],
                  onDelete: Cascade)
  userId   Int
```

```
  created DateTime @default(now())
}

model CommentLike {
  id        Int      @id @default(autoincrement())
  comment   Comment  @relation(fields: [commentId], references: [id],
                     onDelete: Cascade)
  commentId Int
  user      User     @relation(fields: [userId], references: [id],
                     onDelete: Cascade)
  userId    Int
  created   DateTime @default(now())
}

model Follow {
  id          Int      @id @default(autoincrement())
  follower    User     @relation("followers", fields: [followerId],
                       references: [id], onDelete: Cascade)
  followerId  Int
  following   User     @relation("following", fields: [followingId],
                       references: [id], onDelete: Cascade)
  followingId Int
  created     DateTime @default(now())
}
```

`PostLike`, `CommentLike` နဲ့ `Follow` ဆိုတဲ့ Model သုံးခုထပ်တိုးလိုက်တာပါ။ `PostLike` အတွက် Post နဲ့ User ကို Relationship ချိတ်ပေးထားပါတယ်။ `CommentLike` အတွက် Comment နဲ့ User ကို Relationship ချိတ်ပေးထားပါတယ်။ ဒီလိုချိတ်တဲ့အခါ `onDelete: Cascade` Attribute ထည့်ပေးထားပါတယ်။ ဒါကြောင့် သက်ဆိုင်ရာ User တို့ Post တို့ကိုဖျက်လိုက်တဲ့အခါ သူတို့နဲ့ချိတ်ထားတဲ့ Like Data တွေ အလိုအလျှောက် ပျက်သွားမှာ ဖြစ်ပါတယ်။

နောက်ထပ် ထူးခြားချက်ကတော့ `Follow` Model မှာ `follower` နဲ့ `following` နှစ်ခုလုံးက User Model ကို Relationship ချိတ်ထားပါတယ်။ ဒါကြောင့် ဒီနှစ်ခုကို ကွဲပြားသွားအောင် Relationship Name

မတူအောင် ပေးဖို့လိုပါတယ်။ ဒါကြောင့် `@relation` Attribute မှာ `followers` နဲ့ `following` ဆိုပြီး အမည်မတူအောင် ခွဲပေးထားတာကို သတိပြုပါ။ User Model ကိုလည်း အခုလို ပြင်ပေးဖို့လိုပါမယ်။

Prisma - prisma/schema.prisma

```
model User {
  id           Int           @id @default(autoincrement())
  name         String
  username     String        @unique
  bio          String?
  password     String
  posts        Post[]
  comments     Comment[]
  postLikes    PostLike[]
  commentLikes CommentLike[]
  followers    Follow[]      @relation("followers")
  following    Follow[]      @relation("following")
  created      DateTime      @default(now())
}
```

`postLikes`, `commentLikes`, `followers`, `following` ဆိုပြီး Field (၄) ခုထပ်တိုးသွားတာပါ။ အဲ့ဒီလိုတိုးတဲ့အခါ `followers` နဲ့ `following` တို့အတွက် `@relation` Name တွေပေးထားတာကို သတိပြုပါ။ ဒီတော့မှ Follow Model က User Model တစ်ခုထဲကို နှစ်ကြိမ်ချိတ်တဲ့အခါ ဘယ် Field နဲ့ ဘယ် Field ကိုချိတ်ပေးရမလဲ သိနိုင်မှာဖြစ်ပါတယ်။

တကယ်တော့ အခုလို Feature မျိုးတွေနဲ့ပိုသင့်တော်တာက NoSQL Database နည်းပညာတွေ၊ Graph Database နည်းပညာတွေပါ။ Like Data အပါအဝင် ဖွဲ့စည်းပုံမတူကွဲပြားပေမယ့် တွဲသိမ်းဖို့လိုတဲ့ Data တွေကို Embedded Data အနေနဲ့ သက်ဆိုင်ရာ Post နဲ့အတူ တွဲသိမ်းသွားလို့ရနိုင်ပါတယ်။ Table တွေ သပ်သပ်ခွဲနေစရာ မလိုပါဘူး။ ဒီလို Data အမျိုးအစားအတွက် ပိုသဘာဝကျတဲ့နည်းလမ်း ဖြစ်နိုင်ပါတယ်။

အခုသုံးထားတဲ့ SQLite အပါအဝင် တခြား RDBMS System တွေမှာလည်း တွဲသိမ်းချင်တဲ့ Data တွေကို JSON Data အနေနဲ့ တွဲသိမ်းလို့ရနိုင်ပါတယ်။ ဒါပေမယ့် နောင်လိုအပ်လို့ အဲ့ဒီ Data တွေကို စီမံရတဲ့အခါ

အဆင်ပြေလွယ်ကူမှုချင်းကတော့ ကွာသွားပါလိမ့်မယ်။ SQLite မှာ JSON Support သိပ်မပြည့်စုံတာကြောင့် ပါသလို၊ နောင်ပြန်စီမံရတာ အဆင်မပြေမှာလည်း စိုးတဲ့အတွက် ဒီနည်းကိုမသုံးဘဲ Table တွေပဲ သပ်သပ်ခွဲသိမ်းထားပါတယ်။

ဆက်လက်ပြီး စမ်းစရာ Like Data တချို့ နမူနာရအောင် `prisma/seeds/LikeSeeder.js` ဖိုင်နဲ့ ဒီကုဒ်ကိုရေးပေးပါ။

JavaScript - prisma/seeds/LikeSeeder.js

```
const { PrismaClient } = require("@prisma/client");
const { faker } = require("@faker-js/faker");

const prisma = new PrismaClient();

async function LikeSeeder() {
  console.log("Post like seeding started...");
  for (let i = 0; i < 5; i++) {
    await prisma.postLike.create({
      data: {
        postId: 20,
        userId: faker.number.int({ min: 1, max: 10 }),
      },
    });
  }

  console.log("Post like seeding done.");
}

module.exports = { LikeSeeder };
```

`postLike` အတွက် နမူနာ Like Data (၅) ခုထည့်ပေးလိုက်တာပါ။ `postId: 20` နဲ့ အသေပေးထားပါတယ်။ ဒါကြောင့် Like Data တွေအားလုံးက Post 20 တစ်ခုထဲအတွက်ပဲ ဝင်သွားမှာပါ။ နမူနာ User က (၁၀) ယောက်ပဲရှိလို့ `userId` ကို Random 1 to 10 ထည့်တဲ့အခါ `id` တူ နှစ်ခါသုံးခါထည့်မိတာမျိုးတော့

ဖြစ်နိုင်ပါတယ်။ အဲ့ဒီပြဿနာကိုတော့ ထည့်ဖြေရှင်းမနေတော့ပါဘူး။ ဒီတိုင်းပဲ စမ်းကြည့်ကြပါမယ်။

ဒီဖိုင်ကို prisma/seeds/main.js မှာ ချိတ်ပေးပါ။

JavaScript - prisma/seeds/main.js

```javascript
// other imports
const { LikeSeeder } = require("./LikeSeeder");

async function main() {
  try {
    ...
    await LikeSeeder();
  } catch (e) {
    ...
  } finally {
    ...
  }
}

main();
```

ဒါဆိုရင်လိုအပ်တဲ့ Schema နဲ့ Seed Data စုံသွားပါပြီ။ Model အသစ်တွေ ထပ်တိုးထားလို့ Schema ကို အပြည့်အစုံပြန် Run ပေးဖို့လိုပါတယ်။ ဒီလို Run လိုပါ။

```
npx prisma migrate dev --name=social
npx prisma db seed
```

Like / Unlike - API

ဆက်လက်ပြီး routers/content.js မှာ Post တွေ Like/Unlike လုပ်ပေးနိုင်တဲ့လုပ်ဆောင်ချက်ကို အခုလို ရေးပေးပါ။

JavaScript - API - routers/content.js

```
router.post("/like/posts/:id", auth, async (req, res) => {
  const { id } = req.params;
  const user = res.locals.user;

  const like = await prisma.postLike.create({
    data: {
      postId: Number(id),
      userId: Number(user.id),
    },
  });

  res.json({ like });
});

router.delete("/unlike/posts/:id", auth, async (req, res) => {
  const { id } = req.params;
  const user = res.locals.user;

  await prisma.postLike.deleteMany({
    where: {
      postId: Number(id),
      userId: Number(user.id),
    },
  });

  res.json({ msg: `Unlike post ${id}` });
});
```

Like လုပ်တယ်ဆိုတာ Like Table ထဲကို Entry တစ်ခုထည့်လိုက်တာပါပဲ။ ဒါကြောင့် POST ကိုပဲ သုံးထားပါတယ်။ တခြားထူးခြားချက်တော့ မရှိပါဘူး။ Unlike လုပ်တယ်ဆိုတာလည်း Like Table ထဲက ပြန်ဖျက်လိုက်တာပါ။ ဒါကြောင့် DELETE ကိုသုံးထားပါတယ်။ ထူခြားချက်အနေနဲ့ `prisma` ရဲ့ `delete` ကို သုံးရင် ရပေမယ့် `deleteMany` ကိုသုံးထားပါတယ်။ စမ်းရင်းနဲ့ မတော်တဆ နှစ်ကြိမ်သုံးကြိမ် ဝင်နေတဲ့ Like Data တွေရှိရင် တစ်ခါထဲ အကုန်ပြန်ဖျက်သွားစေချင်လို့ပါ။

Comment Like နဲ့ Unlike အတွက်လည်း အခုလို ပုံစံတူ ထပ်ရေးပေးပါ။

JavaScript - API - routers/content.js

```
router.post("/like/comments/:id", auth, async (req, res) => {
  const { id } = req.params;
  const user = res.locals.user;

  const like = await prisma.commentLike.create({
    data: {
      commentId: Number(id),
      userId: Number(user.id),
    },
  });

  res.json({ like });
});

router.delete("/unlike/comments/:id", auth, async (req, res) => {
  const { id } = req.params;
  const user = res.locals.user;

  await prisma.commentLike.deleteMany({
    where: {
      commentId: Number(id),
      userId: Number(user.id),
    },
  });
```

```
  res.json({ msg: `Unlike comment ${id}` });
});
```

ပြီးတဲ့အခါ အဲ့ဒီ Like Data တွေကို လိုတဲ့အခါ ပြန်ထုတ်ယူပေးနိုင်အောင် ထပ်ရေးပေးရပါမယ်။

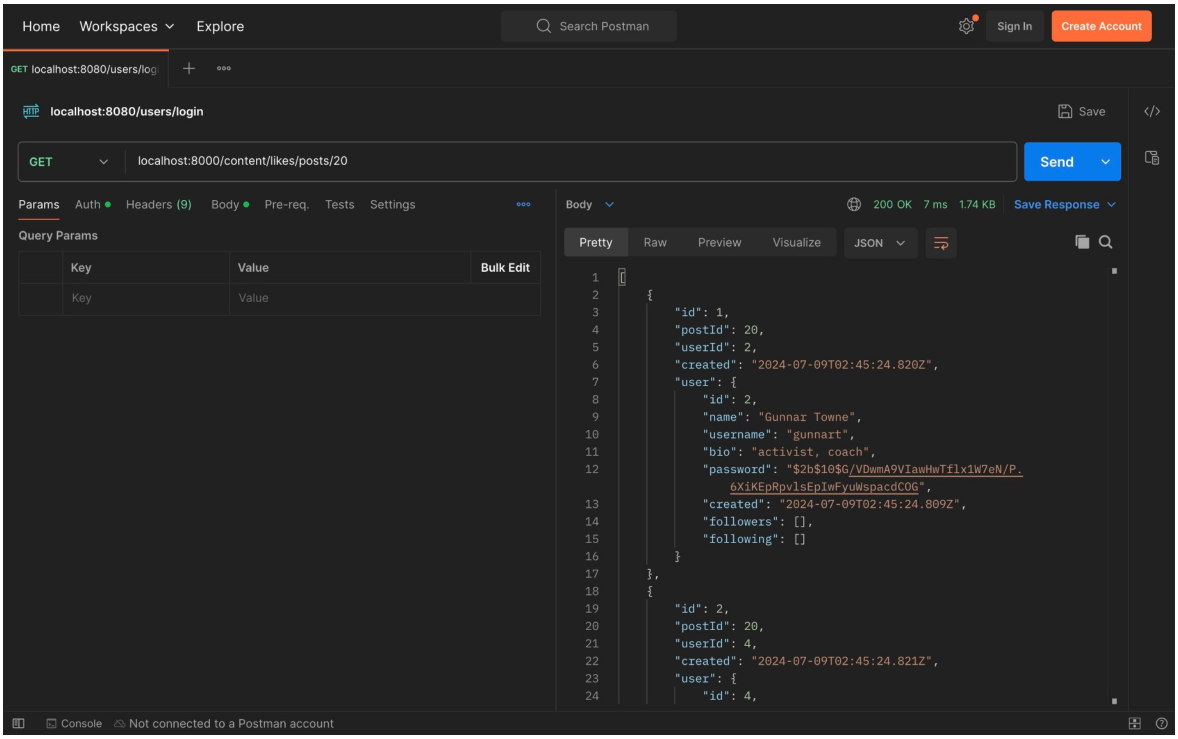
JavaScript - API - routers/content.js

```
router.get("/likes/posts/:id", async (req, res) => {
  const { id } = req.params;

  const data = await prisma.postLike.findMany({
    where: {
      postId: Number(id),
    },
    include: {
      user: {
        include: {
          followers: true,
          following: true,
        },
      },
    },
  });

  res.json(data);
});

router.get("/likes/comments/:id", async (req, res) => {
  const { id } = req.params;

  const data = await prisma.commentLike.findMany({
    where: {
      commentId: Number(id),
```

```
    },
    include: {
      user: {
        include: {
          followers: true,
          following: true,
        },
      },
    },
  });

  res.json(data);
});
```

Relationship နှစ်ဆင့်ပါဝင်ပါတယ်။ Like Data နဲ့အတူ သက်ဆိုင်ရာ User ကို တွဲယူထားပြီး အဲ့ဒီ User မှာ ရှိနေတဲ့ Follower တွေနဲ့ Following တွေကိုပါ တွဲယူထားပါတယ်။ Follow / Unfollow ကုဒ်တွေ မရေးရသေးပေမယ့် Relationship ရှိနေပြီးသားမို့လို့ အခုလိုထုတ်ယူလို့ရပါတယ်။ Data တွေတော့ မရှိသေးလို့ ပါဦးမှာမဟုတ်ပါဘူး။ နောင်ရှိတဲ့အခါမှ ပြန်လာပြင် ရေးနေစရာမလိုအောင် အခုကတည်းက ကြို ထည့်ထားလိုက်တာပါ။

Postman နဲ့ ထပ်တိုးလိုက်တဲ့ Feature တွေကို သေချာအောင်၊ စုံအောင် အရင်စမ်းရပါတယ်။



နမူနာမှာ Post 20 အတွက် Like Data တွေယူပြီး စမ်းပြထားပါတယ်။ Post 20 ကို User 2 က Like လုပ်ထားပြီး User 2 ရဲ့ အချက်အလက်အပြည့်အစုံလည်း `user` အနေနဲ့ တွဲပြီးပါလာပါတယ်။ `followers` နဲ့ `following` မရှိသေးလို့ Array အလွတ်တွေ ပါလာတာကို တွေ့ရပါလိမ့်မယ်။

User Data တွေမှာ Password တွေပါပါနေတဲ့ ပြဿနာတစ်ခုတော့ ကျန်ပါသေးတယ်။ အဲ့ဒါကို API ဘက်က Response မပြန်ခင် ဖယ်ထုတ်ပြီးမှ ပြန်ရင်လည်း ရနိုင်ပေမယ့် ထည့်လုပ်မထားပါဘူး။ ဒီလို ကိုယ့်ဘာသာ လုပ်မယ့်အစား Prisma ရဲ့ `omitApi` ကို သုံးရမှာပါ။ လောလောဆယ် Experimental ပဲရှိသေးလို့ ထည့်မသုံးထားပါဘူး။ လေ့လာကြည့်ပြီး လက်တွေ့လိုအပ်တဲ့အခါမှပဲ ထည့်သုံးလိုက်ပါ။

https://www.prisma.io/docs/orm/prisma-client/queries/excluding-fields

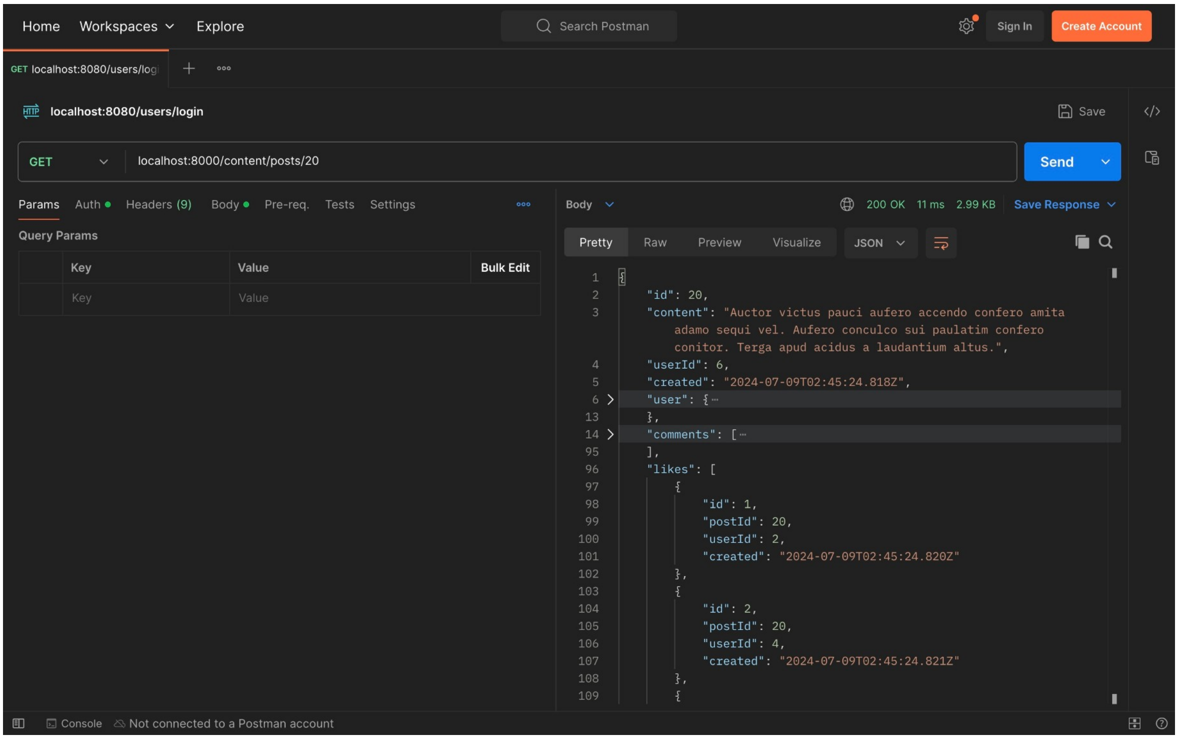
ဆက်လက်ပြီး Post Data တွေ Comment Data တွေ ထုတ်ယူတဲ့အခါ သက်ဆိုင်တဲ့ Like Data တွေ တွဲပါလာအောင်ထုတ်ယူဖို့လိုပါတယ်။ ဒါကြောင့် `GET /posts` နဲ့ `GET /posts/:id` တို့ကို အခုလိုပြင်ပေးပါ။

JavaScript - API - routers/content.js

```
router.get("/posts", async (req, res) => {
  try {
    const data = await prisma.post.findMany({
      include: {
        user: true,
        comments: true,
        likes: true,
      },
      orderBy: { id: "desc" },
      take: 20,
    });

    res.json(data);
  } catch (e) {
    res.status(500).json({ error: e });
  }
});

router.get("/posts/:id", async (req, res) => {
  const { id } = req.params;

  try {
    const data = await prisma.post.findUnique({
      where: { id: Number(id) },
      include: {
        user: true,
        comments: {
          include: {
            user: true,
            likes: true,
          },
        },
        likes: true,
      },
```

```
    });

    res.json(data);
  } catch (e) {
    res.status(500).json({ error: e });
  }
});
```

အသစ်ထပ်ရေးရမှာ မဟုတ်ပါဘူး။ ရေးပြီးသားကုဒ်မှာပဲ Relationship Include အတွက် `likes` တွေ ထပ်တိုးလိုက်တာပါ။ Postman နဲ့ Post 20 ကို ထုတ်ယူကြည့်ရင် အခုလို တွေ့ရပါလိမ့်မယ်။



Post နဲ့အတူ Like Data တွေပါ တစ်ခါထဲ တွဲပါသွားတာပါ။ စမ်းကြည့်လို့ API ဘက်က အားလုံးမှန်ပြီဆို ရင်တော့ Client App ဘက်မှာလည်း Like/Unlike Feature ထည့်လို့ရပါပြီ။

Like / Unlike - Client

အရင်ဆုံး `fetcher.js` မှာ အခုလိုထည့်ပေးပါ။

JavaScript - src/libs/fetcher.js

```javascript
export async function postPostLike(id) {
  const token = getToken();
  const res = await fetch(`${api}/content/like/posts/${id}`, {
    method: "POST",
    headers: {
      Authorization: `Bearer ${token}`,
    },
  });

  return res.json();
}

export async function postCommentLike(id) {
  const token = getToken();
  const res = await fetch(`${api}/content/like/comments/${id}`, {
    method: "POST",
    headers: {
      Authorization: `Bearer ${token}`,
    },
  });

  return res.json();
}

export async function deletePostLike(id) {
  const token = getToken();
  const res = await fetch(`${api}/content/unlike/posts/${id}`, {
    method: "DELETE",
    headers: {
      Authorization: `Bearer ${token}`,
```

```
    },
  });

  return res.json();
}

export async function deleteCommentLike(id) {
  const token = getToken();
  const res = await fetch(`${api}/content/unlike/comments/${id}`, {
    method: "DELETE",
    headers: {
      Authorization: `Bearer ${token}`,
    },
  });

  return res.json();
}

export async function fetchPostLikes(id) {
  const res = await fetch(`${api}/content/likes/posts/${id}`);
  return res.json();
}

export async function fetchCommentLikes(id) {
  const res = await fetch(`${api}/content/likes/comments/${id}`);
  return res.json();
}
```

တစ်ခါထဲ အကုန်စုံသွားအောင်ပေးထားလို့ ကုဒ်တွေများနေပေမယ့် အထူးအဆန်းတော့ တစ်ခုမှ မပါပါဘူး။ စောစောက ရေးထားတဲ့ API Features တွေကို Fetch လုပ်ပေးတဲ့ ပုံစံဆင်တူ Function (၆) ခု ဖြစ်ပါတယ်။ ပြီးတဲ့အခါ `components/LikeButton.js` အမည်နဲ့ Component ဖိုင်သစ်တစ်ခု တည်ဆောက်ပြီး အခုလိုရေးပေးပါ။

React - src/components/LikeButton.jsx

```
import {
    IconButton,
    ButtonGroup,
    Button
} from "@mui/material";

import {
    Favorite as LikedIcon,
    FavoriteBorder as LikeIcon
} from "@mui/icons-material";

import { useNavigate } from "react-router-dom";
import { useApp, queryClient } from "../ThemedApp";

import { useMutation } from "react-query";

import {
    postPostLike,
    deletePostLike,
    postCommentLike,
    deleteCommentLike,
} from "../libs/fetcher";

export default function LikeButton({ item, comment }) {
  const navigate = useNavigate();
  const { auth } = useApp();

  function isLiked() {
    if (!auth) return false;
    if (!item.likes) return false;

    return item.likes.find(like => like.userId == auth.id);
  }
```

```
const likePost = useMutation(id => postPostLike(id), {
  onSuccess: () => {
    queryClient.refetchQueries("posts");
    queryClient.refetchQueries("comments");
  },
});

const likeComment = useMutation(id => postCommentLike(id), {
  onSuccess: () => {
    queryClient.refetchQueries("comments");
  },
});

const unlikePost = useMutation(id => deletePostLike(id), {
  onSuccess: () => {
    queryClient.refetchQueries("posts");
    queryClient.refetchQueries("comments");
  },
});

const unlikeComment = useMutation(id => deleteCommentLike(id), {
  onSuccess: () => {
    queryClient.refetchQueries("comments");
  },
});

return (
  <ButtonGroup>
    {isLiked() ? (
      <IconButton
        size="small"
        onClick={e => {
          comment
            ? unlikeComment.mutate(item.id)
            : unlikePost.mutate(item.id);
```

```
          e.stopPropagation();
        }}>
        <LikedIcon
          fontSize="small"
          color="error"
        />
      </IconButton>
    ) : (
      <IconButton
        size="small"
        onClick={e => {
          comment
            ? likeComment.mutate(item.id)
            : likePost.mutate(item.id);

          e.stopPropagation();
        }}>
        <LikeIcon
          fontSize="small"
          color="error"
        />
      </IconButton>
    )}
    <Button
      onClick={e => {
        if (comment) {
          navigate(`/likes/${item.id}/comment`);
        } else {
          navigate(`/likes/${item.id}/post`);
        }

        e.stopPropagation();
      }}
      sx={{ color: "text.fade" }}
```

```
        variant="text"
        size="small">
        {item.likes ? item.likes.length : 0}
      </Button>
    </ButtonGroup>
  );
}
```

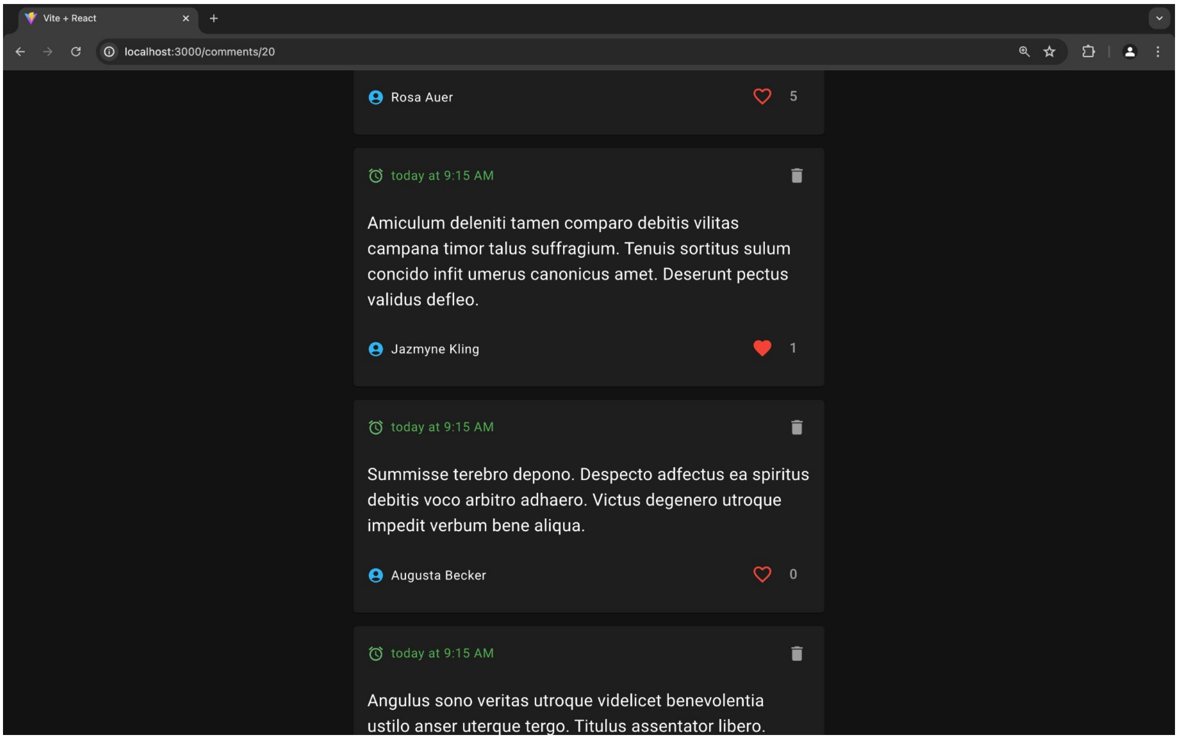
ဒီ Feature အတွက် အရေးအကြီးဆုံးနဲ့ အဓိကအကျဆုံး ကုဒ်ပါပဲ။ ထူးခြားချက်လေးတွေ လိုက်ကြည့်ကြပါမယ်။ ပထမဆုံး Props အနေနဲ့ `item` နဲ့ `comment` ကို တောင်းထားတာ သတိပြုပါ။ Post တွေ ပြဖို့အတွက်ရော၊ Comment တွေ ပြဖို့အတွက်ပါ `Item.jsx` ကိုပဲ သုံးထားပါတယ်။ ဒါကြောင့် လက်ရှိအလုပ်လုပ်နေတာ Post အတွက်လား၊ Comment အတွက်လား သိဖို့လိုပါတယ်။ ဒါကြောင့် `comment` Prop ကိုထည့်တောင်းထားတာပါ။ `comment` ဆိုရင် Comment Like, Comment Unlike လုပ်ဆောင်ချက်တွေကို လုပ်ပေးရမှာဖြစ်ပြီး `comment` မဟုတ်ရင် Post Like, Post Unlike လုပ်ဆောင်ချက်တွေကို လုပ်ပေးရမှာပါ။

Like Button ကို ပြတဲ့အခါ Like လုပ်ထားရင်တစ်မျိုး၊ မလုပ်ထားရင်တစ်မျိုး၊ နှစ်မျိုးပြချင်ပါတယ်။ ဒါကြောင့် `isLiked()` အမည်နဲ့ Function တစ်ခုထည့်ရေးပါတယ်။ ဒီ Function က လက်ရှိ Login User ကို Like List ထဲမှာ ပါမပါ စစ်ပေးတာပါ။ ပါနေရင် `true` ပြန်ပေးပြီး၊ မပါရင် (သို့မဟုတ်) Login မလုပ်ထားရင် တော့ `false` ပြန်ပေးမှာပါ။ Like List မရှိရင်လည်း `false` ပြန်ပေးပါတယ်။ အခုမှ အသစ်ထည့်လိုက်တဲ့ Post တွေ Comment တွေမှာ Like ဆိုတာ ရှိကိုရှိဦးမှာ မဟုတ်ပါဘူး။ အဲ့ဒီလိုအခြေအနေမှာ ထည့်အလုပ်မလုပ်စေချင်လို့ပါ။

ပြီးတဲ့အခါ `useMutation` နဲ့ Post Like/Unlike, Comment Like/Unlike အတွက် Fetcher တွေကို ခေါ်သုံးထားပါတယ်။ ခေါ်သုံးတာက အသစ်အဆန်းမဟုတ်ပါဘူး။ `onSuccess` မှာ `queryClient` ကို သုံးပြီး `refetch` လုပ်ထားတာကိုသာ သတိပြုပါ။ ဒါကြောင့် Like / Unlike လုပ်လိုက်တိုင်း API Refetch ပြန်လုပ်ပြီး Component ကို Update လုပ်ပြေးမှာပါ။ ပိုဖြစ်သင့်တာကတော့ Refetch လုပ်မနေဘဲ `queryData` ကို Client-Side မှာပဲ Update လုပ်ပြီးပြသင့်ပါတယ်။ ဒါဆိုရင် တစ်ခုခုလုပ်လိုက်တိုင်း Refetch ပြန်လုပ်နေစရာမလိုတော့ပါဘူး။ ဒါပေမယ့် Client-Side မှာ Update လုပ်ရတာ နည်းနည်း

အလုပ်ရှုပ်လို့ ဒီနေရာမှာတော့ အလွယ်လုပ်တဲ့အနေနဲ့ `refetch` ပြန်လုပ်ထားခြင်း ဖြစ်ပါတယ်။

UI ပိုင်းမှာတော့ သိပ်ထူးခြားတာ မပါပါဘူး။ MUI ရဲ့ ButtonGroup ကိုသုံးပြီး LikeIcon ကို ပုံစံနှစ်မျိုးပြ ထားပါတယ်။ ဒီလိုပါ။



ကိုယ် Like လုပ်ထားတဲ့ Icon တွေက ပိုထင်ရှားပြီး မြင်သာနေအောင် ပြပေးထားတာပါ။ Like Icon နဲ့ အတူ Like Count ကိုလည်း တွဲပြထားပါတယ်။ Like Icon နှိပ်ပြီး Like / Unlike လုပ်နိုင်ပါတယ်။ Post မှာရော၊ Comment မှာရော အကုန်ရပါတယ်။ Like Count ကို နှိပ်လိုက်ရင်တော့ Like User စာရင်းကို ပြ ပေးနိုင်တဲ့ Page ကို သွားခိုင်းထားပါတယ်။

Like User စာရင်းကိစ္စ မလုပ်ခင်၊ Comment Button လေး အရင်ထည့်ချင်ပါသေးတယ်။ ဒါကြောင့် `components/CommentButton.jsx` မှာ အခုလိုရေးပေးပါ။

React - src/components/CommentButton.jsx

```
import {
    IconButton,
    ButtonGroup,
    Button
} from "@mui/material";

import {
    ChatBubbleOutline as CommentIcon
} from "@mui/icons-material";

export default function Item({ item, comment }) {
  return (
    <>
      {!comment && (
        <ButtonGroup sx={{ ml: 3 }}>
          <IconButton size="small">
            <CommentIcon
              fontSize="small"
              color="info"
            />
          </IconButton>
          <Button
            sx={{ color: "text.fade" }}
            variant="text"
            size="small">
            {item.comments.length}
          </Button>
        </ButtonGroup>
      )}
    </>
  );
}
```

ဒီကုဒ်မှာတော့ ရှုပ်ထွေးတဲ့လုပ်ဆောင်ချက် ဘာမှမပါဘဲ Comment Icon နဲ့အတူ Comment Count ကိုပြထားတာပါ။ ထူးခြားချက်တစ်ခုကတော့ `comment` ဖြစ်နေရင် ထည့်မပြဖို့ စစ်ထားခြင်းဖြစ်ပါတယ်။ ဒါကြောင့် Comment Button ကို Post တွေမှာပဲပြပြီး Comment တွေမှာ ထည့်ပြမှာ မဟုတ်ပါဘူး။

ပြီးတဲ့အခါ Like Button နဲ့ Comment Button တို့ကို `Item.jsx` မှာ အခုလို Import လုပ်လိုက်ပါ။

React - src/components/Item.jsx

```
import LikeButton from "./LikeButton";
import CommentButton from "./CommentButton";
```

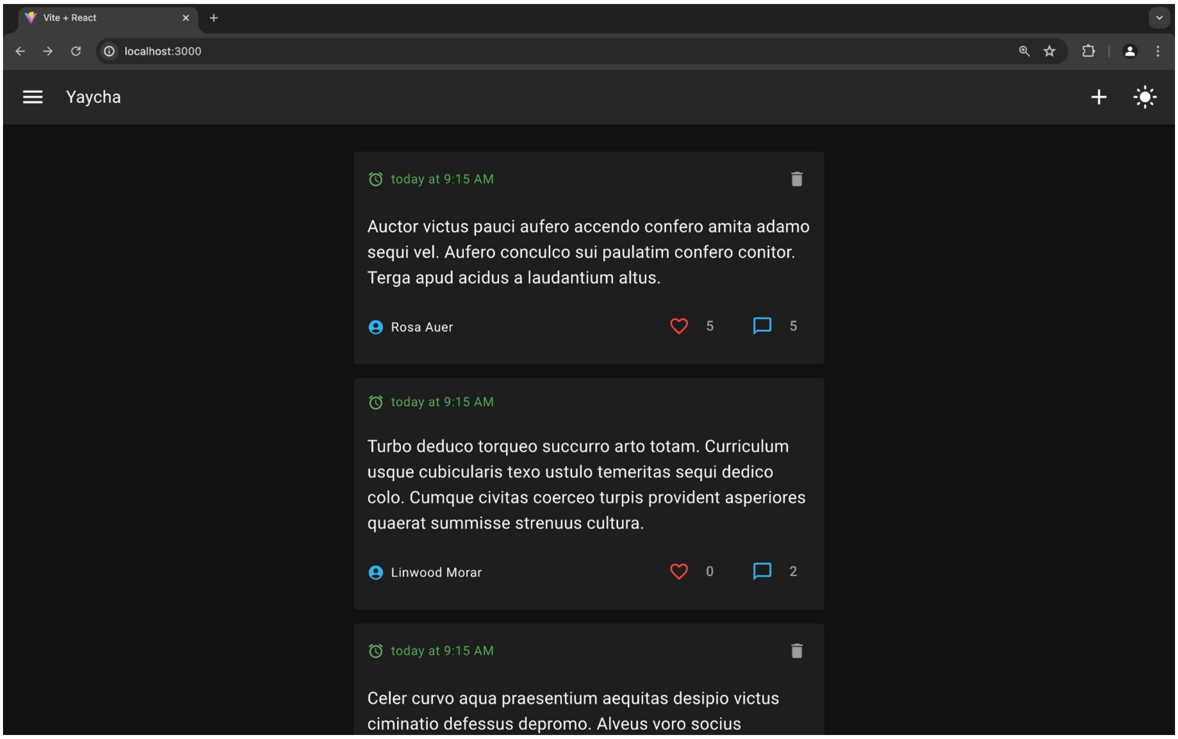
ဆက်လက်ပြီး လက်ရှိ User Name ပြထားတဲ့ Box ထဲမှာ အခုလို ထည့်ပေးလိုက်ပါ။

React - src/components/Item.jsx

```
<Box
  sx={{
    display: "flex",
    flexDirection: "row",
    alignItems: "center",
    justifyContent: "space-between",
  }}>
  <Box
    onClick={e => {
      navigate(`/profile/${item.user.id}`);
      e.stopPropagation();
    }}
    sx={{
      display: "flex",
      flexDirection: "row",
      alignItems: "center",
      gap: 1,
    }}>
```

```
    <UserIcon
      fontSize="12"
      color="info"
    />
    <Typography variant="caption">
      {item.user.name}
    </Typography>
  </Box>
  <Box>
    <LikeButton item={item} comment={comment} />
    <CommentButton item={item} comment={comment} />
  </Box>
</Box>
```

User Name နဲ့အတူ Like Button တွေ Comment Button တွေကို တွဲပြသွားတာ တွေ့ရမှာ ဖြစ်ပါတယ်။



နောက်တစ်ဆင့်အနေနဲ့ Like User List ကြည့်လို့ရအောင် ဆက်လုပ်ကြပါမယ်။ အရင်ဆုံး `ThemedApp.jsx` က Route မှာ အခုလိုပြင်ပေးပါ။

React - src/ThemedApp.jsx

```
const router = createBrowserRouter([
  {
    path: "/",
    element: <Template />,
    children: [
      ...
      {
        path: "/likes/:id/:type",
        element: <Likes />,
      },
    ],
  },
]);
```

Like Page ကိုအရင်ကတည်းက ကြိုချိတ်ပေးထားပြီးသားပါ။ Route ကိုနည်းနည်းပြင်လိုက်တာပါ။ `/likes/:id` မဟုတ်တော့ ပါဘူး။ `/likes/:id/:type` ဖြစ်သွားတာပါ။

ပြီးတဲ့အခါ `components/UserList.jsx` ကို အခုလို ပြင်ပေးပါ။

React - src/components/UserList.jsx

```
import {
  Box,
  Typography,
  List,
  ListItem,
  Avatar,
  ListItemText,
  ListItemAvatar,
  ListItemButton,
} from "@mui/material";

import { useNavigate } from "react-router-dom";

export default function UserList({ title, data }) {
  const navigate = useNavigate();

  return (
    <Box>
      <Typography
        variant="h4"
        sx={{ mb: 3, textAlign: "center" }}>
        {title}
      </Typography>
      <List>
        {data.map(item => {
          return (
            <ListItem key={item.id}>
              <ListItemButton
                onClick={() =>
                  navigate(`/profile/${item.user.id}`)
                }>
                <ListItemAvatar>
                  <Avatar />
                </ListItemAvatar>
```

```
                <ListItemText
                  primary={item.user.name}
                  secondary={item.user.bio}
                />
              </ListItemButton>
            </ListItem>
          );
        })}
      </List>
    </Box>
  );
}
```

နဂိုရေးပြီးသား UserList မှာ data ပါသွားတာပါ။ data ကို map လုပ်ပြီး User စာရင်းကို ပြပေးပါတယ်။ ပြီးတဲ့အခါ `pages/Like.jsx` မှာ အခုလိုပြင်ပေးပါ။

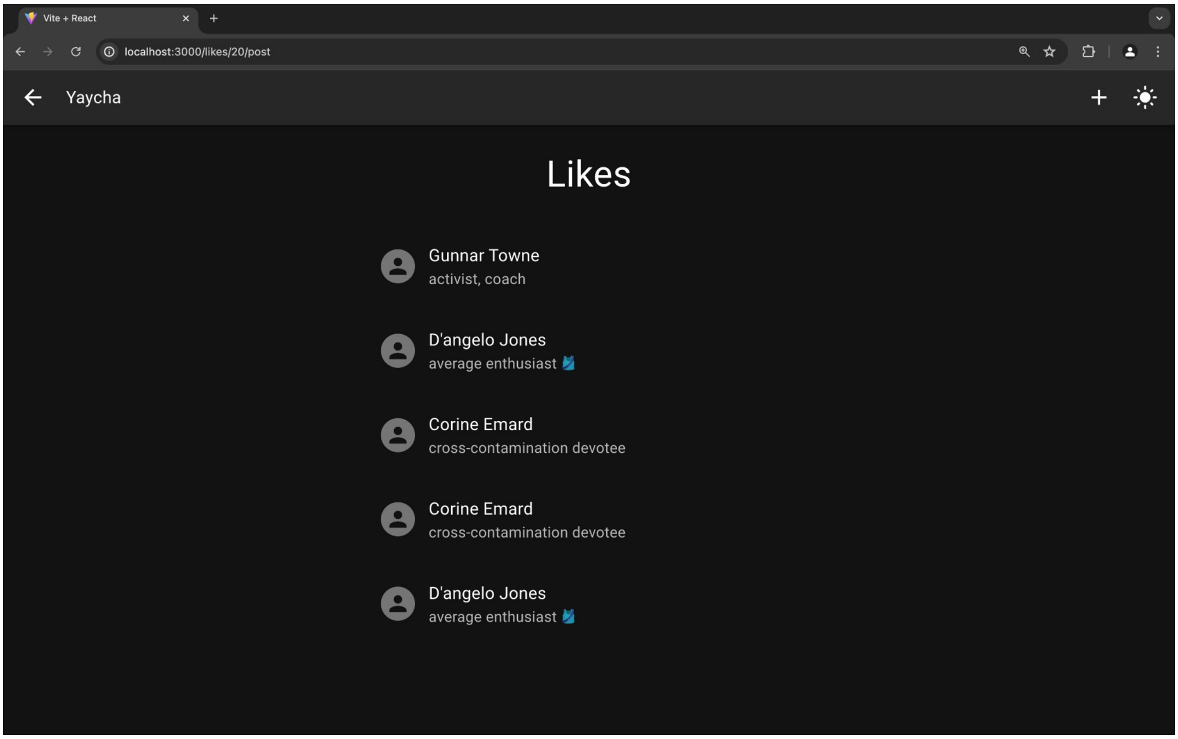
React - src/pages/Like.jsx

```
import { Box, Alert } from "@mui/material";
import { useParams } from "react-router-dom";
import { useQuery } from "react-query";

import { fetchPostLikes, fetchCommentLikes } from "../libs/fetcher";
import UserList from "../components/UserList";

export default function Likes() {
  const { id, type } = useParams();
  const { isLoading, isError, error, data } = useQuery(
    ["users", id, type],
    () => {
      if (type == "comment") {
        return fetchCommentLikes(id);
      } else {
        return fetchPostLikes(id);
```

```
      }
    }
  );

  if (isError) {
    return (
      <Box>
        <Alert severity="warning">{error.message}</Alert>
      </Box>
    );
  }

  if (isLoading) {
    return <Box sx={{ textAlign: "center" }}>Loading...</Box>;
  }

  return (
    <Box>
      <UserList
        title="Likes"
        data={data}
      />
    </Box>
  );
}
```

လိုရင်းကတော့ `useQuery` နဲ့ `type` Parameter ကိုကြည့်ပြီး `fetchCommentLikes` (သို့မဟုတ်) `fetchPostLikes` ကို ခေါ်ပြီး Like လုပ်ထားတဲ့ User List data ကို ယူလိုက်တာပါပဲ။ ထူးခြားချက် အနေနဲ့ `useQuery` ရဲ့ Query Name (Query Key) နေရာမှာ ရိုးရိုး String တစ်ခုမဟုတ်တော့ဘဲ Array တစ်ခုဖြစ်သွားပါတယ်။ `id` နဲ့ `type` Parameter တွေကိုပါ Query Key ထဲမှာ ထည့်ပေးလိုက်တာပါ။ ရည်ရွယ်ချက်ကတော့ <u>`id` သို့မဟုတ် `type` ပြောင်းခဲ့ရင် Fetch ပြန်လုပ်ပေးစေချင်လို့ဖြစ်ပါတယ်</u>။ `useEffect` ရဲ့ Dependencies Array လိုပါပဲ။ React Query ကလည်း Query Key ပြောင်းရင် ထပ် အလုပ်လုပ်ပေးပါတယ်။ Post Like တွေရော Comment Like တွေကိုပါ ဒီ Page တစ်ခုထဲနဲ့ ပြထားတာ

ဖြစ်လို့ ထပ်အလုပ်မလုပ်ရင် ပြတဲ့ Data များနေမှာစိုးလို့ပါ၊ စမ်းကြည့်လို့ရပါပြီ။ Like Count ကို နှိပ်လိုက်ရင် အခုလို Like လုပ်ထားတဲ့ User စာရင်းကို ပြပေးမှာပဲ ဖြစ်ပါတယ်။



Follow / Unfollow - API

ဆက်လက်ပြီး Follow / Unfollow အတွက် API မှာ လိုအပ်တဲ့ လုပ်ဆောင်ချက်တွေ ဆက်ထည့်ကြပါမယ်။ `routers/user.js` မှာ အခုလို ထပ်တိုးပေးရမှာပါ။

JavaScript - API - routers/user.js

```
router.post("/follow/:id", auth, async (req, res) => {
  const user = res.locals.user;
  const { id } = req.params;

  const data = await prisma.follow.create({
    data: {
      followerId: Number(user.id),
      followingId: Number(id),
```

```
    },
  });

  res.json(data);
});

router.delete("/unfollow/:id", auth, async (req, res) => {
  const user = res.locals.user;
  const { id } = req.params;

  await prisma.follow.deleteMany({
    where: {
      followerId: Number(user.id),
      followingId: Number(id),
    },
  });

  res.json({ msg: `Unfollow user ${id}` });
});
```

သူလည်း Like တုန်းကလိုပါပဲ။ Follow လုပ်တယ်ဆိုတာ Follow Table ထဲကို လိုအပ်တဲ့ Data တွေထည့်လိုက်တာ ဖြစ်ပြီး Unfollow လုပ်တယ်ဆိုတာ Follow Table ထဲကနေ ပြန်ဖျက်လိုက်တာပါပဲ။ ဆက်လက်ပြီး User Data တွေယူတဲ့အခါ သက်ဆိုင်ရာ Follower, Following Data တွေအစုံပါလာအောင် အခုလိုပြင်ပေးရပါမယ်။

JavaScript - API - routers/user.js

```
router.get("/users", async (req, res) => {
  const data = await prisma.user.findMany({
    include: {
      posts: true,
      comments: true,
      followers: true,
      following: true,
```

```
    },
    orderBy: { id: "desc" },
    take: 20,
  });

  res.json(data);
});

router.get("/users/:id", auth, async (req, res) => {
  const { id } = req.params;

  const data = await prisma.user.findUnique({
    where: { id: Number(id) },
    include: {
      posts: true,
      comments: true,
      followers: true,
      following: true,
    },
  });

  res.json(data);
});
```

အသစ်ထပ်ရေးရမှာ မဟုတ်ပါဘူး။ ရေးပြီးသားထဲမှာ Relationship Include တွေ ထပ်ထည့်ပေးရမှာပါ။ ဒါဆိုရင် API ဘက်က ပါသင့်တာ စုံသွားပါပြီ။ ရှေ့ဆက်မသွားခင် Postman နဲ့ ထပ်တိုးပြင်ဆင်လိုက်တဲ့ လုပ်ဆောင်ချက်တွေကို စုံအောင်၊ မှန်အောင် အရင် စမ်းကြည့်လိုက်ပါ။

Follow / Unfollow - Client

Client ဘက်မှာတော့ ထုံးစံအတိုင်း fetcher ကနေစကြပါမယ်။

JavaScript - src/libs/fetcher.js

```
export async function postFollow(id) {
  const token = getToken();
  const res = await fetch(`${api}/follow/${id}`, {
    method: "POST",
    headers: {
      Authorization: `Bearer ${token}`,
    },
  });

  return res.json();
}

export async function deleteFollow(id) {
  const token = getToken();
  const res = await fetch(`${api}/unfollow/${id}`, {
    method: "DELETE",
    headers: {
      Authorization: `Bearer ${token}`,
    },
  });

  return res.json();
}
```

အထူးအဆန်းမပါဘဲ စောစောက API ကို Fetch လုပ်ပေးတဲ့ Function (၂) ခုထပ်တိုးသွားတာပါ။

ဆက်လက်ပြီး `components/FollowButton.jsx` အမည်နဲ့ ဖိုင်အသစ်တစ်ခု တည်ဆောက်ပြီး အခုလို ရေးပေးပါ။

React - src/components/FollowButton.jsx

```
import { Button } from "@mui/material";
import { useMutation } from "react-query";

import { useApp, queryClient } from "../ThemedApp";
import { postFollow, deleteFollow } from "../libs/fetcher";

export default function FollowButton({ user }) {
  const { auth } = useApp();
  if (!auth) return <></>;

  function isFollowing() {
    return user.following.find(item => item.followerId == auth.id);
  }

  const follow = useMutation(
    id => {
      return postFollow(id);
    },
    {
      onSuccess: async () => {
        await queryClient.refetchQueries("users");
        await queryClient.refetchQueries("user");
        await queryClient.refetchQueries("search");
      },
    }
  );

  const unfollow = useMutation(
    id => {
      return deleteFollow(id);
    },
    {
      onSuccess: async () => {
        await queryClient.refetchQueries("users");
```

```
        await queryClient.refetchQueries("user");
        await queryClient.refetchQueries("search");
      },
    }
  );

  return auth.id === user.id ? (
    <></>
  ) : (
    <Button
      size="small"
      edge="end"
      variant={isFollowing() ? "outlined" : "contained"}
      sx={{ borderRadius: 5 }}
      onClick={e => {
        if (isFollowing()) {
          unfollow.mutate(user.id);
        } else {
          follow.mutate(user.id);
        }

        e.stopPropagation();
      }}>
      {isFollowing() ? "Following" : "Follow"}
    </Button>
  );
}
```

ဒီမှာလည်း `isFollowing()` Function တစ်ခုနဲ့ လက်ရှိ Login User ဟာ သက်ဆိုင်ရာ `user` ရဲ့ Following List ထဲမှာ ပါနေလား စစ်ထားပါတယ်။ ပြီးတော့ `user` က ကိုယ်ကိုယ်တိုင် ဖြစ်နေရင် ဘာမှမပြဖို့လည်း ထည့်ရေးထားပါတယ်။ Like Button တုန်းကလိုပဲ၊ Following List ထဲမှာ ပါပြီးသားဆိုရင် ပုံစံတစ်မျိုး၊ မပါသေးရင် ပုံစံတစ်မျိုး ပြထားပါတယ်။ ဒီ Button ကို `UserList.jsx` မှာ အခုလို Import လုပ်ပေးပါ။

React - src/components/UserList.jsx

```
import FollowButton from "./FollowButton";
```

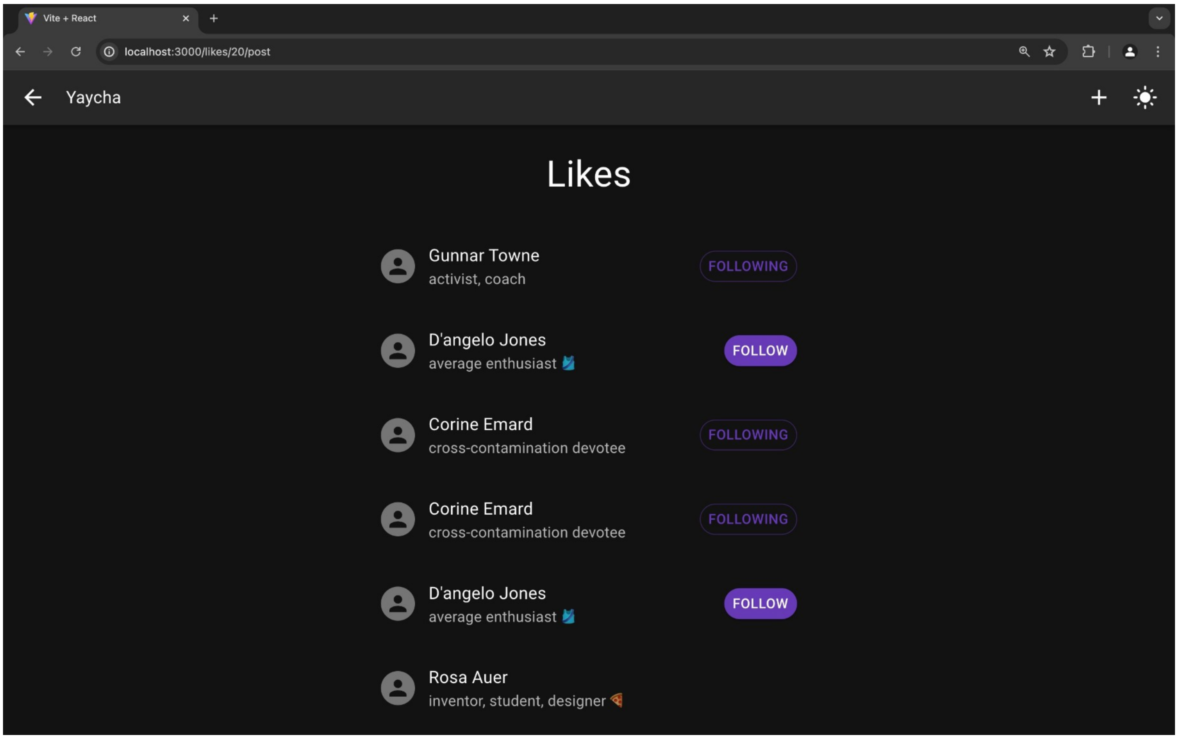
ပြီးတဲ့အခါ ListItem ထဲမှာ FollowButton ကို အခုလိုထည့်ပြပေးပါ။

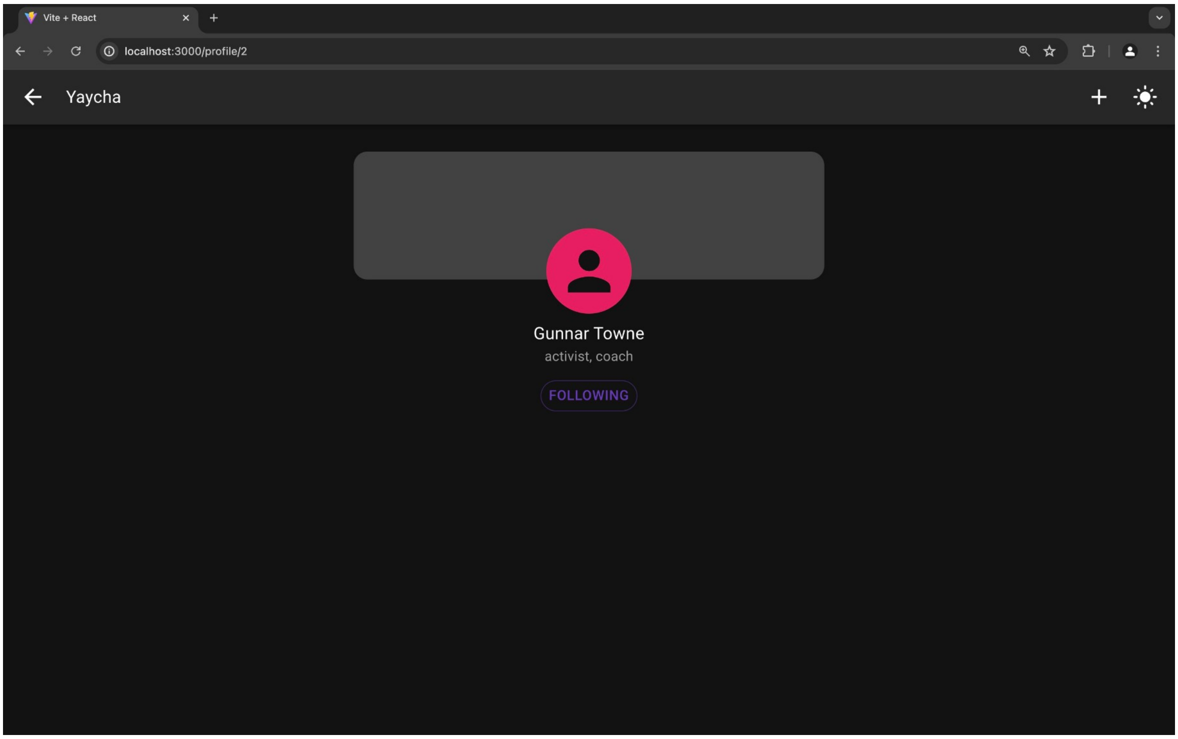
React - src/components/UserList.jsx

```
<ListItem key={item.id}>
  <ListItemButton ...>
    ...
    <ListItemSecondaryAction>
      <FollowButton user={item.user} />
    </ListItemSecondaryAction>
  </ListItemButton>
</ListItem>
```

ListItemSecondaryAction နဲ့ပြတာဖြစ်လို့ ListItemSecondaryAction ကိုတော့ ကိုယ့်ဘာသာ Import ထပ်လုပ်ပေးလိုက်ပါ။ User List ကို ပြတဲ့အခါ အခုလို Follow Button လေးတွေ တွဲပြသွားမှာဖြစ်ပါတယ်။

Button ကိုနှိပ်ပြီး Follow လုပ်လို့ရသလို Unfollow လည်းလုပ်လို့ရတယ်ဆိုတာကို တွေ့ရပါလိမ့်မယ်။ FollowButton ကို Profile Page တွေမှာလည်း ထည့်ပေးသင့်ပါတယ်။ ဒီတော့မှာ User တစ်ယောက်ရဲ့ Profile ကိုသွားလိုက်ရင် Follow / Unfollow လုပ်လို့ရတဲ့ လုပ်ဆောင်ချက် ပါသွားမှာပါ။ ကိုယ့်ဘာသာပဲ ထည့်ကြည့်လိုက်ပါ။


Vite + React
localhost:3000/likes/20/post
Yaycha
Likes
Gunnar Towne
activist, coach
FOLLOWING
D'angelo Jones
average enthusiast
FOLLOW
Corine Emard
cross-contamination devotee
FOLLOWING
Corine Emard
cross-contamination devotee
FOLLOWING
D'angelo Jones
average enthusiast
FOLLOW
Rosa Auer
inventor, student, designer


Vite + React
localhost:3000/profile/2
Yaycha
Gunnar Towne
activist, coach
FOLLOWING

Search User

ဆက်လက်ပြီး User တွေ ရှာလို့ရတဲ့ လုပ်ဆောင်ချက်လည်း ထပ်ထည့်ကြပါမယ်။ API ဘက်က `user.js` မှာ အခုလို ထပ်ဖြည့်ပေးပါ။

JavaScript - API - routers/user.js

```
router.get("/search", async (req, res) => {
  const { q } = req.query;

  const data = await prisma.user.findMany({
    where: {
      name: {
        contains: q,
      },
    },
    include: {
      followers: true,
      following: true,
    },
    take: 20,
  });

  res.json(data);
});
```

`where` နဲ့ `contains` ကိုတွဲသုံးပြီး Search Query Value ပါဝင်တဲ့ User တွေကို ရွေးထုတ်ယူလိုက်တာပါ။ Search Query ဆိုတာ URL Query ကိုပြောတာပါ။ ဥပမာ - `/search?q=bob` ။ ဒါကြောင့် `req.query` နဲ့ URL Query ကို အပေါ်မှာ အရင်ထုတ်ယူထားတာ သတိပြုပါ။

ပြီးတဲ့အခါ Client Side က `fetcher.js` မှာ အခုလိုရေးပေးပါ။

JavaScript - src/libs/fetcher.js

```
export async function fetchSearch(q) {
  const res = await fetch(`${api}/search?q=${q}`);
  return res.json();
}
```

ဘာမှထွေထွေထူးထူးမပါဘဲ စောစောက API Route ကို နောက်ကနေ `?q` Parameter နဲ့ Fetch လုပ်ထားတာပါ။ ပြီးတဲ့အခါ `pages/Search.jsx` မှာ အခုလိုရေးပေးပါ။

React - src/pages/Search.jsx

```
import {
  Alert,
  Avatar,
  Box,
  List,
  ListItem,
  ListItemAvatar,
  ListItemButton,
  ListItemSecondaryAction,
  ListItemText,
    TextField,
} from "@mui/material";

import { useState } from "react";
import { useQuery } from "react-query";

import { fetchSearch } from "../libs/fetcher";
import FollowButton from "../components/FollowButton";

import { useDebounce } from "@uidotdev/usehooks";

export default function Search() {
  const [query, setQuery] = useState("");
```

```
const debouncedQuery = useDebounce(query, 500);

const { isLoading, isError, error, data } = useQuery(
  ["search", debouncedQuery],
  () => {
    return fetchSearch(debouncedQuery);
  }
);

if (isError) {
  return (
    <Box>
      <Alert severity="warning">{error.message}</Alert>
    </Box>
  );
}

return (
  <Box>
    <TextField
      fullWidth={true}
      variant="outlined"
      placeholder="Search user"
      onKeyUp={e => {
        setQuery(e.target.value);
      }}
    />
    {isLoading ? (
      <Box sx={{ textAlign: "center", mt: 4 }}>Loading...</Box>
    ) : (
      <List>
        {data.map(user => {
          return (
            <ListItem key={user.id}>
              <ListItemButton
```

```
                    onClick={() =>
                      navigate(`/profile/${user.id}`)
                    }>
                    <ListItemAvatar>
                      <Avatar />
                    </ListItemAvatar>
                    <ListItemText
                      primary={user.name}
                      secondary={user.bio}
                    />
                    <ListItemSecondaryAction>
                      <FollowButton user={user} />
                    </ListItemSecondaryAction>
                  </ListItemButton>
                </ListItem>
              );
            })}
          </List>
        )}
      </Box>
  );
}
```

ကုဒ်ထဲမှာ useDebounce လို့ခေါ်တဲ့ လုပ်ဆောင်ချက်ထည့်သုံးထားတာကို သတိပြုပါ။ debounce ဆိုတာ ချက်ခြင်း အလုပ်မလုပ်ဘဲ ခဏဆိုင်းပြီးမှ လုပ်စေချင်တဲ့အလုပ်တွေအတွက် အသုံးဝင်ပါတယ်။ ဒီ လုပ်ဆောင်ချက် သုံးလို့ရဖို့အတွက် အခုလို Install လုပ်ပေးဖို့တော့ လိုပါတယ်။

```
npm i @uidotdev/usehooks
```

လိုရင်းကတော့ useQuery နဲ့ စောစောက Search API ကိုလှမ်းခေါ်လိုက်တာပါ။ ရှာချင်တဲ့ User ကို ရိုက်ရှာလို့ရအောင် Textfield တစ်ခုထည့်ပေးထားပါတယ်။ ရိုက်ထည့်လိုက်တဲ့ Value တွေကို keyup လုပ်လိုက်တိုင်း query State နဲ့ သိမ်းပါတယ်။ useQuery ရဲ့ Query Key ကိုပြန်ကြည့်ပါ။ query

State ကနေရတဲ့ debounceQuery ကို ထည့်ပေးထားပါတယ်။ ဒါကြောင့် query State ပြောင်းတိုင်း၊ တစ်နည်းအားဖြင့် Textfield မှာ တစ်ခုခု ရိုက်ထည့်လိုက်တိုင်း Query Refetch လုပ်မှာပါ။

အဲ့ဒီလို ရိုက်လိုက်တိုင်း အလုပ်လုပ်နေရင် နှေးမှားစိုးလို့ query State ကို တိုက်ရိုက်မသုံးဘဲ useDebounce နဲ့တစ်ဆင့်ခံ သုံးထားပါတယ်။ ရေးထားတဲ့ကုဒ်အရ useDebounce က query State မှာ အပြောင်းအလဲ ရှိတဲ့အခါ အနည်းဆုံး 500 မီလီစက္ကန့်စောင့်ပြီးမှ Update လုပ်ပေးမှာဖြစ်ပါတယ်။

User List ကိုပြတဲ့အခါ စောစောက <UserList> ကို ပြန်ယူမသုံးတော့ဘဲ နောက်တစ်ကြိမ် ထပ်ရေးပြီး ထည့်ထားပါတယ်။ Search ကနေရတဲ့ User Data ရဲ့ဖွဲ့စည်းပုံမတူတဲ့အတွက် ပြန်ယူသုံးရတာ အဆင်မပြေနိုင်လို့ပါ။

ဆက်လက်ပြီး ThemedApp.jsx မှာ Search Page ကိုသွားလို့ရတဲ့ Route တစ်ခုထပ်တိုးပေးပါ။

React - src/ThemedApp.jsx

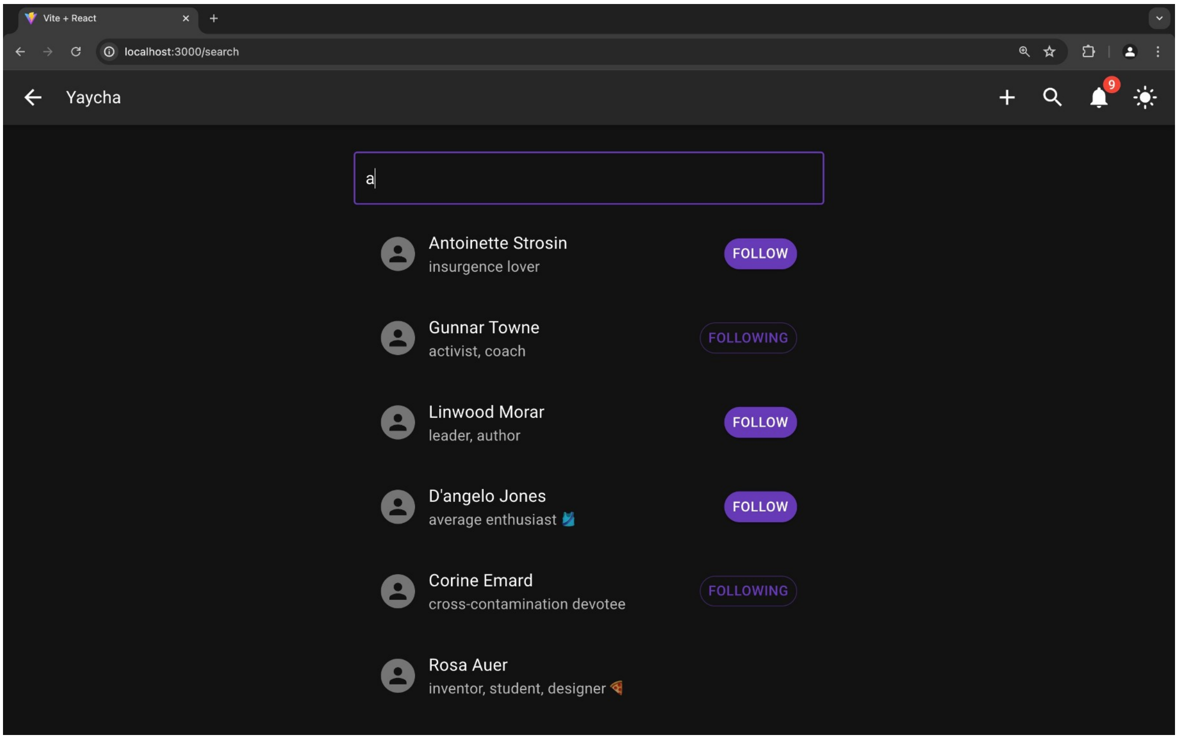
```
{
  path: "/search",
  element: <Search />,
},
```

ပြီးတဲ့အခါ components/Headers.jsx မှာ Search Button လေးတစ်ခု ထည့်ပေးပါ။

React - src/components/Header.jsx

```
<IconButton
  color="inherit"
  onClick={() => navigate("/search")}>
  <SearchIcon />
</IconButton>
```

ဒါတွေစုံရင် စမ်းကြည့်လို့ရပါပြီ။ Search Button ကိုနှိပ်လိုက်လို့ Search Page ကို ရောက်သွားတဲ့အခါ အခုလို User Name နဲ့ ရှာကြည့်လို့ရသွားပါလိမ့်မယ်။



ဒီနည်းနဲ့ ကိုယ် Follow လုပ်ချင်တဲ့ User တွေကိုရှာပြီး Follow လုပ်လို့ရသွားတဲ့သဘောပဲ ဖြစ်ပါတယ်။

Following Content

ဆက်လက်ပြီး ကိုယ် Follow လုပ်ထားတဲ့ သူတွေရဲ့ Content ကိုပဲ ကြည့်လို့ရအောင် လုပ်ပေးပါမယ်။ API ဘက်က `content.js` မှာ အခုလို ထပ်ဖြည့်ပေးပါ။

JavaScript - API - routers/content.js

```
router.get("/following/posts", auth, async (req, res) => {
  const user = res.locals.user;

  const follow = await prisma.follow.findMany({
    where: {
      followerId: Number(user.id),
    },
  });
```

```
  const users = follow.map(item => item.followingId);

  const data = await prisma.post.findMany({
    where: {
      userId: {
        in: users,
      },
    },
    include: {
      user: true,
      comments: true,
      likes: true,
    },
    orderBy: { id: "desc" },
    take: 20,
  });

  res.json(data);
});
```

အရင်ဆုံး ကိုယ် Follow လုပ်ထားတဲ့ User စာရင်းကို ထုတ်ယူပါတယ်။ ရလာတဲ့အခါ `UserId` တွေကို ထပ်ထုတ်ယူပါတယ်။ ပြီးတော့မှ `where` နဲ့ `in` ကို ပေါင်းစပ်ပြီး ကိုယ် Follow လုပ်ထားတဲ့ User တွေရဲ့ Post တွေချည်းပဲ Filter လုပ်ယူလိုက်တာပါ။

ထုံးစံအတိုင်း Client ဘက်က `fetcher.js` မှာလည်း အခုလို ရေးပေးပါ။

JavaScript - src/libs/fetcher.js

```
export async function fetchFollowingPosts() {
  const token = getToken();
  const res = await fetch(`${api}/content/following/posts`, {
    headers: {
      Authorization: `Bearer ${token}`,
    },
  });

  return res.json();
}
```

ပြီးတဲ့အခါ Home.jsx ကို အခုလို ပြင်ပေးပါ။

React - src/pages/Home.jsx

```
import { useState } from "react";
import { Alert, Box, Button, Typography } from "@mui/material";

import Form from "../components/Form";
import Item from "../components/Item";

import { useApp } from "../ThemedApp";

import { useQuery, useMutation } from "react-query";
import { queryClient } from "../ThemedApp";
import {
  deletePost,
  fetchPosts,
  postPost,
  fetchFollowingPosts,
} from "../libs/fetcher";

export default function Home() {
  const [showLatest, setShowLatest] = useState(true);
```

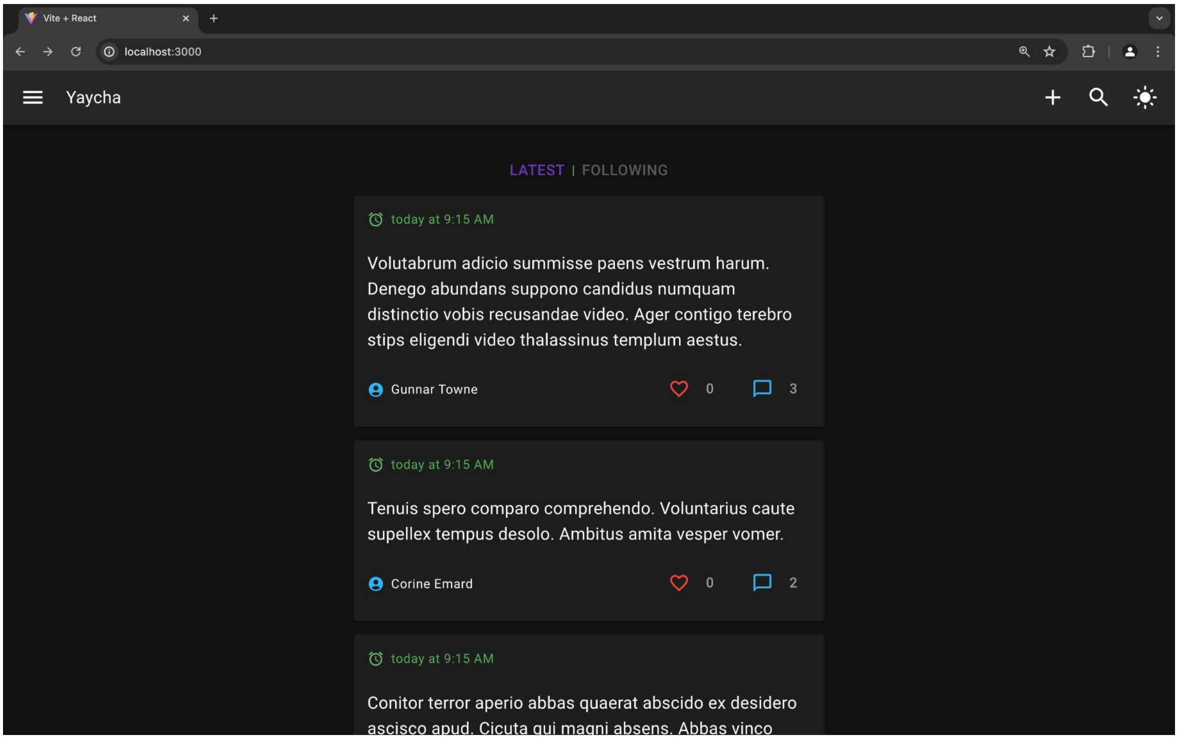
```
const { showForm, setGlobalMsg, auth } = useApp();
const { isLoading, isError, error, data } = useQuery(
  ["posts", showLatest],
  () => {
    if (showLatest) return fetchPosts();
    else return fetchFollowingPosts();
  }
);

const remove = useMutation(async id => deletePost(id), {
  onMutate: async id => {
    await queryClient.cancelQueries("posts");
    await queryClient.setQueryData(["posts", showLatest], old =>
      old.filter(item => item.id !== id)
    );

    setGlobalMsg("A post deleted");
  },
});

const add = useMutation(content => postPost(content), {
  onSuccess: async post => {
    await queryClient.cancelQueries("posts");
    await queryClient.setQueryData(["posts", showLatest], old => [
      post,
      ...old,
    ]);
    setGlobalMsg("A post added");
  },
});

if (isError) {
  return (
    <Box>
```

```
        <Alert severity="warning">{error.message}</Alert>
      </Box>
    );
  }

  if (isLoading) {
    return <Box sx={{ textAlign: "center" }}>Loading...</Box>;
  }

  return (
    <Box>
      {showForm && auth && <Form add={add} />}

      {auth && (
        <Box
          sx={{
            display: "flex",
            justifyContent: "center",
            alignItems: "center",
            mb: 1,
          }}>
          <Button
            disabled={showLatest}
            onClick={() => setShowLatest(true)}>
            Latest
          </Button>
          <Typography sx={{ color: "text.fade", fontSize: 15 }}>
            |
          </Typography>
          <Button
            disabled={!showLatest}
            onClick={() => setShowLatest(false)}>
            Following
          </Button>
        </Box>
```

```
      )}

      {data.map(item => {
        return (
          <Item
            key={item.id}
            item={item}
            remove={remove.mutate}
          />
        );
      })}
    </Box>
  );
}
```

showLatest State ပါဝင်သွားတာကို သတိပြုပါ။ useQuery က ဒီ State ပေါ်မူတည်ပြီး fetchPost() သို့မဟုတ် fetchFollowingPosts() ကို သင့်တော်သလို ခေါ်ပေးသွားမှာပါ။ ပြီးတော့ Query Key မှာလည်း ထည့်ပေးထားလို့ showLatest State ပြောင်းရင်လည်း အလိုအလျှောက် Refetch လုပ်ပေးမှာပါ။

add နဲ့ remove Mutation Function တွေကိုလည်း တစ်ချက်သတိပြုပါ။ setQueryData နဲ့ Update လုပ်တဲ့အခါ showLatest Query Key ကိုပါ ထည့်ပေးထားပါတယ်။ ပြီးတဲ့အခါ <Button> တွေကိုပဲ သုံးပြီးတော့ showLatest State ကို ပြောင်းလို့ရတဲ့ ခလုပ်တွေ ထည့်ပေးထားပါတယ်။ ဒါကြောင့် စမ်းကြည့်လိုက်ရင် အခုလို Latest သို့မဟုတ် ကိုယ် Follow လုပ်ထားသူတွေရဲ့ Post တွေကို ရွေးကြည့်လို့ရသွားတာကို တွေ့ရပါလိမ့်မယ်။



ဒါဟာ လုပ်လက်စပရောဂျက်မှာ Like / Unlike, Follow / Unfollow စတဲ့ Social Media Feature တွေ ပါဝင်သွားခြင်းပဲ ဖြစ်ပါတယ်။ အများအားဖြင့် သိပြီးသားရေးနည်းတွေကို ပေါင်းစပ်အသုံးပြုလိုက်တာဖြစ်ပြီး သတိပြုစရာ၊ ဖြည့်စွက်မှတ်သားစရာလေးတွေလည်း ပါဝင်သွားတာပဲ ဖြစ်ပါတယ်။

အခန်း (၁၇) - JavaScript - WebSocket

Web Technology ရဲ့ အဓိကအသက်သွေးကြောဖြစ်တဲ့ HTTP ဟာ Client ဘက်က Server ကို ဆက်သွယ်ရတဲ့ Pull Technology ခေါ် One Way Communication နည်းပညာဖြစ်ပါတယ်။ ဆက်သွယ်မှုကို Client ဘက်ကသာ ပြုလုပ်ရပြီ။ Server ဘက်က Client ကို ဆက်သွယ်တယ်ဆိုတာ မရှိပါဘူး။

Server Push ခေါ် နည်းပညာတစ်မျိုး HTTP/2 မှာ ပါဝင်လာပေမယ့် တချို့နေရာတွေမှာ အသုံးဝင်တဲ့ Optional နည်းပညာတစ်ခုသာ ဖြစ်ပါတယ်။ မူလနည်းပညာရဲ့ အလုပ်လုပ်ပုံ ပြောင်းလဲခြင်းမရှိပါဘူး။ ဒါကြောင့် HTTP မှာ Client to Server – One Way Communication ပုံစံသာ အလုပ်လုပ်ပါတယ်။

ဒီနည်းပညာရဲ့ အားသာချက်က Implementation ရိုးရှင်းခြင်းပါပဲ။ သုံးရလွယ်တယ်လို့ ပြောတာပါ။ အားနည်းချက်ကတော့ Real–Time မဖြစ်ခြင်း ဖြစ်ပါတယ်။ Server မှာ Update တစ်ခုခုရှိနေလည်း Server က ပို့ပေးမှာ မဟုတ်ပါဘူး။ ပို့လို့ မရပါဘူး။ Client က ဆက်သွယ်တောင်းယူမှသာ ပေးလို့ရမှာပါ။ ဆက်သွယ်မှုကို Client ဘက်ကပဲ စလို့ရတဲ့အတွက်ကြောင့်ပါ။ ဒါကြောင့် Server မှာ ရှိတဲ့ Update တွေ Client က ချက်ခြင်းမရလို့ Real Time မဖြစ်ဘူးလို့ ပြောတာပါ။ သိပ်တော့ အကျယ်မချဲ့တော့ပါဘူး။ ဒီစာအုပ်ကို ဖတ်နေသူဟာ ဒီအခြေခံသဘောကို သိပြီးသားဖြစ်လိမ့်မယ်လို့ ယူဆပါတယ်။ လိုအပ်လို့ နည်းနည်းပြန် Remind လုပ်တဲ့သဘောပါပဲ။

HTTP လို One Way Communication မဟုတ်ဘဲ Client – Server နှစ်ဖက်လုံးက လိုအပ်ရင် အပြန်အလှန် ဆက်သွယ်လို့ရတဲ့ Full Duplex ခေါ် Two Way Communication ကို လုပ်ပေးနိုင်တဲ့ WebSocket အကြောင်းကို လေ့လာကြမှာပါ။ Two Way ဖြစ်သွားလို့ စဉ်းစားရတာ၊ စီမံရတာ နည်းနည်း

ခေါင်းစားပေမယ့် ရေးနည်းက လွယ်ကူပါတယ်။

နမူနာကုဒ်လေးတချို့ ရေးပြီး စမ်းကြည့်ကြပါမယ်။ ပရောဂျက်ဖိုဒါအသစ်တစ်ခု တည်ဆောက်လိုက်ပါ။ ပြီးတဲ့အခါ ws ခေါ် Node နဲ့ WebSocket Server တစ်ခုကို အလွယ်တကူ ဖန်တီးလို့ရတဲ့ Package ကို အခုလို Install လုပ်ယူလိုက်ပါ။

```
npm i ws
```

ဆက်လက်ပြီး `index.js` အမည်နဲ့ ဒီကုဒ်ကိုရေးပေးပါ။

JavaScript - Server - index.js

```
const { WebSocketServer } = require("ws");

const wss = new WebSocketServer({ port: 4000 });

wss.on("connection", ws => {
  ws.on("message", msg => {
    console.log(`Message received: ${msg}`);
    ws.send(String(msg));
  });
});
```

`ws` Package ရဲ့အကူအညီနဲ့ WebSocket Server တစ်ခုကို Port 4000 မှာ Run လိုက်တာပါ။ `on connection` မှာ လုပ်ရမယ့် အလုပ်တွေကို စုစည်းသတ်မှတ်ထားပါတယ်။ `on message` နဲ့ Client ပေးပို့တဲ့ Message တွေလက်ခံရရှိရင် ဘာလုပ်ရမလဲ ဆက်လက်သတ်မှတ်ထားပါတယ်။ နမူနာအရ ဘာအလုပ်မှ ထူးထူးထွေထွေ မလုပ်ပါဘူး။ လက်ခံရရှိတဲ့ Message ကို String Format နဲ့ ပြန်ပို့ထားပါတယ်။ ဒါကြောင့် Client က သူပို့တာ သူပြန်ရမယ်ဆိုတဲ့ သဘောပါပဲ။

ဒီ ကုဒ်ကို `node` (သို့မဟုတ်) `nodemon` နဲ့ Run လိုက်ပါ။

```
nodemon index.js
```

ဒီ WebSocket Server နဲ့ ချိတ်ဆက်စမ်းသပ်ကြည့်ဖို့ Client ကုဒ်ဖိုင်တစ်ခု ရေးကြပါမယ်။ `index.html` အမည်နဲ့ နှစ်သက်ရာနေရာမှာ ရေးစမ်းကြည့်ပါ။

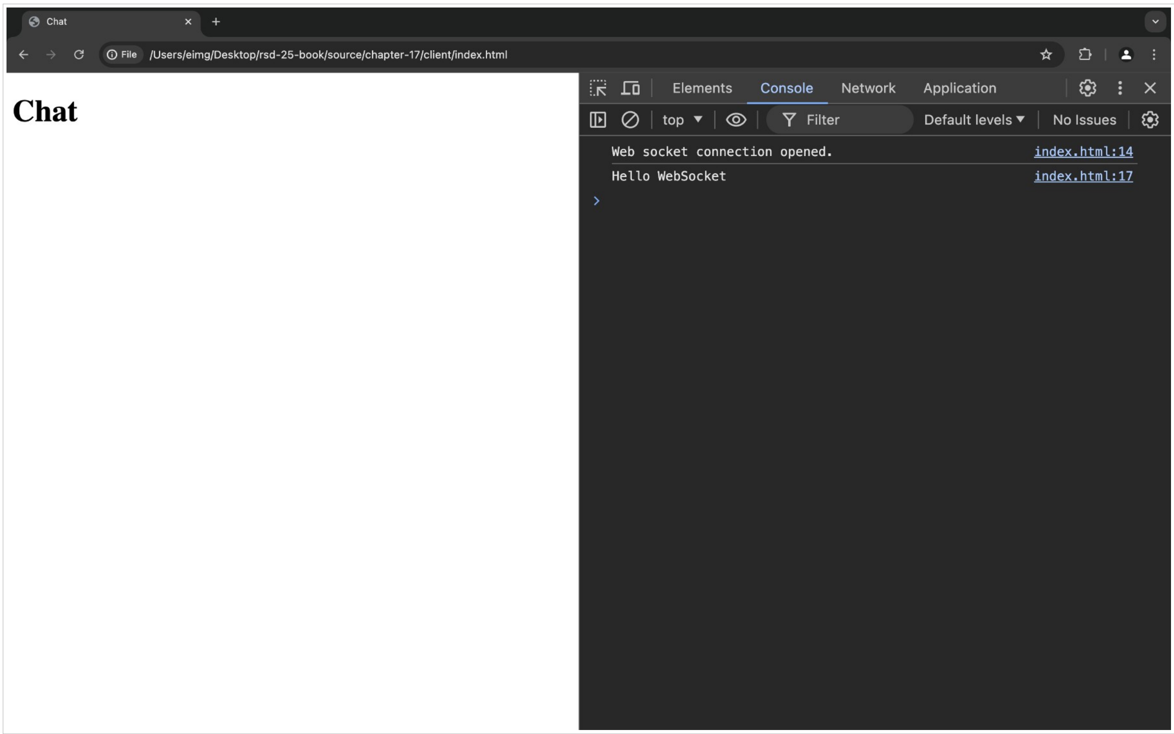
HTML - Client - index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Chat</title>
</head>
<body>
  <h1>Chat</h1>

  <script>
    const ws = new WebSocket("ws://localhost:4000");

    ws.addEventListener("open", () => {
      console.log("Web socket connection opened.");

      ws.addEventListener("message", e => {
        console.log(e.data);
      });

      ws.send("Hello WebSocket");
    });
  </script>
</body>
</html>
```

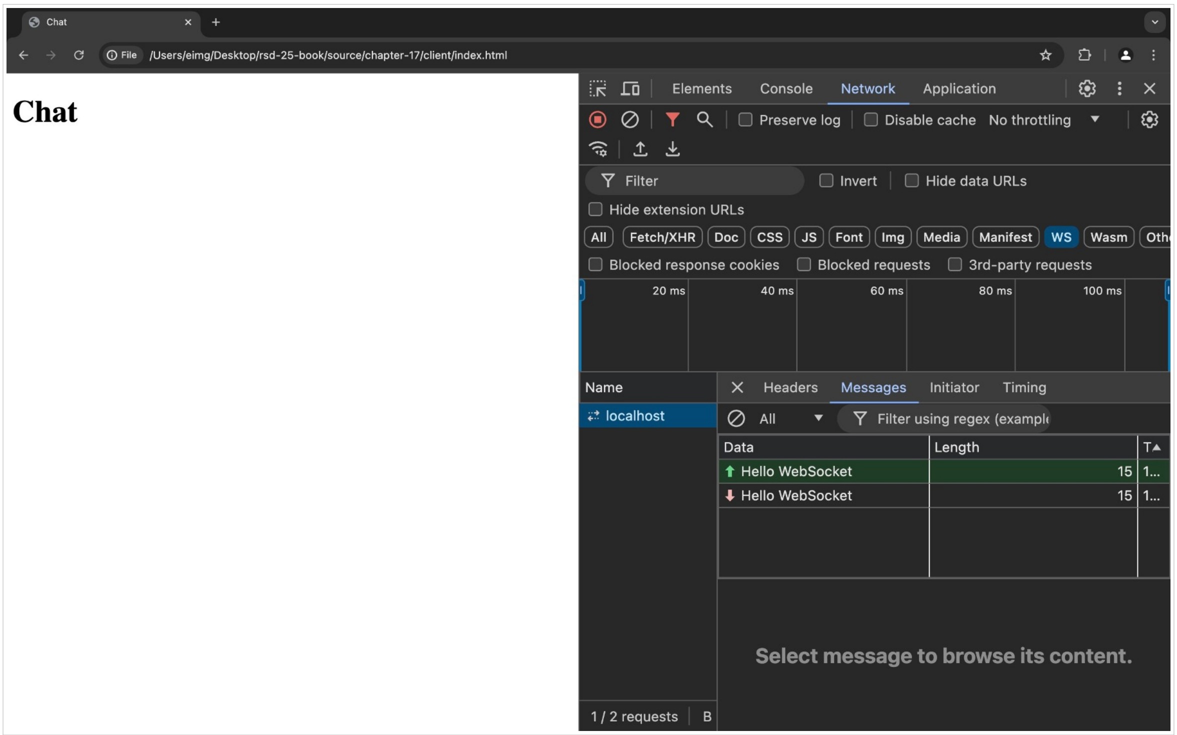
စောစောက Run လိုက်တဲ့ WebSocket Server ကိုဆက်သွယ်လိုက်တဲ့ Client-side ကုဒ်ပါ။ `on open` မှာ လုပ်ရမယ့် အလုပ်တွေကို စုရေးထားပါတယ်။ `on message` မှာ လက်ခံရရှိတဲ့ `data` ကို Console

မှာ ရိုက်ထုတ်ထားပါတယ်။ ပြီးတော့ `send()` နဲ့ စာလေးတစ်ကြောင်း ပို့လိုက်ပါတယ်။ ရေးထားတဲ့ကုဒ် အရ သူပို့လိုက်တဲ့စာကိုပဲ သူပြန်ရမှာပါ။

ဒီကုဒ်ကို Browser မှာ ဖွင့်ပြီးစမ်းကြည့်လို့ရပါတယ်။ Console မှာ အခုလို တွေ့ရပါလိမ့်မယ်။



သူပို့လိုက်တဲ့စာ သူပြန်ရတာမို့လို့ ဘာမှမထူးဆန်းဘူး ထင်ရပေမယ့်၊ သူပို့လိုက်တဲ့စာ Server ကိုရောက်ပြီး Server က ချက်ခြင်းပြန်ပို့တဲ့စာကို သူကလက်ခံရရှိတာ ဖြစ်ပါတယ်။ Client – Server အပြန်အလှန် ဆက်သွယ်လို့ ရသွားတာပါ။ ဒီ ဆက်သွယ်မှုကို Browser DevTools ရဲ့ Network မှာလည်း လေ့လာကြည့်လို့ရပါတယ်။



ညာဘက်ခြမ်းက DevTools → Network → ws → (localhost:4000) → messages မှာ Client နဲ့ Server တို့အပြန်အလှန် ပေးပို့တဲ့ Message တွေကို တွေ့ရခြင်း ဖြစ်ပါတယ်။

Real-Time Chat App

ဒီနည်းပညာကိုသုံးပြီး ရိုးရိုး JavaScript နဲ့ပဲ Real-Time Chat App လေးတစ်ခု လုပ်ကြည့်ကြပါမယ်။ အရင်ဆုံး Server-side ကုဒ်ကို အခုလို ပြင်လိုက်ပါ။

JavaScript - Server - index.js

```
const { WebSocketServer } = require("ws");

const wss = new WebSocketServer({ port: 4000 });

const clients = [];
```

```
wss.on("connection", ws => {
  clients.push(ws);

  ws.on("message", msg => {
    clients.map(client => {
      client.send(String(msg));
    });
  });
});
```

စာစာကကုဒ်နဲ့ သိပ်မကွာပါဘူး။ `clients` Array အလွတ်တစ်ခု ကြေညာလိုက်ပြီး `on connection` မှာ `ws` Connection ကို `clients` Array ထဲမှာ ထည့်သိမ်းလိုက်တာပါ။ ပြီးတဲ့အခါ `on message` မှာ `clients` Array ကို `map` လုပ်ပြီး ရှိသမျှ Client အကုန်လုံးထံ Message ကို ပြန်ပို့ထားပါတယ်။ ဒါကြောင့် Client တစ်ခုက Message ပို့လာရင် အဲ့ဒီ Message ကို ချိတ်ဆက်ထားသမျှ Client အားလုံးထံ ပြန်ပို့လိုက်တဲ့ သဘောပါ။

Client-side HTML ကုဒ်ထဲမှာ လိုအပ်တဲ့ Element တွေ အခုလို ထည့်ပေးပါ။

HTML - Client - index.html

```
<h1>Chat</h1>

<ul id="list"></ul>

<form id="form">
  <input type="text" id="user" placeholder="Name">
  <input type="text" id="msg" placeholder="Message">
  <button>Send</button>
</form>
```

`<ul>` List တစ်ခုနဲ့ `<form>` တစ်ခု ထပ်ထည့်ပေးလိုက်တာပါ။ ပြီးတဲ့အခါ Function တစ်ခု အခုလိုရေးပေးပါ။

JavaScript - Client - index.html

```
function chatItem({ name, msg }) {
  const li = document.createElement("li");
  li.innerHTML = `<b>${name}:</b> ${msg}`;
  list.appendChild(li);
}
```

ဒီ Function က ပေးလာတဲ့ `name` နဲ့ `msg` ကိုသုံးပြီး `<li>` Element တစ်ခု Create လုပ်ပေးပါတယ်။ ရလာတဲ့ `<li>` Element ကို List ထဲမှာ `appendChild()` နဲ့ထည့်ပေးလိုက်မှာပါ။ ပြီးတဲ့အခါ Form Submit အတွက် အခုလို ရေးပေးပါ။

JavaScript - Client - index.html

```
form.onsubmit = e => {
  e.preventDefault();

  ws.send(JSON.stringify({
    name: user.value,
    msg: msg.value
  }));

  msg.value = "";
  msg.focus();
};
```

Form ကို Submit လုပ်လိုက်တဲ့အခါ `user` Input နဲ့ `msg` Input တို့က Value တွေကို JSON Format နဲ့ WebSocket Server ထံ လှမ်းပို့လိုက်တာပါ။ WebSocket ရဲ့ `onMessage` က အခုလိုဖြစ်မှာပါ။

JavaScript - Client - index.html

```
ws.addEventListener("open", () => {
  ws.addEventListener("message", e => {
    chatItem(JSON.parse(e.data));
  });
});
```

ပြန်ရလာတဲ့ JSON ကို `parse()` လုပ်ပြီး ကြိုရေးထားတဲ့ `chatItem()` ကိုခေါ်ပေးလိုက်တာပါ။

ဒါကြောင့် တစ်ယောက်က အမည်နဲ့ စာကို ပို့လိုက်ရင်၊ ချိတ်ဆက်ထားသူ အားလုံးဆီကို ချက်ခြင်း ရောက်သွားတဲ့ Real-Time Chat App ကလေးတစ်ခု ရသွားတာပဲ ဖြစ်ပါတယ်။ Client ကို Browser နှစ်ခုသုံးခုနဲ့ ဖွင့်ပြီး အပြန်အလှန် စာပို့ပြီး စမ်းကြည့်လို့ ရပါတယ်။

CSS Style တွေပါ အပြည့်အစုံပါတဲ့ Client HTML ကုဒ်အပြည့်အစုံကိုလည်း ထပ်ထည့်ပေးလိုက်ပါတယ်။

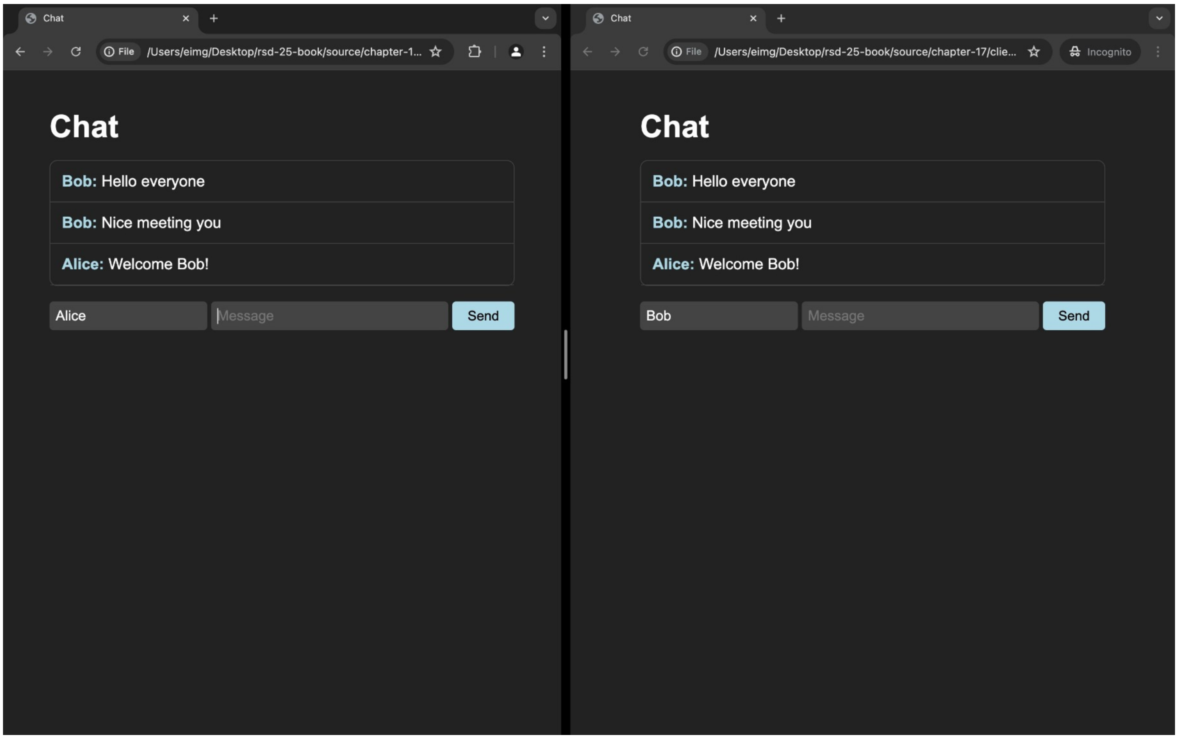
HTML - Client - index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Chat</title>
  <style>
    body {
      font-family: Arial, Helvetica, sans-serif;
      max-width: 600px;
      margin: 30px auto;
      background: #222;
      padding: 20px;
      border-radius: 5px;
      color: white;
    }
```

```
h1 {
  margin: 0;
  padding: 0;
}

#list {
  list-style: none;
  margin: 20px 0;
  padding: 0;
  border: 1px solid #444;
  border-radius: 10px;
}

#list li {
  padding: 15px;
  border-bottom: 1px solid #444;
}

#list li b {
  color: lightblue;
}

#form {
  display: flex;
  gap: 5px;
}

#form input:focus {
  outline: 0 none;
}

#form input {
  background: #444;
  color: white;
  border: 0 none;
```

```
      border-radius: 5px;
      font-size: 18px;
      padding: 8px;
    }

    #user {
      font-weight: bold;
    }

    #msg {
      flex-grow: 1;
    }

    #form button {
      border: 0 none;
      background: lightblue;
      padding: 8px 20px;
      border-radius: 5px;
    }
  </style>
</head>
<body>
  <h1>Chat</h1>

  <ul id="list"></ul>

  <form id="form">
    <input type="text" id="user" placeholder="Name">
    <input type="text" id="msg" placeholder="Message">
    <button>Send</button>
  </form>
```

```
  <script>
    const ws = new WebSocket("ws://localhost:4000");

    ws.addEventListener("open", () => {
      console.log("Connection opened.");

      ws.addEventListener("message", e => {
        chatItem(JSON.parse(e.data));
      });
    });

    form.onsubmit = e => {
      e.preventDefault();

      ws.send(JSON.stringify({
        name: user.value,
        msg: msg.value
      }));

      msg.value = "";
      msg.focus();
    };

    function chatItem({ name, msg }) {
      const li = document.createElement("li");
      li.innerHTML = `<b>${name}:</b> ${msg}`;
      list.appendChild(li);
    }
  </script>
</body>
</html>
```



အထက်မှာပြောခဲ့သလိုပါပဲ။ ရေးနည်းက အခက်ကြီးမဟုတ်ပါဘူး။ ဒါပေမယ့် Client – Server နှစ်ဘက် အသွားအပြန်ကို စဉ်းစားရေးသားရတဲ့အတွက် စီမံရတာတော့ နည်းနည်းခေါင်းစားနိုင်ပါတယ်။ နောက်တစ်ခန်းမှာ ဒီလုပ်ဆောင်ချက်ကို အသုံးပြုပြီး လုပ်လက်စပရောဂျက်မှာ Real-Time Notification စနစ် ထည့်သွင်းကြမှာ ဖြစ်ပါတယ်။

အခန်း (၁၈) - Project - Notifications

ဒီအခန်းမှာ လုပ်လက်စပရောဂျက်ကို နောက်ဆုံးတစ်ပိုင်းအနေနဲ့ Notification လုပ်ဆောင်ချက်ထည့်ပြီး လက်စသတ်လိုက်ကြတော့မှာပါ။ API – Client နှစ်ဖက်လုံးမှာ ပြင်ဆင်ဖြည့်စွက်စရာတွေ ရှိပါတယ်။ ပထမဆုံးအနေနဲ့ API ဘက်က `prisma.schema` မှာ Model အသစ်တစ်ခု ထပ်တိုးပေးပါ။

Prisma - prisma/schema.prisma

```
model Noti {
  id      Int      @id @default(autoincrement())
  type    String
  content String
  user    User     @relation(fields: [userId], references: [id],
                   onDelete: Cascade)
  userId  Int
  post    Post     @relation(fields: [postId], references: [id],
                   onDelete: Cascade)
  postId  Int
  read    Boolean  @default(false)
  created DateTime @default(now())
}
```

`Noti` Model ဖြစ်ပြီး `type` မှာ Like လုပ်တဲ့ Noti လား Comment လုပ်တဲ့ Noti မှတ်သားသိမ်းဆည်းမှာ ဖြစ်ပါတယ်။ `content` ကတော့ Notification Message ကိုသိမ်းချင်တာပါ။ User Relationship ပါပါ

တယ်။ ဘယ်သူက Like လုပ်တာလဲ သို့မဟုတ် ဘယ်သူက Comment လုပ်တာလဲဆိုတဲ့ လုပ်လိုက်တဲ့သူကို သိမ်းချင်တာပါ။ Post Relationship ပါပါတယ်။ ဘယ် Post မှာ Like လုပ်တာလဲ သို့မဟုတ် ဘယ် Post မှာ Comment လုပ်တာလဲ ဆိုတာကို သိမ်းချင်တာပါ။ Comment ကို Like လုပ်လိုက်ရင်လည်း အဲ့ဒီ Comment ရှိနေတဲ့ Post ကိုပဲသိမ်းမှာပါ။ ဒါကြောင့် Comment Relationship တော့ ထပ်ထည့်မထားပါဘူး။ `read` ကတော့ Noti အသစ်လား၊ ဖတ်ပြီးသားလား မှတ်ဖို့အတွက်ဖြစ်ပါတယ်။ Default ကို `false` လို့ပေးထားတာ တွေ့ရပါလိမ့်မယ်။

Relationship တွေပါလို့ User Model နဲ့ Post Model တို့ကိုလည်း အခုလို ထပ်တိုးပေးပါ။

Prisma - prisma/schema.prisma

```
model User {
  ...
  notis     Noti[]
  ...
}

model Post {
  ...
  notis     Noti[]
  ...
}
```

လိုအပ်တဲ့ Schema ရေးပြီးရင် Migrate ပြန်လုပ်ပေးဖို့လိုပါတယ်။

```
npx prisma migrate dev --name=notis
npx prisma migrate reset
```

ပထမ Command က Migration အသစ်ပြန်ထုတ်တာဖြစ်ပြီး၊ ဒုတိယ Command ကတော့ Seed Data တွေကို အစကနေ အသစ်ပြန် Run ပြီး ထည့်ပေးစေချင်လို့ပါ။

Noti - API

ဆက်လက်ပြီး `routers/content.js` မှာ Noti Data တွေနဲ့ပက်သက်တဲ့လုပ်ဆောင်ချက်တွေ အခုလို ထပ်တိုးပေးပါ။

JavaScript - API - routers/content.js

```
router.get("/notis", auth, async (req, res) => {
  const user = res.locals.user;
  const notis = await prisma.noti.findMany({
    where: {
      post: {
        userId: Number(user.id),
      },
    },
    include: { user: true },
    orderBy: { id: "desc" },
    take: 20,
  });

  res.json(notis);
});

router.put("/notis/read", auth, async (req, res) => {
  const user = res.locals.user;

  await prisma.noti.updateMany({
    where: {
      post: {
        userId: Number(user.id),
      },
    },
    data: { read: true },
  });

  res.json({ msg: "Marked all notis read" });
```

```
});

router.put("/notis/read/:id", auth, async (req, res) => {
  const { id } = req.params;

  const noti = await prisma.noti.update({
    where: { id: Number(id) },
    data: { read: true },
  });

  res.json(noti);
});

async function addNoti({ type, content, postId, userId }) {
  const post = await prisma.post.findUnique({
    where: {
      id: Number(postId),
    },
  });

  if (post.userId == userId) return false;

  return await prisma.noti.create({
    data: {
      type,
      content,
      postId: Number(postId),
      userId: Number(userId),
    },
  });
}
```

Route (၃) ခုနဲ့ Function (၁) ခု ထပ်တိုးလိုက်တာပါ။

- `GET /notis`
- `PUT /notis/read`
- `PUT /notis/read/:id`

`GET /notis` က လက်ရှိ User ရဲ့ Post တွေအတွက် Noti တွေကို ထုတ်ယူပြီး ပြန်ပေးပါတယ်။ Latest ဖြစ်တာကို အရင်မြင်ရအောင် `desc` နဲ့ ပြောင်းပြန်စီယူပြီး အခု (၂၀) ပဲထုတ်ယူထားပါတယ်။

`PUT /notis/read` ကတော့ လက်ရှိ User ရဲ့ Post တွေအတွက် Noti အားလုံးကို `read: true` လုပ်ပေးလိုက်တာပါ။ `PUT /notis/read/:id` ကတော့ `id` အတိအကျနဲ့ Noti တစ်ခုကို `read: true` လုပ်ပေးလိုက်တာပါ။

`addNoti()` ကတော့ Route မဟုတ်ဘဲ လိုတဲ့နေရာက ခေါ်သုံးဖို့အတွက် ရေးပေးထားတဲ့ Function တစ်ခုပါ။ Post ပိုင်ရှင်က Like / Comment လုပ်သူ ဟုတ်မဟုတ် စစ်ထားပါတယ်။ ဟုတ်ရင်ဆက်အလုပ်မလုပ် ပါဘူး။ ဆိုလိုတာက <u>ကိုယ့် Post မှာ ကိုယ့်ဘာသာ Like / Comment လုပ်တာတွေကို Noti အနေနဲ့ ထည့်မသိမ်းဘူးလို့ ဆိုလိုတာပါ</u>။

Noti ထည့်မယ့်အခြေအနေက လက်ရှိမှာ (၃) ခုပဲ ရှိပါတယ်။ Comment, Post Like နဲ့ Comment Like ဖြစ်ပါတယ်။ ဒါကြောင့် အဲ့ဒီလုပ်ဆောင်ချက် (၃) ခုမှာ အခုလို ဖြည့်ပေးပါ။

JavaScript - API - routers/content.js

```
router.post("/comments", auth, async (req, res) => {
  ...

  await addNoti({
    type: "comment",
    content: "reply your post",
    postId,
    userId: user.id,
  });

  res.json(comment);
```

```
});

router.post("/like/posts/:id", auth, async (req, res) => {
  ...

  await addNoti({
    type: "like",
    content: "likes your post",
    postId: id,
    userId: user.id,
  });

  res.json({ like });
});

router.post("/like/comments/:id", auth, async (req, res) => {
  ...

  await addNoti({
    type: "like",
    content: "likes your comment",
    postId: id,
    userId: user.id,
  });

  res.json({ like });
});
```

အရင်ရေးပြီးသားတွေကိုပြင်စရာမလိုပါဘူး။ `addNoti()` တွေ Response မပြန်ခင်ထည့်ပေးလိုက်တာပါ။ ဒါကြောင့် Like / Comment လုပ်တိုင်း သက်ဆိုင်ရာ Noti ကိုလည်း တစ်ခါထဲ ထည့်သိမ်းသွားပါပြီ။

Noti - Client

Client ဘက်ခြမ်းမှာ လိုတာဆက်လုပ်ကြပါမယ်။ ထုံးစံအတိုင်း လိုအပ်တဲ့ Fetcher Function တွေကို `fetcher.js` မှာ အခုလို ရေးပေးရပါမယ်။

JavaScript - src/libs/fetcher.js

```
export async function fetchNotis() {
  const token = getToken();
  const res = await fetch(`${api}/content/notis`, {
    headers: {
      Authorization: `Bearer ${token}`,
    },
  });

  return res.json();
}

export async function putAllNotisRead() {
  const token = getToken();
  const res = await fetch(`${api}/content/notis/read`, {
    method: "PUT",
    headers: {
      Authorization: `Bearer ${token}`,
    },
  });

  return res.json();
}

export async function putNotiRead(id) {
  const token = getToken();
  const res = await fetch(`${api}/content/notis/read/${id}`, {
    method: "PUT",
    headers: {
      Authorization: `Bearer ${token}`,
```

```
    },
  });

  return res.json();
}
```

သက်ဆိုင်တဲ့ Noti တွေကိုယူပေးဖို့ `fetchNotis()`၊ Noti အကုန်လုံးကို `read` လုပ်ဖို့ `putAllNotisRead()` နဲ့ သက်ဆိုင်ရာ Noti တစ်ခုကို `read` လုပ်ဖို့ `putNotiRead()` ဆိုပြီး (၃) ခုထပ်ဖြည့်လိုက်တာပါ။

ဆက်လက်ပြီး `pages/Notis.jsx` ဖိုင်တစ်ခု တည်ဆောက်ပြီး အခုလို ရေးပေးလိုက်ပါ။

React - src/pages/Notis.jsx

```
import {
  Box,
  Card,
  Avatar,
  Button,
  Typography,
  CardContent,
  CardActionArea,
} from "@mui/material";

import {
  Comment as CommentIcon,
  Favorite as FavoriteIcon,
} from "@mui/icons-material";

import { useNavigate } from "react-router-dom";
import { format } from "date-fns";
import { useQuery, useMutation } from "react-query";
import { queryClient } from "../ThemedApp";
```

```
import {
  fetchNotis,
  putAllNotisRead,
  putNotiRead
} from "../libs/fetcher";

export default function Notis() {
  const navigate = useNavigate();

  const { isLoading, isError, error, data } = useQuery(
    "notis", fetchNotis
  );

  const readAllNotis = useMutation(putAllNotisRead, {
    onMutate: async () => {
      await queryClient.cancelQueries("notis");
      await queryClient.setQueryData("notis", old => {
        return old.map(noti => {
          noti.read = true;
          return noti;
        });
      });
    },
  });

  const readNoti = useMutation(id => putNotiRead(id));

  if (isError) {
    return (
      <Box>
        <Alert severity="warning">{error.message}</Alert>
      </Box>
    );
  }
```

```
if (isLoading) {
  return <Box sx={{ textAlign: "center" }}>Loading...</Box>;
}

return (
  <Box>
    <Box sx={{ display: "flex", mb: 2 }}>
      <Box sx={{ flex: 1 }}></Box>
      <Button
        size="small"
        variant="outlined"
        sx={{ borderRadius: 5 }}
        onClick={() => {
          readAllNotis.mutate();
        }}>
        Mark all as read
      </Button>
    </Box>

    {data.map(noti => {
      return (
        <Card
          sx={{ mb: 2, opacity: noti.read ? 0.3 : 1 }}
          key={noti.id}>
          <CardActionArea
            onClick={() => {
              readNoti.mutate(noti.id);
              navigate(`/comments/${noti.postId}`);
            }}>
            <CardContent
              sx={{
                display: "flex",
                opacity: 1,
              }}>
```

```
                {noti.type == "comment" ? (
                  <CommentIcon color="success" />
                ) : (
                  <FavoriteIcon color="error" />
                )}

                <Box sx={{ ml: 3 }}>
                  <Avatar />

                  <Box sx={{ mt: 1 }}>
                    <Typography
                      component="span"
                      sx={{ mr: 1 }}>
                      <b>{noti.user.name}</b>
                    </Typography>

                    <Typography
                      component="span"
                      sx={{
                        mr: 1,
                        color: "text.secondary",
                      }}>
                      {noti.content}
                    </Typography>

                    <Typography
                      component="span"
                      color="primary">
                      <small>
                        {format(
                          noti.created,
                          "MMM dd, yyyy"
                        )}
                      </small>
                    </Typography>
```

```
              </Box>
            </Box>
          </CardContent>
        </CardActionArea>
      </Card>
    );
  })}
</Box>
);
}
```

လိုအပ်တာ အကုန်စုံအောင်ရေးထားလို့ ကုဒ်တွေများပေမယ့် လိုရင်းကတော့ `fetchNotis()` နဲ့ Data တွေယူပြီး ပြပေးလိုက်တာပါပဲ။ ထူးခြားချက်အနေနဲ့ Noti က `read: true` ဖြစ်နေရင် Opacity လျှော့ပြီး ပြထားပါတယ်။ Noti ကို နှိပ်လိုက်ရင် သက်ဆိုင်ရာ Post (Comments) ရောက်သွားမှာပါ။ နှိပ်လိုက်တာနဲ့ `putNotiRead()` ကိုလည်း ခေါ်ထားပါတယ်။ `useMutation` သုံးထားပေမယ့် ဖြည့်စွက်လုပ်ဆောင်ချက် ဘာမှ မထည့်ထားပါဘူး။

`Mark all as read` ခလုပ်တစ်ခုလည်း ထည့်ပေးထားပါတယ်။ `useMutation` နဲ့ပဲ `putAllNotisRead()` ကို ခေါ်လိုက်တာပါ။ သူ့အတွက်တော့ QueryData တွေကို Update လုပ်ပေးထားပါတယ်။ ဒါကြောင့် API က ပြန်လာတာကို စောင့်စရာမလိုဘဲ ခလုပ်နှိပ်လိုက်တာနဲ့ Client-Side မှာတင် အားလုံး Read ဖြစ်သွားတဲ့ ပုံစံနဲ့ ချက်ခြင်းပြပေးမှာပါ။

ဒီ Page အတွက် Route တစ်ခုလည်း `ThemedApp.jsx` မှာ အခုလိုထည့်ပေးပါ။

React - src/ThemedApp.jsx

```
{
  path: "/notis",
  element: <Notis />,
},
```

ပြီးရင် components/Header.jsx မှာ Noti Count ကိုယူပေးနိုင်တဲ့ လုပ်ဆောင်ချက်တွေ သင့်တော်တဲ့နေရာမှာ အခုလို ထည့်ပေးပါ။

React - src/components/Header.jsx

```
const { isLoading, isError, data } = useQuery(
  ["notis", auth], fetchNotis
);

function notiCount() {
  if (!auth) return 0;
  if (isLoading || isError) return 0;

  return data.filter(noti => !noti.read).length;
}
```

useQuery() နဲ့ fetchNotis() ကို ပေါင်းစပ်ပြီးတော့ပဲ Noti တွေကို ရယူလိုက်တာပါ။ useQuery ရဲ့ Query Key ကိုတော့ သတိပြုပါ။ auth ကိုထည့်ပေးထားပါတယ်။ auth အပြောင်းအလဲရှိရင် Noti Update ကို ထပ်ယူပေးစေချင်လို့ပါ။ ပြီးတဲ့အခါ notiCount() Function နဲ့ read: false ဖြစ်နေတဲ့ Noti length ကို ပြန်ပေးထားပါတယ်။

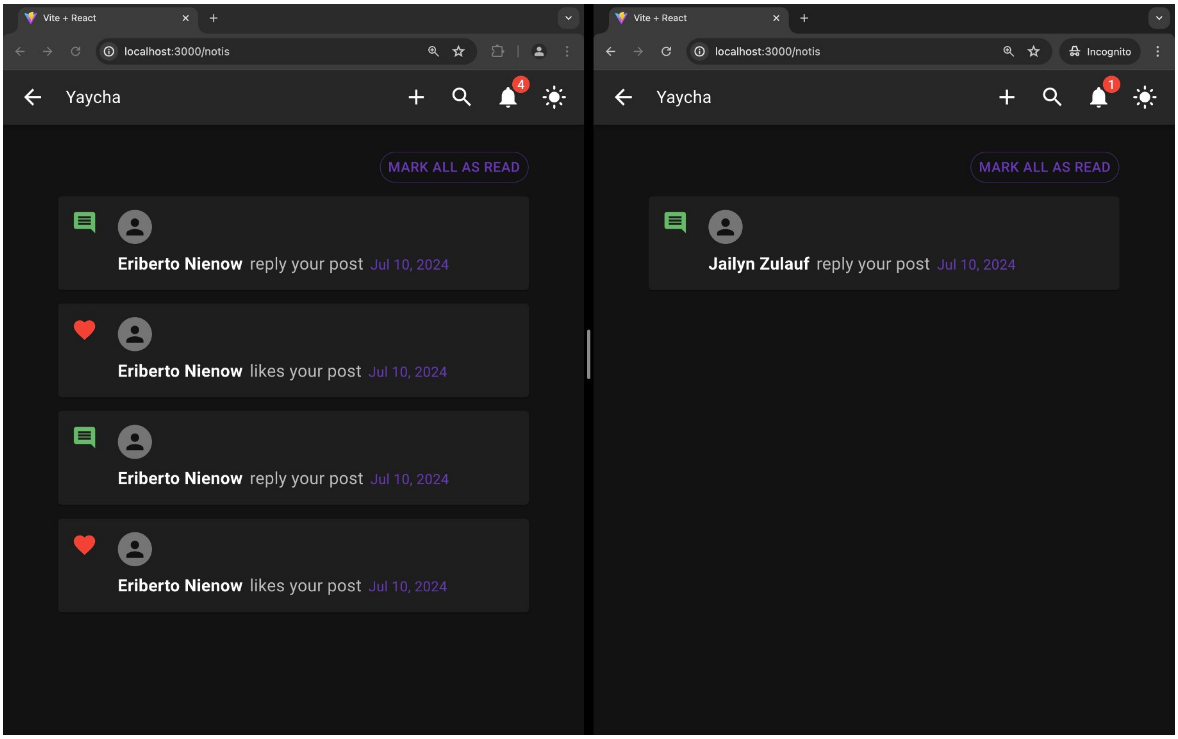
ဆက်လက်ပြီး Add Button တွေ Search Button တွေနားမှာ Noti Button ကို Count နဲ့တကွ အခုလို ပြပေးပါ။

React - src/components/Header.jsx

```
{
  auth && (
    <IconButton
      color="inherit"
      onClick={() => navigate("/notis")}>
      <Badge
        color="error"
```

```
        badgeContent={notiCount()}>
        <NotiIcon />
      </Badge>
    </IconButton>
  );
}
```

Badge Component ကိုသုံးထားပါတယ်။ MUI ကနေ Import လုပ်ပေးရပါလိမ့်မယ်။ ဒါတွေစုံအောင် ထည့်ပြီးရင် ဖော်ပြပုံက အခုလို ဖြစ်မှာပါ။



Notification စနစ်တစ်ခု ထည့်သွင်းရရှိသွားပြီဖြစ်ပါတယ်။ စမ်းကြည့်ဖို့အတွက် အကောင့် (၂) ခုလောက်နဲ့ ဟိုဘက်ဒီဘက် Login ဝင်ပြီး Like တွေ Comment တွေလုပ်ကြည့်ပါ။

Real-Time Notification - API

Notification စနစ်တစ်ခု ရသွားပြီဆိုပေမယ့် Real-Time တော့ဖြစ်ဦးမှာ မဟုတ်ပါဘူး။ တခြား User တစ်ယောက်က Like / Comment လုပ်လိုက်ချိန်မှာ Noti က ကိုယ့်ဆီကို ချက်ခြင်းရောက်လာမှာ မဟုတ်ပါဘူး။ React Query ရဲ့ အကူအညီနဲ့ Noti Page ကို သွားလိုက်တဲ့အချိန်တွေ Browser Focus ဖြစ်သွားချိန်တွေမှာ Update ယူပေးလို့ အဆင်ပြေသင့်သလောက်ပြေပေမယ့် ပိုကောင်းသွားအောင် WebSocket နဲ့ ဖြည့်စွက်ပေးကြပါမယ်။

အရင်ဆုံး API ဘက်မှာ လိုအပ်တဲ့ Package ကို အခုလို Install လုပ်ပေးပါ။

```
npm i express-ws
```

`express-ws` ဟာ ရှေ့အခန်းမှာ လေ့လာခဲ့တဲ့ `ws` Package ကိုပဲအသုံးပြုပြီး Express နဲ့ တွဲပေးတဲ့ နည်းပညာဖြစ်ပါတယ်။ WebSocket လုပ်ဆောင်ချက်ကို သပ်သပ်ခွဲထားစရာမလိုဘဲ လက်ရှိ Express ကုဒ်တွေထဲမှာပဲ တွဲရေးလို့ ရသွားပါတယ်။ Install လုပ်ပြီးရင် `index.js` မှာ အခုလို Import လုပ်ယူပေးဖို့လိုပါတယ်။

JavaScript - API - index.js

```javascript
require("express-ws")(app);
```

Import လုပ်ရင်း `express-ws` ကို `app` နဲ့တွဲပေးလိုက်တာပါ။ ဒီကုဒ်ကို Route တွေမရေးခင်၊ Router တွေ Import မလုပ်ခင် အပေါ်နားမှာ ရေးပေးဖို့လိုပါတယ်။

ပြီးတဲ့အခါ `routers/ws.js` နဲ့ အခုလို Web Socket လုပ်ဆောင်ချက်ရေးပေးပါ။

JavaScript - API - routers/ws.js

```
const express = require("express");
const router = express.Router();
const jwt = require("jsonwebtoken");

const secret = process.env.JWT_SECRET;

let clients = [];

router.ws("/subscribe", (ws, req) => {
  console.log("WS: New connection received");

  ws.on("message", msg => {
    const { token } = JSON.parse(msg);
    console.log("WS: token received");

    jwt.verify(token, secret, (err, user) => {
      if (err) return false;

      clients.push({ userId: user.id, ws });

      console.log(`WS: Client added: ${user.id}`);
    });
  });
});

module.exports = { clients, wsRouter: router };
```

`router.ws()` နဲ့ WebSocket Route တစ်ခု ထည့်ရေးထားပါတယ်။ Client က ချိတ်ဆက်ပြီး `on message` မှာ JWT token ကိုလက်ခံရရှိတဲ့အခါ လက်ရှိ WebSocket Connection ကို clients Array ထဲမှာ `token` ပိုင်ရှင် `userId` နဲ့အတူ တွဲသိမ်းလိုက်ပါတယ်။ ပြီးတဲ့အခါ `clients` Array ရော `router` ကိုပါ Export လုပ်ပေးထားပါတယ်။

ဆက်လက်ပြီး Noti အသစ်ဝင်တာနဲ့ သက်ဆိုင်ရာ Client ဆီတန်းပို့ပေးစေဖို့အတွက် `content.js` က `addNoti()` Function မှာ အခုလို ဖြည့်စွက်ပေးပါ။

JavaScript - API - routers/content.js

```
async function addNoti({ type, content, postId, userId }) {
  const post = await prisma.post.findUnique({
    where: {
      id: Number(postId),
    },
  });

  if (post.userId == userId) return false;

  clients.map(client => {
    if (client.userId == post.userId) {
      client.ws.send(JSON.stringify({ event: "notis" }));
      console.log(`WS: event sent to ${client.userId}: notis`);
    }
  });

  return await prisma.noti.create({
    data: {
      type,
      content,
      postId: Number(postId),
      userId: Number(userId),
    },
  });
}
```

စောစောကရေးလက်စ `addNoti()` မှာ Noti တွေမထည့်ခင် `clients` Array ကို `map()` လုပ်ပြီး သက်ဆိုင်ရာ Client ဆီကို `{ event: "notis" }` ဆိုတဲ့ Message လေးလှမ်းပို့လိုက်တာပါ။ Noti Update ရှိတယ်လို့ ပြောလိုက်တဲ့ သဘောပါပဲ။ အဲ့ဒီလို Update ရှိတယ်လို့ ပြောယုံမဟုတ်ဘဲ Update

Noti Data တွေ လှမ်းပို့ရင်လည်း ရနိုင်ပါတယ်။ မပို့တော့ပါဘူး။ Update ရှိတာနဲ့ ရှိတယ်လို့ပဲ လှမ်းပြောလိုက်တာပါ။

ဒီလိုရေးလို့ရဖို့အတွက် `clients` Array ကို `routers/ws.js` ကနေ Import လုပ်ယူဖို့တော့လိုပါတယ်။ ကိုယ့်ဘာသာ Import လုပ်ပေးလိုက်ပါ။ ဆက်လက်ပြီး `index.js` မှာ `wsRouter` ကို အခုလို ချိတ်ဆက်ပြီး ထည့်ပေးဖို့လိုပါတသေးတယ်။

JavaScript - API - index.js

```
const { wsRouter } = require("./routers/ws");
app.use("/", wsRouter);
```

ဒါဆိုရင် API ဘက်ခြမ်းက စုံသွားပါပြီ။ လုပ်ဆောင်ချက်အားလုံး Real-Time ဖြစ်သွားတာ မဟုတ်သေးပေမယ့် Noti Update ရှိတာနဲ့ ချက်ခြင်း Client ကို အကြောင်းကြားပေးသွားမှာပါ။

Real-Time Notification - Client

Client-side ဘက်မှာ WebSocket လုပ်ဆောင်ချက်တွေ ရေးသားဖို့အတွက် `useWebSocket` လို့ခေါ်တဲ့ နည်းပညာကို သုံးကြပါမယ်။ ဒါကြောင့် အခုလို Install လုပ်ပေးပါ။

```
npm i react-use-websocket
```

ဆက်လက်ပြီး WebSocket Route လိပ်စာကို `.env` ဖိုင်မှာ အခုလို ထည့်ပေးလိုက်ပါ။

.env

```
VITE_API="http://localhost:8000"
VITE_WS="http://localhost:8000/subscribe"
```

ပြီးတဲ့အခါ `src` ထဲမှာပဲ `AppSocket.jsx` အမည်နဲ့ ဖိုင်အသစ်တည်ဆောက်ပြီး အခုလိုရေးပေးပါ။

React - src/AppSocket.jsx

```
import { useEffect } from "react";
import { useApp, queryClient } from "./ThemedApp";
import useWebSocket, { ReadyState } from "react-use-websocket";

export default function AppSocket() {
  const { auth } = useApp();

  const { sendJsonMessage, lastJsonMessage, readyState } =
  useWebSocket(
    import.meta.env.VITE_WS
  );

  useEffect(() => {
    if (auth && readyState === ReadyState.OPEN) {
      sendJsonMessage({
        token: localStorage.getItem("token"),
      });

      console.log("WS: connection ready & token sent");
    }
  }, [readyState, auth]);

  useEffect(() => {
    console.log("WS: new message received");
    if (lastJsonMessage && lastJsonMessage.event) {
      queryClient.invalidateQueries(lastJsonMessage.event);
    }
  }, [lastJsonMessage]);

  return <></>;
}
```

useWebSocket နဲ့ ReadyState တို့ကို react-use-websocket ကနေ Import လုပ်ယူထားပါ

တယ်။ `useWebSocket` အတွက် `.env` ထဲမှာ သိမ်းထားတဲ့ WebSocket Route ကို ပေးလိုက်ပြီး WebSocket Client ကိုတည်ဆောက်ယူပါတယ်။ WebSocket Server ကို ပို့စရာရှိရင် `sendJsonMessage` နဲ့ပို့လို့ရသွားပါပြီ။ `lastJsonMessage` မှာ WebSocket Server က ပြန်ပို့ထားတဲ့ Message ရှိနေပါလိမ့်မယ်။ `readyState` ကတော့ လက်ရှိ WebSocket Connection ရဲ့ အခြေအနေကို ဖော်ပြပါလိမ့်မယ်။ Connecting လုပ်နေတာလား၊ Open ဖြစ်သွားပြီလား၊ Closing လုပ်နေတာလား၊ Close ဖြစ်သွားပြီလား စတဲ့အခြေအနေတွေ သူ့ဆီမှာ ရှိနေမှာပါ။

ဆက်လက်ပြီး `useEffect()` နဲ့ `readyState` ပြောင်းရင် လုပ်ရမယ့်အလုပ်တွေ ပေးထားပါတယ်။ `auth` ပြောင်းရင်လည်း လုပ်စေချင်လို့ Dependency Array ထဲမှာ `auth` ကိုလည်းထည့်ပေးထားပါတယ်။ `auth` ရှိနေပြီး `readyState` က `OPEN` ဖြစ်နေရင် `token` ကိုပို့ခိုင်းလိုက်တာပါ။ <u>လက်ရှိ WebSocket Connecting က Server ရဲ့ `clients` Array ထဲကို ရောက်သွားပါလိမ့်မယ်</u>။

နောက်ထပ် `useEffect()` တစ်ခုနဲ့ `lastJsonMessage` ပြောင်းရင် လုပ်ရမယ့်အလုပ်တွေ ပေးထားပါတယ်။ ဆိုလိုတာက WebSocket Server က ပို့လာတဲ့ Message Update တွေရှိရင်လုပ်ရမယ်ဆိုတဲ့ သဘောပါ။ React Query နဲ့ Invalidate လုပ်ထားပါတယ်။ Invalidate လုပ်တယ်ဆိုတာ လက်ရှိ Cache Data ကို မသုံးတော့ဘဲ အသစ်ယူဖို့ပြောလိုက်တာပါ။ ဒါကြောင့် React Query က အသစ်ယူပြီး Update လုပ်ပေးသွားပါလိမ့်မယ်။ ဒါကြောင့် Server က Update ရှိကြောင်း အကြောင်းကြားတာနဲ့ ချက်ခြင်း အသစ်ယူပြီး အလုပ်လုပ်ပေးသွားလို့ Real-Time Update ကိုရရှိသွားမှာပဲ ဖြစ်ပါတယ်။

ဒီလုပ်ဆောင်ချက်ကို `ThemedApp.jsx` မှာ Import လုပ်ပြီး အခုလိုထည့်ပေးလိုက်ပါ။

React - src/ThemedApp.jsx

```
<ThemeProvider theme={theme}>
  <AppContext.Provider value={{ ... }}>
    <QueryClientProvider client={queryClient}>
      <RouterProvider router={router} />
      <AppSocket />
    </QueryClientProvider>
    <CssBaseline />
```

```
  </AppContext.Provider>
</ThemeProvider>
```

Real-Time Notification လုပ်ဆောင်ချက်ရသွားပါပြီ။ Browser နှစ်ခုသုံးခု ဘေးချင်းယှဉ်ဖွင့်ပြီး စမ်းကြည့်လို့ရပါတယ်။

လိုအပ်ချက်တွေတော့ ကျန်ပါသေးတယ်။ API ဘက်က WebSocket Connecting တွေကို `clients` Array ထဲမှာ ထည့်သိမ်းပေမယ့် ပြန်ထုတ်တဲ့လုပ်ဆောင်ချက် ထည့်ပေးမထားပါဘူး။ Connection ပိတ်သွားရင် `on close` မှာ Client ကို အလိုအလျှောက် ပြန်ထုတ်ပေးသင့်ပါတယ်။ ပြီးတော့ Noti Count တစ်ခုထဲကိုပဲ Real-Time ဖြစ်အောင် လုပ်ထားတာပါ။ Comment တစ်ခုထည့်လိုက်တာနဲ့ တစ်ဖက်မှာ ချက်ခြင်းပေါ်အောင်၊ Like လုပ်လိုက်တာနဲ့ တစ်ဖက်မှာ Like Count ချက်ခြင်းတက်သွားအောင်၊ စသည်ဖြင့် ထည့်လုပ်မထားပါဘူး။

ဒီလိုမျိုးလိုအပ်ချက်လေးတွေ ဟိုနားဒီနား အများကြီးကျန်သေးပေမယ့် အခြေခံအားဖြင့် ပါဝင်သင့်တဲ့ လုပ်ဆောင်ချက်တွေပါတဲ့ Social Media ပရောဂျက်လေးတစ်ခု ရသွားတာပါ။ အသုံးဝင်တဲ့ လေ့လာစရာတွေ အများကြီး ရသွားမယ်လို့ထင်ပါတယ်။

အခန်း (၁၉) - Next.js - Server-Side Rendering Project

Next.js ဟာ React Server–side Rendering (SSR) နည်းပညာပါ။ အခုနောက်ပိုင်း React နဲ့ Next.js ဟာ ခွဲမရတော့ဘဲ အတွဲလိုက်လိုတောင် ဖြစ်နေပါပြီ။ React Core Team မှာ Next.js Team ကလူတွေ ပါဝင်နေသလို Next.js ရဲ့လွှမ်းမိုးမှုတွေလည်း React ပေါ်မှာ ပိုပြီးတော့များလာပါတယ်။ React နဲ့ ပရောဂျက်တွေ တိုက်ရိုက်ရေးမယ့်အစား Next.js နဲ့ ရေးကြတာတွေလည်း များလာပါတယ်။

ဒီနေရာမှာ ဆက်စပ်သဘောသဘာဝတွေဖြစ်ကြတဲ့ SPA, SSR, RSC စတဲ့ အကြောင်းအရာတွေ ပြောဖို့လိုလာပါတယ်။ သိပ်အကျယ်မချဲ့တော့ဘဲ အကျဉ်းချုပ်လေးကြည့်ကြပါမယ်။

SPA – ပုံမှန်အားဖြင့် React နဲ့ React Router လိုနည်းပညာမျိုးသုံးပြီး ကုဒ်တွေရေးတဲ့အခါ Route တွေ Page တွေ ခွဲပြီးရေးလို့ရပါတယ်။ ဒီလိုခွဲရေးထားပေမယ့် အလုပ်လုပ်တဲ့အခါ HTML Document တစ်ခုပေါ်မှာ ရေးထားတဲ့ကုဒ်တွေ အကုန်လုံးကို စုပေါင်းထည့်သွင်း အလုပ်လုပ်သွားမှာဖြစ်ပါတယ်။ ဒီသဘောသဘာဝကို Single Page Application (SPA) လို့ခေါ်ပါတယ်။

Page တစ်ခုကနေ နောက်တစ်ခု ပြောင်းသွားတယ်ဆိုတာ တစ်ကယ့် HTML Document Page ပြောင်းသွားတာ မဟုတ်ပါဘူး။ Page တစ်ခုထဲပေါ်မှာ လက်ရှိ Component ကို Unmount လုပ်ပြီး နောက် Page Component ကို Mount လုပ်လိုက်တာပါပဲ။ စမ်းကြည့်ချင်ရင် ပြီးခဲ့တဲ့အခန်းတွေမှာ ရေးခဲ့တဲ့ **yaycha** App ကို အခုလို Build လုပ်ကြည့်နိုင်ပါတယ်။

```
npm run build
```

ဒီလို Build လုပ်လိုက်တဲ့အခါ `build` အမည်နဲ့ ဖိုဒါတစ်ခုဝင်သွားပြီး ပရောဂျက်ကို အသင့် Publish လုပ်လို့ရအောင် HTML, CSS, JavaScript (Static) ဖိုင်တွေ ထုတ်ပေးထားမှာပါ။ ဘာ Dependencies မှ ထပ်ထည့်ဖို့မလိုတော့ဘဲ ဒီဖိုင်တွေကို Web Server တစ်ခုနဲ့ Publish လုပ်ပြီး Browser နဲ့ချိတ်သုံးယုံပါပဲ။ CSS, JavaScript ဖိုင်တွေ လိုအပ်သလောက် ပါဝင်မှာဖြစ်ပေမယ့် `index.html` ဖိုင်တစ်ခုသာ ပါဝင်တာကိုလည်း တွေ့ရပါလိမ့်မယ်။ ကုဒ်အားလုံးက အဲ့ဒီ `index.html` ဖိုင်တစ်ခုထဲပေါ်မှာ အလုပ်လုပ်သွားမှာပါ။

ဒီနည်းရဲ့ အားနည်းချက်က User က App ကို အသုံးပြုဖို့ ရယူချိန်မှာ App ကြီးတစ်ခုလုံးကို ရယူရမှာဖြစ်ပါတယ်။ ဒါကြောင့် ပထမဆုံးအကြိမ် Loading ကြာတာမျိုးဖြစ်နိုင်ပါတယ်။ အားသာချက်ကတော့ တစ်ကြိမ်ရယူပြီးရင် App ကြီးတစ်ခုလုံးကို ရပြီးနေပြီမို့လို့ နောက် Page တွေ လုပ်ဆောင်ချက်တွေမှာ သွက်သွက်လက်လက်နဲ့ နေရာတင် ချက်ခြင်း အလုပ်လုပ်ပေးနိုင်ပါလိမ့်မယ်။ SPA ရဲ့ ဒုတိယအားနည်းချက် ဖြစ်နိုင်တာကတော့ Component Style ကိစ္စတွေ၊ JavaScript ကုဒ် Run တဲ့ကိစ္စတွေ အကုန်လုံးကို Browser ထဲမှာလုပ်တာဖြစ်လို့ Complex ဖြစ်တဲ့ App တွေမှာ Rendering နှေးတာမျိုးလည်း ဖြစ်နိုင်ပါတယ်။ ဒါကြောင့် အကုန်လုံးကို စုပြုံ Run တာမျိုးမဖြစ်စေဘဲ လိုတဲ့ Module ကို လိုတဲ့အချိန်မှာ ယူပြီး Run ပေးနိုင်တဲ့ လုပ်ဆောင်ချက်မျိုးတွေ ထည့်သုံးပေးဖို့ လိုနိုင်ပါတယ်။

SSR – Next.js လို Server–side Rendering နည်းပညာရဲ့ အလုပ်လုပ်ပုံက အစဉ်အလာ Server–side ပရောဂျက်တွေနဲ့ ပိုဆင်သွားပါတယ်။ ဥပမာ – Home, About, Contact ဆိုပြီး Page သုံးခုပါဝင်တယ်ဆိုကြပါစို့။ ပထမဆုံးအကြိမ်မှာ Home Page တစ်ခုထဲကိုပဲ Client Browser က ရယူအသုံးပြုလို့ ရနိုင်ပါတယ်။ User က လိုအပ်လို့ About ကို သွားလိုက်တော့မှ About Page ကို Server ကနေ ထပ်ယူပြီး ပြပေးလိုက်တာမျိုးပါ။ ရိုးရိုး အစဉ်အလာ Server–side ပရောဂျက်တွေနဲ့ မတူတာကတော့ ဒီလိုရယူတဲ့အလုပ်ကို Asynchronous ပုံစံ လုပ်ပေးနိုင်ပြီး Cache တွေဘာတွေနဲ့ တစ်ကြိမ်ယူဖူးရင် ထပ်ယူစရာမလိုအောင် စီစဉ်ပေးတာတွေ ပါသွားလို့ လိုမှယူတာချင်း အတူတူ ပိုမြန်သွားတဲ့သဘော ဖြစ်ပါတယ်။

ဒါ Server–side Rendering မဟုတ်သေးပါဘူး။ Server–side Routing ပဲရှိပါသေးတယ်။ ဒီနည်းရဲ့ အားသာချက်ကတော့ ပထမဆုံးအကြိမ် Initial Load မြန်သွားပါတယ်။ အကုန်ယူစရာမလိုတော့လို့ပါ။ အားနည်းချက်ကတော့ နောက် Page တွေကို လိုတဲ့အချိန်မှ ယူတာဖြစ်လို့ Page တစ်ခုကနေတစ်ခုကို အကူးမှာ အခုမှ ပထမဆုံးယူတဲ့ Page တွေဆိုရင် ခဏစောင့်ရတာမျိုးဖြစ်နိုင်ပါတယ်။

စောစောက SAP မှာ CSS တွေ JavaScript တွေကို Browser ထဲမှာပဲ အကုန် Run လို့ Rendering နှေး တာဖြစ်နိုင်တယ်လို့ ပြောခဲ့ပါတယ်။ Next.js လိုနည်းပညာမျိုးကတော့ Server-side မှာကတည်းက လုပ် သင့်တဲ့ JavaScript ကုဒ်တွေကို အကုန်ကြိုလုပ်ပြီး Static HTML ပြောင်းပေးထားပါတယ်။ Browser ဆီ ကိုရောက်တဲ့အခါ JavaScript ကုဒ်တွေကနေ HTML Elements တွေ Render လုပ်နေစရာ မလိုတော့ပါ ဘူး။ ဒါကြောင့် Rendering ပိုမြန်သွားပါတယ်။ ဒီလို Server-side ဘက်မှာ JavaScript ကုဒ်တွေကို Static HTML ဖြစ်အောင်ကြိုပြောင်းပြီး အလုပ်လုပ်လို့ Server-side Rendering နည်းပညာလို့ ခေါ်တာပါ။

ဒီနေရာမှာ Hydration လို့ခေါ်တဲ့ သဘောသဘာဝတစ်ခု ဖြည့်စွက်ပါဝင်လာပါတယ်။ လိုအပ်တဲ့ HTML Element တွေကို ကြို Render လုပ်လို့ရပေမယ့် Client-side ကျမှလုပ်ဖို့ မဖြစ်မနေလိုအပ်တဲ့ JavaScript တွေ ကျန်ပါသေးတယ်။ Hydration ဆိုတာ Render လုပ်ပြီးသား Static HTML ပေါ်မှာ UI Interaction တွေ Events တွေလို လိုအပ်တဲ့ Client-side JavaScript တွေ တွဲဖက်ပေးတဲ့ လုပ်ဆောင်ချက်ပါ။ ဒါ ကြောင့် ရံဖန်ရံခါ UI ကို မြင်နေရပေမယ့် Hydration မပြီးသေးလို့ ခလုပ်တွေ ချက်ခြင်းနှိပ်လို့ မရသေးတာ မျိုး တွေ့ရတတ်ပါတယ်။

SSG – SSR နဲ့နာမည်ဆင်တူတဲ့ Static Site Generator (SSG) ဆိုတာရှိပါသေးတယ်။ မတူကြပါဘူး။ SSG က ပရောဂျက်ကို Build လုပ်စဉ်မှာ လိုအပ်တဲ့ HTML Static Page တွေကို Generate လုပ်ထုတ်တဲ့ နည်း ပညာအမျိုးအစားဖြစ်ပါတယ်။ SSR က အဲ့ဒီလို ဖိုင်တွေ ကြိုမထုတ်ပါဘူး။ လိုတဲ့အချိန်မှ Server-side ဘက်မှာ Render လုပ်ပြီးပို့ပေးသွားတာ ဖြစ်ပါတယ်။

ဒီ (၃) နည်းလုံးက သူ့နေရာနဲ့သူ အသုံးဝင်ပါတယ်။ Static Content များတဲ့ Documentation လို Personal Portfolio တို့လို ပရောဂျက်တွေမှာ SSG နဲ့ပိုသင့်တော်နိုင်ပါတယ်။ E-Commerce ပရောဂျက် လို Dynamic Content များတဲ့ ပရောဂျက်တွေမှာ SSR နဲ့ ပိုသင့်တော်နိုင်ပါတယ်။ Social Media လို User Interaction ပိုများတဲ့ ပရောဂျက်တွေမှာတော့ SPA ကပိုသင့်တော်နိုင်ပါတယ်။ ဒါကြောင့် ပုံသေနည်းနဲ့ ဘယ်ဟာကပိုကောင်းတဲ့ ဆိုတာမျိုးမရှိဘဲ ပရောဂျက်အမျိုးအစားပေါ် မူတည်ဆုံးဖြတ်ရမှာ ဖြစ်ပါတယ်။

RSC – React Server Component ခေါ် နည်းပညာတစ်မျိုး ကျန်ပါသေးတယ်။ ဒါက Next.js တို့လို Third-Party Framework နည်းပညာ မဟုတ်ပါဘူး။ React ရဲ့ နောက်ဆုံး Version မှာပါလာတဲ့ React ရဲ့

Build-in နည်းပညာပါ။ သူ့ရဲ့ အခြေခံအလုပ်လုပ်ပုံကတော့ React Component တစ်ခုအတွက် လိုအပ်တဲ့၊ အလုပ်များတဲ့ လုပ်ဆောင်ချက်တွေ၊ Import တွေ၊ Data Fetching တွေကို Server-side ဘက်မှာ ကြိုလုပ်နိုင်စေတဲ့ နည်းပညာပါ။

SSR လို့ Static HTML ကြို Render လုပ်တဲ့နည်းပညာမျိုး မဟုတ်ပါဘူး။ ဒီလိုမြင်ကြည့်လို့ရပါတယ်။ ပုံမှန်အားဖြင့် Data Fetch လုပ်ရတဲ့ React Component တစ်ခုဟာ Client-side ဖြစ်တဲ့ Browser ကိုရောက်မှ လိုတဲ့ Data ကို လှမ်း Fetch လုပ်ယူပြီး Component ကို Render လုပ်မှာဖြစ်ပါတယ်။ Server Component ကတော့ Server-side မှာကတည်းက Data ကို တခါထဲ Fetch လုပ်ယူလာခဲ့မှာဖြစ်လို့ Client-side ကိုရောက်တဲ့အခါ Fetch လုပ်ယူစရာမလိုတော့ပါဘူး။ ပါရမယ့် Data ပါလာပြီး ဖြစ်နေပါပြီ။ ဒီသဘောနဲ့ အလုပ်လုပ်တဲ့ နည်းပညာအမျိုးအစား ဖြစ်ပါတယ်။ Next.js က သူ့ရဲ့ SSR နည်းပညာကို React Server Component နည်းပညာနဲ့ ပေါင်းစပ်ပြီး အလုပ်လုပ်ပေးပါတယ်။

ဒါတွေက နောက်ကွယ်က နည်းပညာသဘောတရားတွေကို ပြောပြတာပါ။ ရှုပ်ထွေးကျယ်ပြန့်ပါတယ်။ ဒါပေမယ့် လက်တွေ့အသုံးပြုမှုမှာတော့ အဲ့ဒီလောက် ရှုပ်ထွေးခြင်းမရှိပါဘူး။ အသုံးပြုရ လွယ်ကူပါတယ်။ Component တွေ Props တွေက အတူတူပါပဲ။ အဓိကထူးခြားချက်ဖြစ်တဲ့ Routing နဲ့ Data Fetching အကြောင်းကို ရှင်းပြပြီး သိသင့်တာ သိပြီဆိုတော့မှ ပရောဂျက်ငယ်လေးတစ်ခု တစ်ခါထဲလုပ်ပြီး လေ့လာသွားကြမှာဖြစ်ပါတယ်။

Next.js Routing

Next.js က အရင် Version တွေမှာ Page Router ခေါ် File Name ကိုအခြေခံတဲ့ Routing နည်းစနစ်ကို အသုံးပြုပါတယ်။ Route ကို သပ်သပ်ကြေညာသတ်မှတ်စရာမလိုဘဲ File Path အတိုင်း ပြန်အသုံးပြုရတဲ့ သဘောပါ။ နောက်ပိုင်း Version မှာ App Router ခေါ် အနည်းငယ်ပြောင်းလဲသွားပေမယ့် အခြေခံအားဖြင့် တူညီတဲ့ နည်းစနစ်ကို အသုံးပြုပါတယ်။ ဒီလိုပါ –

```
/              →     ./page.js
/about              →     about/page.js
/user/profile       →     user/profile/page.js
```

Route ရဲ့ Path နဲ့ ဖိုဒါ Path ကို အတူတူပေးရတာပါ။ Route တစ်ခုရဲ့ Default Page အဖြစ် `page.js` ကို အသုံးပြုရပါတယ်။ Dynamic Route တွေ သတ်မှတ်လိုရင် အခုလို သတ်မှတ်နိုင်ပါတယ်။

```
/posts/1   →   post/[id]/page.js
/posts/2   →   post/[id]/page.js
```

ဖိုဒါအမည်ကို လေးထောင့်ကွင်းနဲ့ ပေးလိုက်ရင် Dynamic Route ဖြစ်သွားပါတယ်။ နမူနာအရ 1, 2 စတဲ့ Route Parameter တွေက `id` ဖြစ်သွားမှာပါ။ Route Parameter တန်ဖိုးကိုလိုချင်ရင် သက်ဆိုင်ရာ Page မှာ အခုလို ရယူနိုင်ပါတယ်။ ဥပမာ – `post/[id]/page.js` ရဲ့ကုဒ်က ဒီလိုဖြစ်နိုင်ပါတယ်။

Next.js

```
export default function Page({ params }) {
  const { id } = params;
  ...
}
```

Route Parameters တွေကို Page Component ရဲ့ Props အဖြစ် အဆင်သင့် လက်ခံရရှိမှာဖြစ်ပါတယ်။ React Router ရဲ့ `useParams()` လို Hook တွေကနေ တစ်ဆင့်ယူစရာ မလိုတော့ပါဘူး။

Route Layout တွေကို `layout.js` နဲ့ ရေးလို့ရပါတယ်။ ဥပမာ –

```
app/
├── about/
│   └── page.js
├── user/
│   └── profile/
│       └── page.js
├── layout.js
└── page.js
```

Root ဖိုဒါအောက်မှာ `layout.js` ရှိနေပါတယ်။ ဒါကြောင့် ကျန် Page တွေအားလုံးကို ပြတဲ့အခါ ဒီ `layout.js` ထဲမှာ ထည့်ပြပေးမှာဖြစ်ပါတယ်။ `layout.js` ရဲ့ကုဒ်က ဒီလိုဖြစ်နိုင်ပါတယ်။

Next.js

```
export default function RootLayout({ children }) {
  return <body>
    <Header />
    <Sidebar />
    {children}
    <Footer />
  </body>/
}
```

`children` နေရာမှာ Page Content တွေကို ပြပေးမှာပါ။ ဒါကြောင့် ကြိုတင်သတ်မှတ်ထားတဲ့ Layout တစ်ခုနဲ့ Page တွေကို ပြလို့ရသွားပါတယ်။

`page.js` နဲ့ `layout.js` အပြင် နောက်ထပ်အသုံးဝင်တာတွေကတော့ `loading.js` နဲ့ `error.js` တို့ဖြစ်ပါတယ်။ Page ကို Loading လုပ်နေစဉ်ပြစေချင်တာတွေကို `loading.js` မှာ သတ်မှတ်ထားနိုင်ပြီး အကြောင်းအမျိုးမျိုးကြောင့် Error ရှိခဲ့ရင် ပြစေချင်တာတွေကို `error.js` မှာ သတ်မှတ်ပေးထားနိုင်ပါတယ်။

Page တွေတစ်ခုနဲ့တစ်ခု သွားဖို့အတွက် `<Link>` Component ကိုသုံးနိုင်ပါတယ်။ `href` မှာ သွားစေချင်တဲ့ Route Path ကို ပေးလိုက်ယုံပါပဲ။ ဥပမာ –

Next.js

```
import Link from "next/link";

export default function Home() {
  return <div>
    <Link href="/about">About</Link>
  </div>
}
```

Home Page Component မှာ About Page ကို သွားလို့ရတဲ့ Link တစ်ခုထည့်လိုက်တာပါ။ Component မသုံးဘဲ Function နဲ့သွားချင်ရင် `useRouter` ကို သုံးနိုင်ပါတယ်။ ဒီလိုပါ –

Next.js

```
"use client"

import { useRouter } from "next/navigation";

export default function Home() {
  const router = useRouter();

  return (
    <button type="button" onClick={() => router.push('/about')}>
      About
    </button>
  )
}
```

ဟိုးအပေါ်ဆုံးက `"use client"` ကြေညာချက်လေးကို သတိပြုပါ။ ဒီအကြောင်း ခဏနေမှ ဆက်ပြောပါမယ်။ နောက်ထပ်အသုံးဝင်နိုင်တဲ့လုပ်ဆောင်ချက်ကတော့ Route Group ဖြစ်ပါတယ်။ တချို့ ဖိုဒါခွဲပြီး Organize လုပ်ချင်ပေမယ့် Route မှာ ထည့်အသုံးစေချင်ရင် ဒီလိုသတ်မှတ်ထားလို့ ရပါတယ်။

- `(user)/account/page.js` → `/account`
- `(user)/settings/page.js` → `/settings`

ဖိုဒါအမည်ကို ဝိုက်ကွင်းအဖွင့်အပိတ်ထဲ ထည့်ပေးလိုက်တာပါ။ နမူနာအရ `(user)` အမည်နဲ့ ဖိုဒါခွဲပြီးဖိုင်တွေကို စုစည်းသိမ်းဆည်းပေမယ့် Route မှာ အဲ့ဒီ `(user)` ကိုထည့်ပေးစရာမလိုတဲ့သဘောကို တွေ့ရမှာပါ။ ဒီသဘောကို Expo Router မှာလည်း တွေ့ခဲ့ကြပြီး ဖြစ်ပါတယ်။ Expo Router ကနောက်မှ ပေါ်တာဖြစ်လို့ သူက Next.js ကို နမူနာယူပုံရပါတယ်။ ဒီလောက်ဆိုရင် Routing ပိုင်းက ရပါပြီ။

Server Component vs. Client Component

Next.js က React Server Component ကို အသုံးပြုအလုပ်လုပ်ပါတယ်။ ဒါကြောင့် Component တွေရေးတဲ့အခါ Server Component အဖြစ်ရေးလို့ရသလို၊ Client Component အဖြစ်လည်း ရေးလို့ရပါတယ်။ ကျယ်ပြန့်တဲ့အကြောင်းအရာတစ်ခု ဖြစ်ပေမယ့် လိုရင်းကို ဒီလိုမှတ်နိုင်ပါတယ်။

1. Next.js မှာ Default အနေနဲ့ Component တိုင်းဟာ Server Component တွေဖြစ်ကြပါတယ်။
2. `useState`, `useEffect` စတဲ့ Hook တွေနဲ့ `onClick`, `onSubmit` တို့လို Event တွေကို Server Component မှာသုံးလို့မရပါဘူး။ Client Component မှာပဲ သုံးလို့ရပါတယ်။ Client Component တွေရေးသားဖို့ Component Function ရဲ့ အပေါ်ဆုံးမှာ `"use client"` လို့ ကြိုတင်ကြေညာပေးရပါတယ်။
3. Server Component ထဲမှာ Client Component ကို Child အနေနဲ့ ထည့်သုံးလို့ ရပါတယ်။ Client Component ထဲမှာ Server Component ကို ထည့်သုံးလို့ မရပါဘူး။ ရအောင်လုပ်တဲ့ ခြွင်းချက်နည်းလမ်းတချို့ ရှိပေမယ့် အခြေခံမူအားဖြင့် မရဘူးလို့ပဲ မှတ်ထားသင့်ပါတယ်။

ဘာကြောင့်လဲ၊ နောက်ကွယ်မှာဘယ်လိုတွေ အလုပ်လုပ်သွားတာလိုဆိုတဲ့ကိစ္စတွေက တော်တော်လေး ကျယ်ပြန့်ပြီး နားလည်ရခက်ပါတယ်။ နောက်တစ်ဖြည်းဖြည်း မြင်လာပါလိမ့်မယ်။ လောလောဆယ် သူ့နေရာနဲ့သူ မှန်ကန်အောင် အသုံးပြုနိုင်ဖို့ ဒီသုံးချက်မှတ်ထားရင် လုံလောက်ပါတယ်။

Data Fetching

Data Fetching မှာ Server-side Data Fetching နဲ့ Client-side Data Fetching ဆိုပြီး နှစ်မျိုးရှိပါတယ်။ Client-side Data Fetching က ရှေ့ပိုင်းမှာ လေ့လာခဲ့တဲ့နည်းတွေနဲ့ သိပ်မကွာလို့ အများကြီး ထပ်ပြော စရာ မရှိပါဘူး။ Server-side Data Fetching အကြောင်းပဲကြည့်ရပါမယ်။

နှစ်မျိုးဖြစ်နိုင်ပါတယ်။ API ရှိနေပြီး API Data ကို Fetch လုပ်ယူခြင်းနဲ့၊ Database ထဲက Data ကို တိုက်ရိုက် ထုတ်ယူခြင်းဖြစ်ပါတယ်။ ဒါဟာ Server Component ရဲ့အားသာချက်ပါပဲ။ လိုအပ်ရင် Database ထဲက Data နဲ့တိုက်ရိုက်အလုပ်လုပ်နိုင်ပြီး API Fetch လုပ်ရရင်တောင် Server-side ဘက်မှာ ကတည်းက ကြို Fetch လုပ်လာမှာဖြစ်လို့ Client က သူ့ဆီရောက်မှ Data ယူနေရတာမျိုး လုပ်စရာ မလို တော့ပါဘူး။ ဒီလိုပါ –

Next.js

```
async function fetchData(id) {
  const res = await fetch(`${api}/${id}`);
  return await res.json();
}

export default async function Page({ params }) {
  const { id } = params;
  const data = await fetchData();

  return <div>{/* data */}</div>;
}
```

ဒီကုဒ်ဟာ Server-side ဘက်မှာ တစ်ခါထဲ Data Fetch လုပ်ပြီးမှ Data ပါဝင်ပြီး Render လုပ်ပြီးသား ရလဒ်ကို ပြန်ပေးမယ့် ကုဒ်ဖြစ်ပါတယ်။ ရိုးရိုး `fetch()` ကိုပဲသုံးရပါမယ်။ ဒါပေမယ့် Next.js က `fetch()` ကို Cache လုပ်တာတို့၊ Revalidate လုပ်တာတို့လို အခြေခံ Data Fetching ပိုင်း လိုအပ်ချက် တွေ ထပ်ဖြည့်ပြီး ပြင်ထားပါတယ်။ နည်းပညာသစ်၊ အမည်သစ်နဲ့ မလုပ်ဘဲ JavaScript ရဲ့ Build-in နည်း ပညာကို ပြင်လိုက်လို့ မလုပ်သင့်တာကို လုပ်တယ်ဆိုပြီး ဝေဖန်တဲ့သူတွေလည်း အများကြီးရှိကြပါတယ်။ ဒီနေရာမှာ လိုရင်းကို နားလည်ရမှာကတော့ ရေးတဲ့အခါ `fetch()` နဲ့ ရေးရိုးရေးစဉ်အတိုင်းပဲရေးရပြီး

နောက်ကွယ်မှာ Cache တွေ Revalidate တွေ အလိုအလျှောက် အလုပ်လုပ်သွားတယ်လို့ နားလည်ရမှာ ပါ။

အကယ်၍ Database ထဲက Data နဲ့ တိုက်ရိုက်အလုပ်လုပ်ရင်တော့ အခုလိုဖြစ်နိုင်ပါတယ်။

Next.js

```
// db.js
import { cache } from "react";

export const getItem = cache(async (id) => {
  const post = await prisma.post.findUnique({ id });
  return post;
});

// posts/page.js
export default async function Post({ params }) {
  const { id } = params;
  const data = await getItem(id);
  ...
}

// profile/page.js
export default async function Profile({ params }) {
  const { id } = params;
  const data = await getItem(id);
  ...
}
```

ရေးရိုးရေးစဉ်အတိုင်း ရေးရင်လည်း ရနိုင်ပေမယ့် `cache()` Function တစ်ဆင့်ခံပြီးရေးထားတာကို သတိပြုပါ။ ဒီနည်းနဲ့ Data ကို Database ထဲကနေတိုက်ရိုက်ယူ အလုပ်လုပ်ယုံသာမက လိုအပ်တိုင်း ထပ်ခါထပ်ခါ ပြန်ယူစရာမလိုအောင်လည်း Cache ပေးနိုင်သွားမှာဖြစ်ပါတယ်။

ထူးခြားချက်အနေနဲ့ Next.js မှာ Route Handler လို့ခေါ်တဲ့ သဘောသဘာဝတစ်ခုလည်း ပါဝင်ပါတယ်။ ပုံမှန်အားဖြင့် Route တွေသတ်မှတ်ဖို့အတွက် Route အမည်ဖိုဒါထဲမှာ `page.js` ကို အသုံးပြုရပါတယ်။ `page.js` အစား `route.js` ကို အစားထိုး အသုံးပြုနိုင်ပါတယ်။ ဥပမာ –

```
app/
├── api/
│   └── route.js
├── user/
│   └── profile/
│       └── page.js
├── layout.js
└── page.js
```

နမူနာအရ `api/` ထဲမှာရှိနေတာ `page.js` မဟုတ်တော့ပါဘူး။ `route.js` ဖြစ်သွားပါတယ်။ သူ့ထဲမှာ ရေးထားတဲ့ ကုဒ်က အခုလိုဖြစ်နိုင်ပါတယ်။

Next.js

```
export default async function GET(req, { params }) {
  const { id } = params;
  const post = await prisma.user.findUnique({ id });

  return Response.json(post);
}
```

ဒါဟာ Database ထဲက Data ကိုထုတ်ယူပြီး JSON အနေနဲ့ Response ပြန်ပေးလိုက်တာပါပဲ။ Page တွေ Template တွေမပါတော့ပါဘူး။ ဒါကြောင့် Client-side ကနေ Fetch လုပ်ဖို့လိုတဲ့ Data တွေရှိလာတဲ့အခါ အခုလို Server-side ကနေ Route Handler နဲ့ ဖွင့်ထားပေးလို့ ရနိုင်သွားပါတယ်။

သတိပြုရမယ့်အချက်က Route ဖိုဒါတစ်ခုမှာ `route.js` ရှိရင် `page.js` ရှိလို့မရပါဘူး။ `page.js` ရှိ

ရင် `route.js` ရှိလို့မရပါဘူး။ နှစ်ခုထဲက တစ်ခုကို ရွေးသုံးပေးရမှာ ဖြစ်ပါတယ်။ဒီလောက်ဆိုရင် သိသင့်တာတွေစုံပါပြီ။ နမူနာ ပရောဂျက်လေးတစ်ခုလောက်လုပ်ပြီး စမ်းကြည့်ကြပါမယ်။

SSR Project

နမူနာပရောဂျက်လုပ်တဲ့အခါ UI အတွက် TailwindCSS နဲ့ shadcn/ui ခေါ် Component Library တစ်ခုတို့ကို ပူးတွဲပြီး အသုံးပြုသွားပါမယ်။ TailwindCSS ဟာ CSS ကောင်းကောင်းရပြီး Bootstrap လောက်သုံးတတ်ရင် သိပ်အများကြီး ထပ်လေ့လာစရာမလိုဘဲ အသင့်ကောက်သုံးလို့ရနိုင်တဲ့ နည်းပညာဖြစ်ပါတယ်။ ဒါကြောင့် သပ်သပ်ကြိုပြီး ရှင်းပြမနေတော့ဘဲ လက်တွေ့လုပ်ရင်းနဲ့ တွဲကြည့်သွားကြပါမယ်။

အရင်ဆုံး Next.js ပရောဂျက်တစ်ခုကို အခုလို အစပြုလိုက်ပါ။

```
npx create-next-app next-movies
```

```
✔ Would you like to use TypeScript? … No / Yes
✔ Would you like to use ESLint? … No / Yes
✔ Would you like to use Tailwind CSS? … No / Yes
✔ Would you like to use `src/` directory? … No / Yes
✔ Would you like to use App Router? (recommended) … No / Yes
✔ Would you like to customize the default import alias (@/*)? … No / Yes
✔ What import alias would you like configured? … @/*
```

`next-movies` အမည်နဲ့ ပရောဂျက်သစ်စလိုက်တာပါ။ နောက်ဆုံးထွက် ရုပ်ရှင်ဇာတ်ကား စာရင်းကို ဖော်ပြပြီး ကိုယ်စိတ်ဝင်စားတဲ့ ဇာတ်ကားအကြောင်း အသေးစိတ်ကြည့်လို့ရတဲ့ ပရောဂျက်ပါ။ ဇာတ်ကားတွေကို Genres အလိုက် ခွဲကြည့်လို့ရမှာဖြစ်သလို Search နဲ့ ရှာကြည့်လို့ရအောင်လည်း လုပ်ပေးမှာပါ။

ပရောဂျက် Create လုပ်စဉ်မှာ မေးခွန်းတချို့ မေးပါလိမ့်မယ်။ TypeScript သုံးမလားဆိုရင် No လို့ပဲ ပြောရမှာပါ။ ESLint သုံးမလားဆိုရင်လည်း No လို့ပဲပြောရမှာပါ။ ဒီနည်းပညာ နှစ်ခုလုံးက သုံးနိုင်ရင်ကောင်းပါတယ်။ အခုမှလေ့လာဆဲဖြစ်လို့ အတတ်နိုင်ဆုံး ရှင်းချင်တဲ့အတွက် မသုံးတာပါ။ Tailwind CSS သုံးမှာလားကိုတော့ Yes လို့ ဖြေပေးရပါမယ်။

`src/` directory သုံးမလားမေးတာကိုတော့ No လို့ပဲဖြေပေးရပါမယ်။ ကုဒ်တွေကို `app` ဖိုဒါထဲမှာပဲ

တိုက်ရိုက်ရေးကြပါမယ်။ `src/` တစ်ဆင့်ခံမနေတော့ပါဘူး။ App Router သုံးမှာလားကိုတော့ Yes လို့ဖြေပေးရပါမယ်။ နည်းပညာဟောင်းဖြစ်တဲ့ Page Router ဆိုတာရှိပါတယ်၊ မသုံးသင့်တော့ပါဘူး။ App Router ပဲ သုံးကြပါမယ်။

`import alias` သုံးမှာလားဆိုတဲ့မေးခွန်းကိုတော့ Yes လို့ဖြေပေးရပါမယ်။ ကျန်မေးခွန်းတွေက ကိုယ်ရွေးစရာမလိုဘဲ သူ့ဘာသာကိုယ်လိုအပ်တာကို ရွေးပြီးဖြစ်ပေမယ့် ဒီတစ်ခုတော့ ကိုယ့်ဘာသာ Yes ကို ရွေးပေးရမှာပါ။ ကုဒ်တွေ Import လုပ်တဲ့အခါ `../../utils/db.js` စသည်ဖြင့် လိုချင်တဲ့ဖိုင်ရအောင် အဆင့်ဆင့်ပြန်ဝင်ရ ထွက်ရတာတွေ လုပ်စရာမလိုဘဲ `@/utils/db.js` စသည်ဖြင့် ရှေ့ဆုံးက `@` ကို Root သင်္ကေတအဖြစ် သုံးပြီး Import လုပ်လို့ရစေဖို့အတွက် ဖြစ်ပါတယ်။ ဒါကြောင့် နောက်တစ်ဆင့်မှာ alias configured လုပ်ခိုင်းတဲ့အခါ ဘာမှမပြောင်းတော့ဘဲ သူပေးထားတဲ့အတိုင်းပဲ Enter နှိပ်ပေးလိုက်ပါ။

ဒါတွေအားလုံးပြည့်စုံတဲ့အခါ `next-movies` အမည်နဲ့ ပရောဂျက်ဖိုဒါသစ်တစ်ခု ရသွားပါပြီ။ ကုဒ်တွေမရေးလာခင် shadcn/ui Library ကို ဆက်လက် ထည့်သွင်းကြပါမယ်။ `next-movies` ဖိုဒါထဲကို သွားလိုက်ပါ။

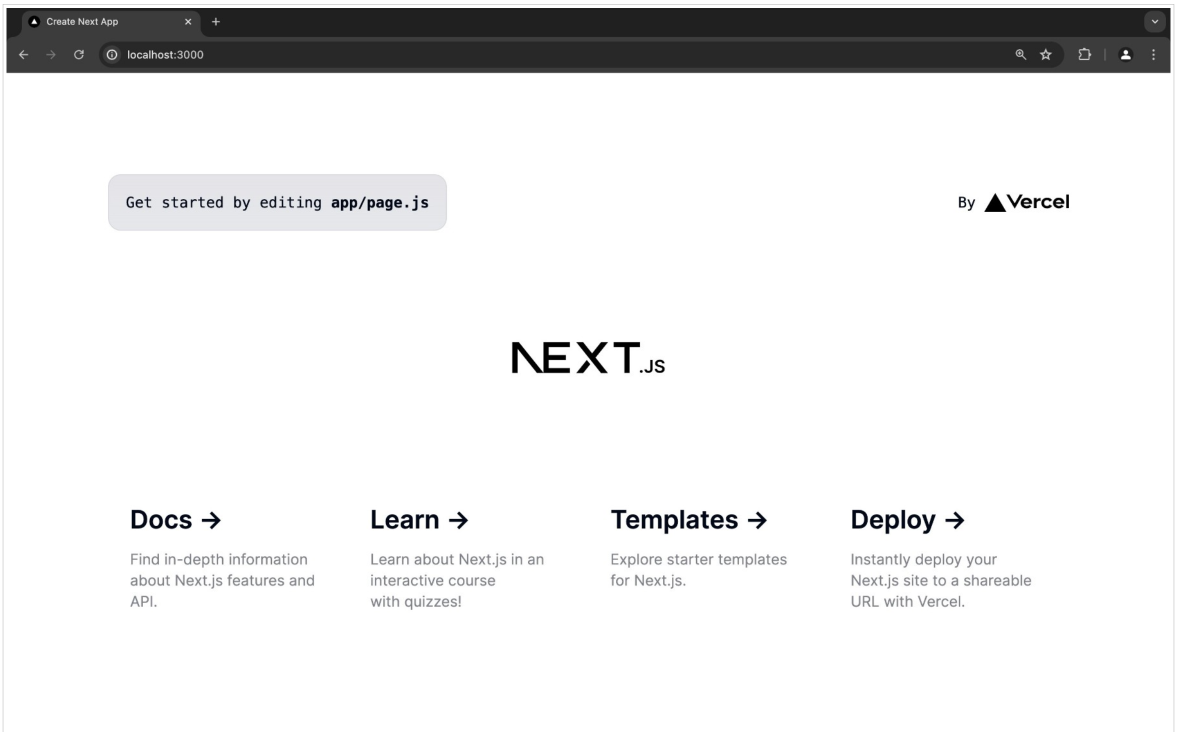
```
cd next-movies
```

ပြီးတဲ့အခါ shadcn/ui ကို အခုလို Install လုပ်ပေးပါ။

```
npx shadcn-ui@latest init

✔ Which style would you like to use? › Default
✔ Which color would you like to use as base color? › Slate
✔ Would you like to use CSS variables for colors? … no / yes
```

npm မဟုတ်ပါဘူး။ `npx` နဲ့ သူ့ရဲ့ `init` Script ကို လှမ်း Run လိုက်တာပါ။ သူလည်း မေးခွန်းတချို့ မေးပါလိမ့်မယ်။ ပြောင်းစရာမလိုပါဘူး။ သူပေးထားတဲ့အတိုင်းပဲ အကုန်ရွေးလိုက်လို့ ရပါတယ်။ အခုလို Run ပြီး စမ်းကြည့်လို့ရပါပြီ။

```
npx next
```



ပထမဆုံးအနေနဲ့ ပရောဂျက်ရဲ့ ဖွဲ့စည်းပုံကို အခုလိုပုံစံဖြစ်အောင် လုပ်ပေးပါ။

```
next-movies
├── node_modules/
├── app/
│   ├── favicon.ico
│   ├── globals.css
│   ├── layout.js
│   ├── page.js
│   ├── genres/
│   │   └── [name]/
│   │       └── [id]/
│   │           └── page.js
│   ├── movie/
│   │   └── [id]/
│   │       └── page.js
```

```
│     └── search/
│         └── page.js
├── components/
│     ├── Header.js
│     ├── Movies.js
│     ├── Persons.js
│     └── Sidebar.js
├── lib/
│     └── utils.js
├── README.md
├── ...
├── public/
└── tailwind.config.js
```

`lib/utils.js` ဆိုတာ `shadcn/ui` ကထည့်ပေးလိုက်တဲ့ ဖိုင်နဲ့ဖိုဒါဆိုတာကိုသတိပြုပါ။ Next.js ရဲ့ မူလဖိုဒါ မဟုတ်သလို ကိုယ့်ဘာသာလည်း ထပ်ထည့်စရာမလိုပါဘူး။ ကိုယ့်ဘာသာထပ်ထည့်ရမှာတွေက `components` ဖိုဒါထဲက `Header.js`, `Movies.js`, `Persons.js`, `Sidebar.js` တို့ဖြစ်ပါတယ်။

ပြီးတဲ့အခါ `app` ဖိုဒါထဲမှာ Route ဖိုဒါတွေဖိုင်တွေ ထပ်တိုးထားပါတယ်။ `genres/[name]/[id]/page.js` မှာ Dynamic Route ဖိုဒါ နှစ်ဆင့်ပေးထားတာကို သတိပြုပါ။ ရုပ်ရှင်ဇာတ်ကားတွေကို Action, Drama, Comedy စသည်ဖြင့် Genres အလိုက် ဒီ Page မှာပြချင်တာပါ။ `movie/[id]/page.js` နဲ့ ရုပ်ရှင်တစ်ကားချင်း အသေးစိတ်အချက်အလက်တွေ ပြဖို့အတွက် Page တစ်ခု ထည့်ထားပါတယ်။ `search/page.js` ကတော့ Search Result တွေပြဖို့ဖြစ်ပါတယ်။

ဆက်လက်ပြီး `components/Header.js` မှာ အခုလို ရေးပေးပါ။

Next.js - app/components/Header.js

```
import Link from "next/link";

export default async function header() {
  return (
    <nav className="flex justify-between items-center border
                    rounded border-slate-300 px-4 py-2 my-4">

      <h1 className="font-bold text-lg">
        <Link href="/">Next Movies</Link>
      </h1>
    </nav>
  );
}
```

<nav> Element ကို Return ပြန်ပေးပါတယ်။ အဲ့ဒီ Nav Element ကို Header Bar / App Bar ပုံစံရအောင် Tailwind CSS နဲ့ `className` တွေပေးထားပါတယ်။ `flex` နဲ့ `flex-box` ဖြစ်ကြောင်းသတ်မှတ်ပါတယ်။ `justify-between` ကိုသုံးပြီး `justify-contents: space-between` လုပ်ဆောင်ချက်ရအောင် သတ်မှတ်ပေးထားပါတယ်။ `items-center` နဲ့ `align-items: center` လုပ်ဆောင်ချက်ရအောင် သတ်မှတ်ပေးထားပါတယ်။ CSS ရရင် ဒီ `class` တွေရဲ့ အဓိပ္ပါယ်ကသိပ်ခေါင်းစားစရာမလိုဘဲ အလိုအလျှောက် ပေါ်လွင်နေမှာဖြစ်ပြီး၊ CSS မရရင်တော့ နားလည်ရခက်နေပါလိမ့်မယ်။ ဒါကြောင့် Tailwind CSS ကို အလွယ်တကူလေ့လာ အသုံးပြုနိုင်သလား မသုံးနိုင်ဘူးလားဆိုတာ CSS ရမရပေါ်မှာ မူတည်နေမှာ ဖြစ်ပါတယ်။

`border` ပါပြီး `rounded` လို့ပြောထားပါတယ်။ ရှင်းပြနေစရာမလိုဘဲ ဘယ်လိုပုံစံဖြစ်မလဲ ခန့်မှန်းကြည့်လို့ ရနိုင်ပါတယ်။ `px`, `py`, `my` Class တွေကတော့ Bootstrap Utilities Class တွေနဲ့ သွားတူပါတယ်။ ဒါကြောင့် Bootstrap ရရင် Tailwind ကအလွယ်တကူ ဆက်အသုံးပြုနိုင်တယ်လို့ ပြောတာပါ။

ဖြည့်စွက်ထူးခြားချက်ဆိုလို့ `border-slate-500` နဲ့ Border Color သတ်မှတ်ထားခြင်းဖြစ်ပါတယ်။ Tailwind မှာ `bg-blue-100`, `text-green-800` စသည်ဖြင့် Border Color, Background Color, Text Color တို့ကို သတ်မှတ်နိုင်ပါတယ်။ ပါဝင်တဲ့ Class တွေ Color Code တွေများလို့ လက်တမ်းအလွတ်

ရဖို့တော့ မလွယ်ပါဘူး။ အသုံးပြုနေတဲ့ Code Editor မှာ Tailwind CSS Extension ထည့်သွင်းထားရမှာ ဖြစ်ပါတယ်။ ဒါဆိုရင် Color Code တွေအပါအဝင် များပြားလှတဲ့ Tailwind Class တွေကို Autocomplete လုပ်ပေးနိုင်ပြီး အဆင်ပြေသွားပါလိမ့်မယ်။

`<h1>` Element ကိုတော့ စာလုံးကြီးကြီး ထင်ထင်ရှားရှားပြချင်လို့ `font-bold` နဲ့ `text-lg` လို့ သတ်မှတ်ပေးထားပါတယ်။ တခြားဘာမှ ထွေထွေထူးထူး မပါသေးပါဘူး။ ဒီလောက်နဲ့အရင်စပြီး နောက်မှ လိုအပ်တာတွေ ဖြည့်စွက်ကြပါမယ်။

နောက်တစ်ဆင့်မှာ shadcn/ui ရဲ့ `Button` Component ကို စတင်ထည့်သွင်း အသုံးပြုကြမှာ ဖြစ်ပါတယ်။ shadcn/ui ကို စောစောက ကြိုတင် Setup လုပ်ထားပေမယ့် လိုအပ်တဲ့ Component တွေ မပါသေးပါဘူး။ ကိုယ်လိုတာကို ရွေးပြီး ထည့်သွင်းအသုံးပြုရပါတယ်။ ဒါကြောင့် `Button` Component ကို အခုလို ထည့်သွင်းပေးပါ။

```
npx shadcn-ui@latest add button
```

ဒီလိုထည့်သွင်းလိုက်တဲ့အခါ `components/ui/Button.jsx` ဆိုတဲ့အမည်နဲ့ Component ဖိုင်တစ်ခု ရရှိသွားတာကို တွေ့ရပါလိမ့်မယ်။ ဒါဟာ `shadcn/ui` ရဲ့ ထူးခြားချက်ဖြစ်ပါတယ်။ MUI လို Component Framework မှာ လိုချင်တဲ့ Component ကို ပင်မ Package ကနေ Import လုပ်ယူပြီး သုံးရပါတယ်။ shadcn/ui က အဲ့လိုမဟုတ်ပါဘူး။ Component ကို ကိုယ့်ပရောဂျက်ထဲမှာ ဖိုင်တစ်ခုအနေနဲ့ ထည့်ပေးလိုက်မှာပါ။

ဒါကြောင့် ဥပမာ MUI မှာဆိုရင် မူရင်း Component ကို တိုက်ရိုက် ပြင်လို့မရပါဘူး။ Component က သူ့ Package ထဲမှာ သူ့ဘာသာ ရှိနေတာပါ။ ပြင်လို့ရရင်တောင် မပြင်သင့်ပါဘူး။ shadcn/ui Component တွေကတော့ သူ့ Component ကို လှမ်းချိတ်သုံးတာမျိုး မဟုတ်တော့ဘဲ၊ Component က ကိုယ့်ပရောဂျက်ရဲ့ အစိတ်အပိုင်းတစ်ခု ဖြစ်သွားလို့ လိုအပ်သလို တိုက်ရိုက်ပြင်ဆင်အသုံးပြုလို့ရသွားပါတယ်။ ဒီနည်းကို အခုနောက်ပိုင်း လူတွေအတော် သဘောကျနှစ်သက်နေကြပါတယ်။

ဆက်လက်ပြီး `components/Sidebar.js` မှာ အခုလို ရေးပေးပါ။

Next.js - app/components/Sidebar.js

```
import Link from "next/link";
import { Button } from "@/components/ui/button";

export default async function sidebar() {
  return (
    <aside className="w-[220px] flex flex-col gap-1">
      <Button
        className="justify-start"
        variant="outline"
        asChild>
        <Link href="/genres/action/1">Action</Link>
      </Button>
      <Button
        className="justify-start"
        variant="outline"
        asChild>
        <Link href="/genres/drama/1">Drama</Link>
      </Button>
    </aside>
  );
}
```

<aside> Element တစ်ခုကို ပြန်ပေးတဲ့ Component ဖြစ်ပါတယ်။ စောစောကထည့်လိုက်တဲ့ <Button> ကို Import လုပ်ပြီးသုံးထားပါတယ်။ အဲ့ဒီလိုသုံးတဲ့အခါ @ သင်္ကေတကို Root အဖြစ်နဲ့သုံးပြီး Import လုပ်ထားတာကို တွေ့ရပါလိမ့်မယ်။

<aside> အတွက် `flex` နဲ့ `flex-box` လုပ်ထားပြီး `flex-col` နဲ့ `flex-direction: column` လုပ်ဆောင်ချက် ရအောင် ထည့်ထားပါတယ်။ ပြီးတော့ `w-[ ]` နဲ့ ကိုယ်လိုချင်တဲ့ Width အတိအကျပေးထားပါတယ်။ Tailwind မှာ `m-1`, `m-2`, `p-3`, `p-4` စသည်ဖြင့် Margin တွေ Padding တွေအတွက် သူကြိုတင်သတ်မှတ်ပေးထားတဲ့ Size တွေကိုသုံးလို့ရသလို `w-20`, `w-40`

စသည်ဖြင့် သူကြိုတင်သတ်မှတ်ပေးထားတဲ့ Width, Height တွေကိုလည်း သုံးလို့ရပါတယ်။ သူကြိုတင်သတ်မှတ်ပေးထားတဲ့ Size က အဆင်မပြေရင် လေးထောင့်ကွင်းနဲ့ ကိုယ်လိုချင်တဲ့ Size အတိအကျပေးလို့ ရပါတယ်။ ဒီလိုလုပ်ဆောင်ချက်တွေက Bootstrap ရဲ့ Utilities Class တွေထက် အများကြီး ပိုပြည့်စုံ သာလွန်သွားစေတာပါ။

`<Button>` ထဲမှာ `<Link>` နဲ့ `/genres/[name]/[id]` Route ကို သွားခိုင်းထားပါတယ်။ `<Button>` မှာ `asChild` prop ထည့်ထားတာက သူ့ထဲက `<Link>` ကို `<Button>` Size ရှိသလောက် အပြည့်ပြစေချင်လို့ ထည့်ထားတာပါ။ မထည့်ရင်လည်း ဖော်ပြပုံမှာ သိပ်အပြောင်းအလဲ ရှိမှာမဟုတ်ပါဘူး။ ဒါပေမယ့် နှိပ်တဲ့အခါ Button အစွန်တွေကို နှိပ်လို့ရမှာ မဟုတ်ဘဲ Link စာသားပေါ် တည့်တည့်နှိပ်ရပါလိမ့်မယ်။ ဒါကြောင့် ပိုအဆင်ပြေအောင် `asChild` ထည့်ပေးထားတာပါ။

ဆက်လက်ပြီး app အောက်တည့်တည့်က အဓိက Page ဖြစ်တဲ့ `page.js` မှာ မူလကရှိနေတဲ့ နမူနာတွေကို ဖျက်ပစ်လိုက်ပြီး အခုလိုရေးပေးပါ။

Next.js - app/page.js

```
export default async function Home() {
    return <div>Home</div>;
}
```

ဘာမှမပါသေးပါဘူး။ နောက်မှ အသေးစိတ် ဆက်ကြပါမယ်။ ဆက်လက်ပြီး `layout.js` မှာလည်း သူ့နဂိုပါတဲ့ နမူနာတွေကို ဖျက်ပစ်လိုက်ပြီး အခုလို ရေးပေးပါ။

Next.js - app/layout.js

```
import { Inter } from "next/font/google";
import "./globals.css";

import Sidebar from "@/components/Sidebar";
import Header from "@/components/Header";

const inter = Inter({ subsets: ["latin"] });
```

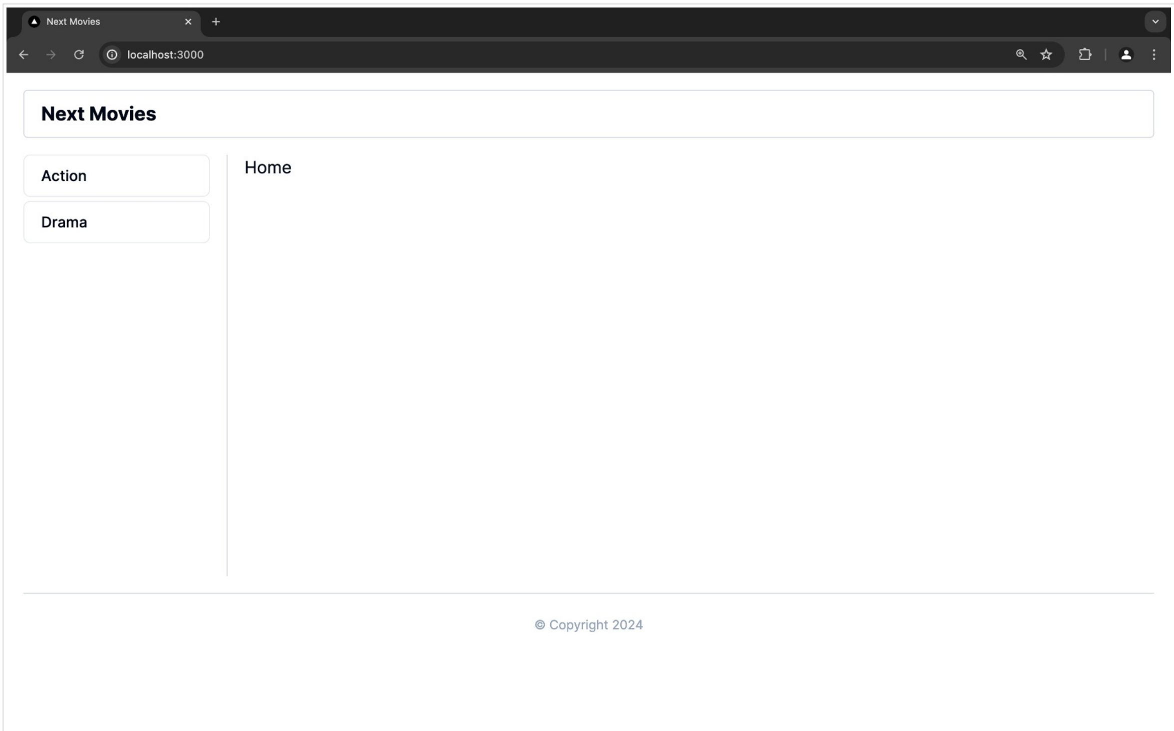
```
export const metadata = {
  title: "Next Movies",
  description: "RSD25 Tutorial Project",
};

export default function RootLayout({ children }) {
  return (
    <html lang="en">
      <body className={inter.className}>
        <div className="mx-4 m-auto">
          <Header />

          <section className="flex gap-4 min-h-[500px]">
            <Sidebar />

            <main className="border-l border-slate-300 px-4">
              {children}
            </main>
          </section>

          <footer className="text-center py-4 text-slate-400
                            mt-4 border-t border-slate-300">
            <small className="text-xs">&copy; Copyright 2024</small>
          </footer>
        </div>
      </body>
    </html>
  );
}
```

ဒီပရောဂျက်အတွက် Main Template ဖိုင်ဖြစ်ပါတယ်။ HTML Document အပြည့်အစုံပြန်ပေးထားပါတယ်။ <head> တော့မပါပါဘူး။ <head> မှာရေးလေးရှိတဲ့ HTML တွေကို metadata အနေနဲ့ ရေးထားပေးလို့ရပါတယ်။ Inter ဆိုတဲ့ Font ကို မူလနမူနာမှာကတည်းက Import လုပ်ထားတာ ရှိပါတယ်။ ဆက်

သုံးချင်တဲ့အတွက် ပြန်ထည့်ထားပါတယ်။

စောစောကရေးလိုက်တဲ့ Header နဲ့ Sidebar တို့ကို Import လုပ်ထားပါတယ်။

သူက Layout ဖြစ်တဲ့အတွက် `children` Prop ပါဝင်ပါတယ်။ နမူနာအရ `children` တွေကို `<main>` Element ထဲမှာ ပြခိုင်းထားပါတယ်။ ဒါကြောင့် Route ပြောင်းလိုက်ရင် ပြောင်းသွားတဲ့ Page Content က `<main>` ထဲမှာ လာပြပေးမှာပါ။ ကျန် Header တွေ Sidebar တွေ `<footer>` တွေက အပြောင်းအလဲရှိမှာ မဟုတ်ပါဘူး။ အခုနေ စမ်းကြည့်လိုက်ရင် ရလဒ်က အခုလိုဖြစ်မှာပါ။



Content တွေမပါသေးတဲ့ Layout အထိုင်တစ်ခုကို ရရှိသွားတာပါ။

ဆက်လက်ပြီး Data Fetching လုပ်ဆောင်ချက်ထည့်ကြပါမယ်။ လိုအပ်တဲ့ Data တွေ ရဖို့အတွက် **TMDB** (The Movie Database) ခေါ် ရုပ်ရှင်ဇာတ်လမ်း အချက်အလက် API ကို အခမဲ့ပေးတဲ့ Service ကို သုံးကြပါမယ်။ အောက်ပါလိပ်စာမှာ Account ဆောက်ဖို့လိုပါတယ်။

TMDB – https://www.themoviedb.org/

Account ဆောက်ပြီးနောက် Profile Settings မှာ API Access Token ရဖို့အတွက် ဖောင်တွေဘာတွေ ဖြည့်ရပါလိမ့်မယ်။ ဒီအဆင့်တွေကိုတော့ သက်ဆိုင်ရာ Website မှာပဲ ကိုယ့်ဘာသာဆက်လုပ်လိုက်ပါ။ ရလာတဲ့ Access Token ကို ကူးယူပြီး လက်ရှိ ပရောဂျက်ဖိုဒါထဲမှာ `.env` ဖိုင်နဲ့ အခုလို သိမ်းထားလိုက်ပါ။

.env

```
TMDB_TOKEN="eyJhbGciOiJIUzI1NiJ9 … AevGc"
```

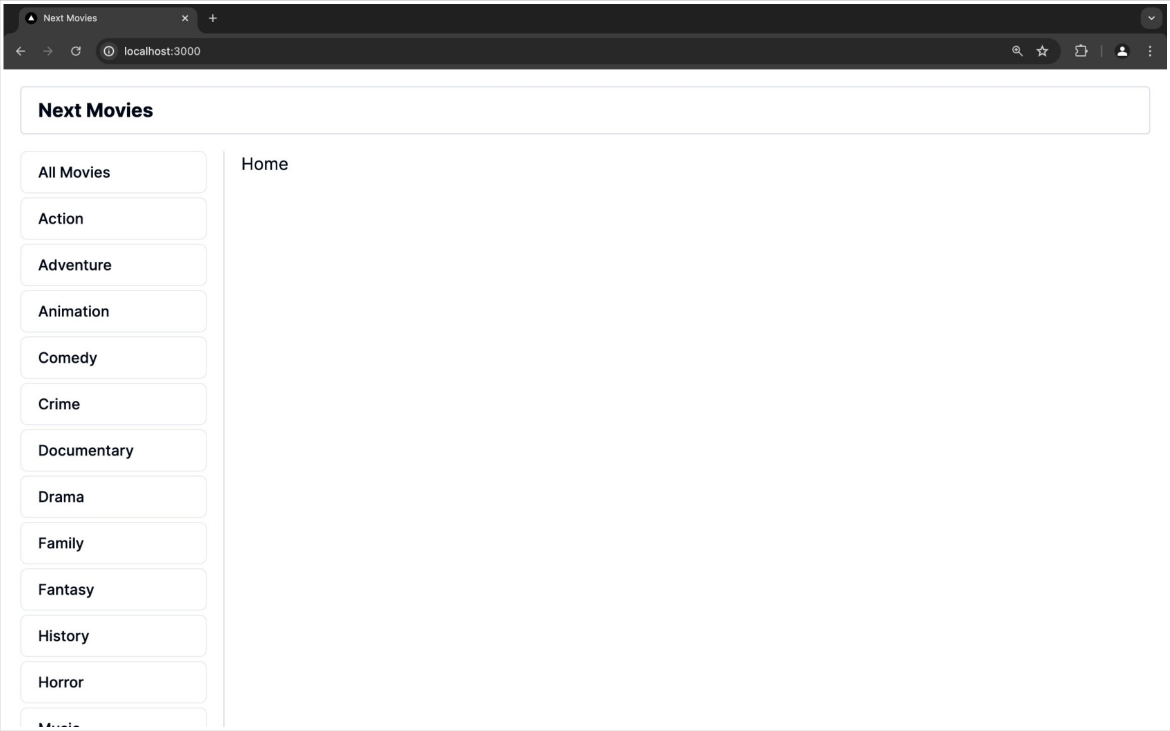
ပြီးတဲ့အခါ လိုအပ်တဲ့ Data ယူတဲ့အလုပ်တွေကို Sidebar ကနေစလုပ်ကြပါမယ်။ `components/Sidebar.js` ကို အခုလို ပြင်ရေးလိုက်ပါ။

Next.js - app/components/Sidebar.js

```
import Link from "next/link";
import { Button } from "@/components/ui/button";

const token = process.env.TMDB_TOKEN;

async function fetchGenres() {
  const res = await fetch("https://api.themoviedb.org/3/genre/movie/list", {
    headers: {
      Authorization: `Bearer ${token}`,
    },
  });

  return await res.json();
}

export default async function sidebar() {
  const { genres } = await fetchGenres();
```

```
  return (
    <aside className="w-[220px] flex flex-col gap-1">
      <Button
        className="justify-start"
        variant="outline"
        asChild>
        <Link href="/">All Movies</Link>
      </Button>
      {genres.map(genre => {
        return (
          <Button
            key={genre.id}
            className="justify-start"
            variant="outline"
            asChild>
            <Link href={`/genres/${genre.name}/${genre.id}`}>
              {genre.name}
            </Link>
          </Button>
        );
      })}
    </aside>
  );
}
```

fetchGenres() Function တစ်ခုကြိုရေးလိုက်ပါတယ်။ Access Token နဲ့အတူ TMDB API ကို ဆက်သွယ်ပြီးပြန်ရလာတဲ့ Data တွေကို ပြန်ပေးတာပါ။ ပြီးတဲ့အခါ Component ထဲမှာ အဲ့ဒီ Function ကရတဲ့ genres တွေကို map() လုပ်ပြီး <Button> တွေတန်းစီပြီး ပြပေးလိုက်တာပါ။ အပေါ်ဆုံးမှာ All Movies အတွက် <Button> တစ်ခုလည်း သပ်သပ်ထည့်ပေးထားပါတယ်။ စမ်းကြည့်ရင် ရလဒ်က အခုလိုဖြစ်မှာပါ။



TMDB က ပြန်ရတဲ့ Genres List အတိုင်းပြပေးတဲ့ Button စာရင်းတစ်ခု ရသွားပါပြီ။ ဒီနေရာမှာ ထူးခြားတာက `fetch()` ကိုသုံးပြီး Data Fetch လုပ်ပေမယ့် Server-side မှာ လုပ်တာဖြစ်လို့ Client-side Browser ရဲ့ DevTools (Network) ကြည့်ရင် ဒီလုပ်ဆောင်ချက်ကို မြင်တွေ့ရမှာ မဟုတ်ပါဘူး။ အဲ့ဒီလို မမြင်တွေ့ရတာကိုက Server-side မှာ အလုပ်လုပ်သွားတဲ့ သဘောပါပဲ။

ဆက်လက်ပြီး ရုပ်ရှင်ဇာတ်ကားစာရင်းကို ပြပေးဖို့အတွက် `components/Movies.js` မှာ အခုလိုရေးပေးပါ။

Next.js - app/components/Movies.js

```
import Link from "next/link";

export default function Movies({ movies }) {
  const poster = "http://image.tmdb.org/t/p/w342";
```

```
  return (
    <>
      <div className="flex flex-wrap flex-row gap-4">
        {movies.map(movie => {
          return (
            <div
              key={movie.id}
              className="w-[200px] text-center flex flex-col">
              {movie.poster_path ? (
                <Link href={`/movie/${movie.id}`}>
                  <img
                    src={poster + movie.poster_path}
                    className="w-full hover:scale-105 transition-all"
                  />
                </Link>
              ) : (
                <div className="h-[300px]"></div>
              )}
              <div>
                <h4 className="mt-2">{movie.title}</h4>
                <span className="text-sm text-gray-500">
                  {movie.release_date.split("-")[0]}
                </span>
              </div>
            </div>
          );
        })}
      </div>
    </>
  );
}
```

Data Fetch တွေဘာတွေ မလုပ်ထားပါဘူး။ movie Data ကို Prop အနေနဲ့ လက်ခံပြီး၊ Data နဲ့အတူ

ပါလာတဲ့ ပိုစတာပုံနဲ့ Name, Release Data တို့ကို ပြပေးလိုက်တာပါပဲ။ Release Date ကို အကုန်မပြဘဲ Year ပဲပြချင်လို့ - နဲ့ Split လုပ်ပြီး ရှေ့ဆုံး Index ကိုပဲ ဖြတ်ယူလိုက်တာပါ။ ပိုစတာပုံရဲ့ Path လမ်းကြောင်းကို TMDB ရဲ့ API Documentation မှာကြည့်လို့ရပါတယ်။ နမူနာမှာတော့ ကြိုထည့်ပေးထားပြီးသားပါ။

ပိုစတာမရှိရင် `<div>` အလွတ်ကိုပြပေးဖို့ ထည့်စစ်ပြီးရေးပေးထားတာ သတိပြုပါ။ ပြီးတဲ့အခါ ပိုစတာပုံပြတဲ့ `<img>` Element မှာ `hover:scale-105` ဆိုတဲ့ Tailwind Class ပါပါတယ်။ Mouse Pointer နဲ့ ထောက်လိုက်ရင် ပုံလေးကို နည်းနည်းပိုကြီးပြီး ပြပေးမှာပါ။ အဲ့ဒါကို `transaction-all` နဲ့ တွဲသုံးထားပါတယ်။ ဒါကြောင့် CSS Transaction ကိုသုံးပြီး မသိမသာ Animation လေးနဲ့ ပုံကိုကြီးပြပေးမှာပါ။

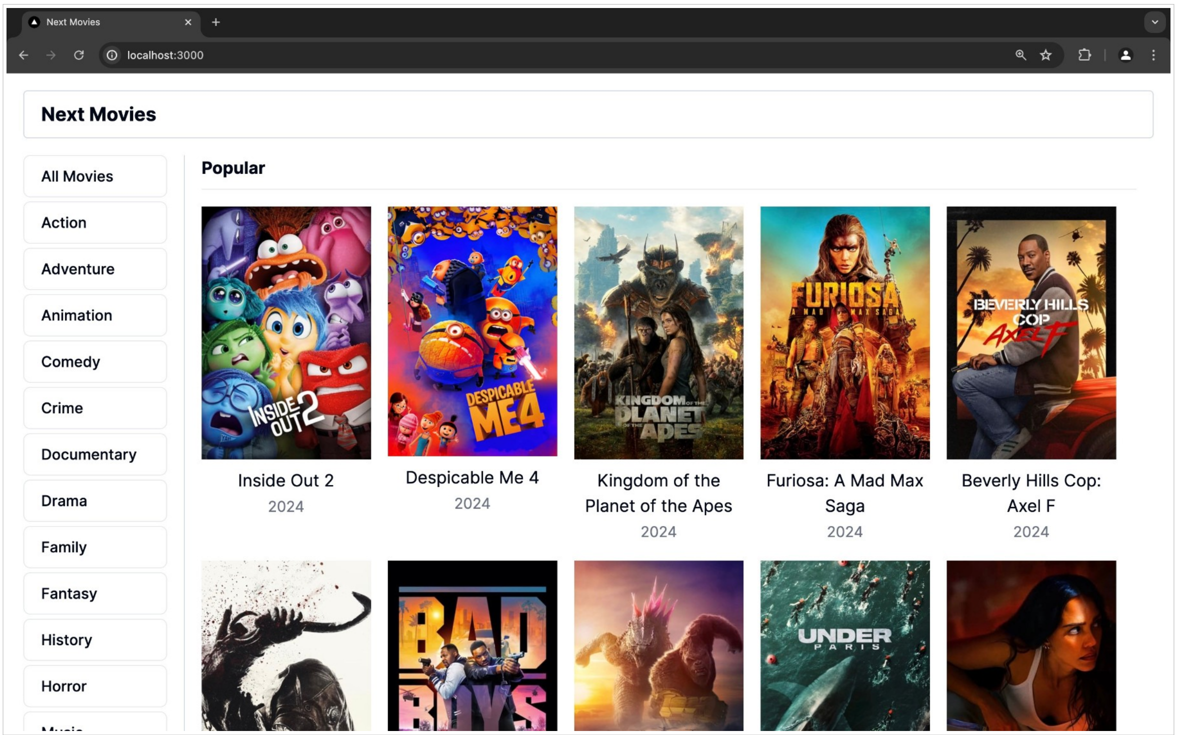
ဒီပရောဂျက်ရဲ့ Home Page လို့ဆိုရမယ့် ပင်မ `page.js` မှာ အခုလိုရေးပေးပါ။

Next.js - app/page.js

```
import Movies from "@/components/Movies";

const token = process.env.TMDB_TOKEN;

async function fetchPopular() {
  const res = await fetch("https://api.themoviedb.org/3/movie/popular", {
    headers: {
      Authorization: `Bearer ${token}`,
    },
  });

  return await res.json();
}

async function fetchTrending() {
  const res = await fetch("https://api.themoviedb.org/3/trending/movie/day", {
    headers: {
      Authorization: `Bearer ${token}`,
    },
```

```
  });

  return await res.json();
}

export default async function Home() {
  const popular = await fetchPopular();
  const trending = await fetchTrending();

  return (
    <div>
      <h3 className="font-bold border-b mb-4 pb-2">Popular</h3>
      <Movies movies={popular.results} />

      <h3 className="font-bold border-b my-4 pb-2">Trending</h3>
      <Movies movies={trending.results} />
    </div>
  );
}
```

စောစောက ရေးလိုက်တဲ့ Movies Component ကို Import လုပ်ထားပါတယ်။ ပြီးတဲ့အခါ Popular Movie နဲ့ Trending Movie တွေကို API ကနေ Fetch လုပ်ယူပါတယ်။ ရလာတဲ့ Data တွေကိုပေးပြီး Movies Component ကို အသုံးပြုလိုက်တာပါ။ စမ်းကြည့်ရင် ရလဒ်က အခုလို ဖြစ်ပါလိမ့်မယ်။



သူက ရုပ်ရှင်ဇာတ်ကား (၂၀) ကိုပဲပြန်ပေးထားပါတယ်။ လိုအပ်ရင် Paging လုပ်ဆောင်ချက်နဲ့ ကျန်ရုပ်ရှင်တွေကို ဆက်ယူလို့ရနိုင်ပါတယ်။ အဲ့ဒီလောက်ထိတော့ ဒီနေရာမှာ ထည့်မလုပ်တော့ပါဘူး။

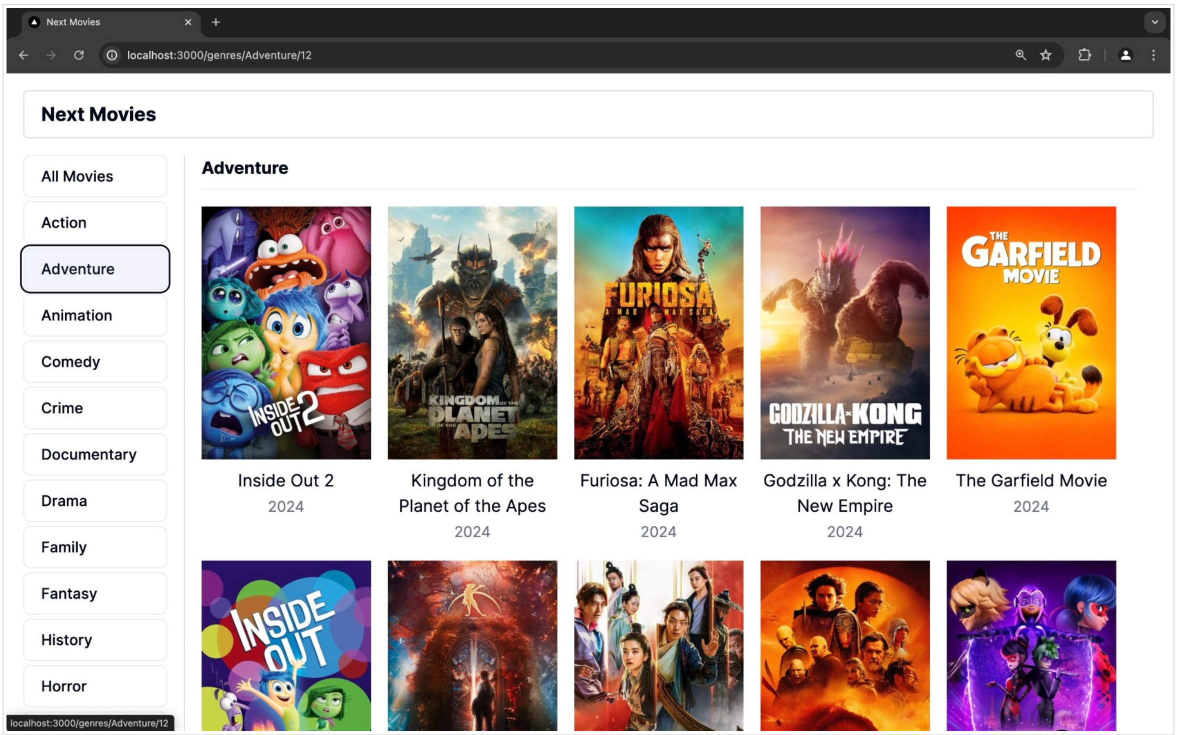
နောက်တစ်ဆင့်အနေနဲ့ Genres အလိုက် ရုပ်ရှင်တွေကို ခွဲကြည့်လို့ရအောင် `genres/[name]/[id]/page.js` မှာ အခုလိုရေးပေးပါ။

Next.js - app/genres/[name]/[id]/page.js

```
import Movies from "@/components/Movies";

const token = process.env.TMDB_TOKEN;

async function fetchMovies(id) {
  const res = await fetch(
    `https://api.themoviedb.org/3/discover/movie?with_genres=${id}`, {
      headers: {
        Authorization: `Bearer ${token}`,
```

```
      },
    }
  );

  return await res.json();
}

export default async function Home({ params }) {
  const byGenres = await fetchMovies(params.id);

  return (
    <>
      <h3 className="font-bold border-b mb-4 pb-2">{params.name}</h3>
      <Movies movies={byGenres.results} />
    </>
  );
}
```

ရေးပုံရေးနည်းခပ်ဆင်ဆင်ပါပဲ။ ဒီတစ်ခါ `with_genres` Parameter နဲ့သက်ဆိုင်ရာ Genres အောက်မှာရှိတဲ့ ရုပ်ရှင်တွေကိုပဲ ရွေးယူလိုက်တာပါ။ ပြတာကတော့ စောစောက Movies Component ကိုပဲ ပြန်သုံးလိုက်ပါတယ်။ Sidebar ထဲက Genres Button တွေကို နှိပ်ပြီး စမ်းကြည့်လို့ရပါပြီ။



ပြီးတဲ့အခါ ရုပ်ရှင်ဇာတ်ကား တစ်ကားချင်းစီကို ကြည့်လို့ရအောင်လုပ်ကြပါမယ်။ shadcn/ui ရဲ့ Badge ကို ထည့်သုံးမှာမို့လို့ အခုလို Install လုပ်ပေးပါ။

```
npx shadcn-ui@latest add badge
```

ဆက်လက်ပြီး `movie/[id]/page.js` မှာ အခုလို ရေးပေးပါ။

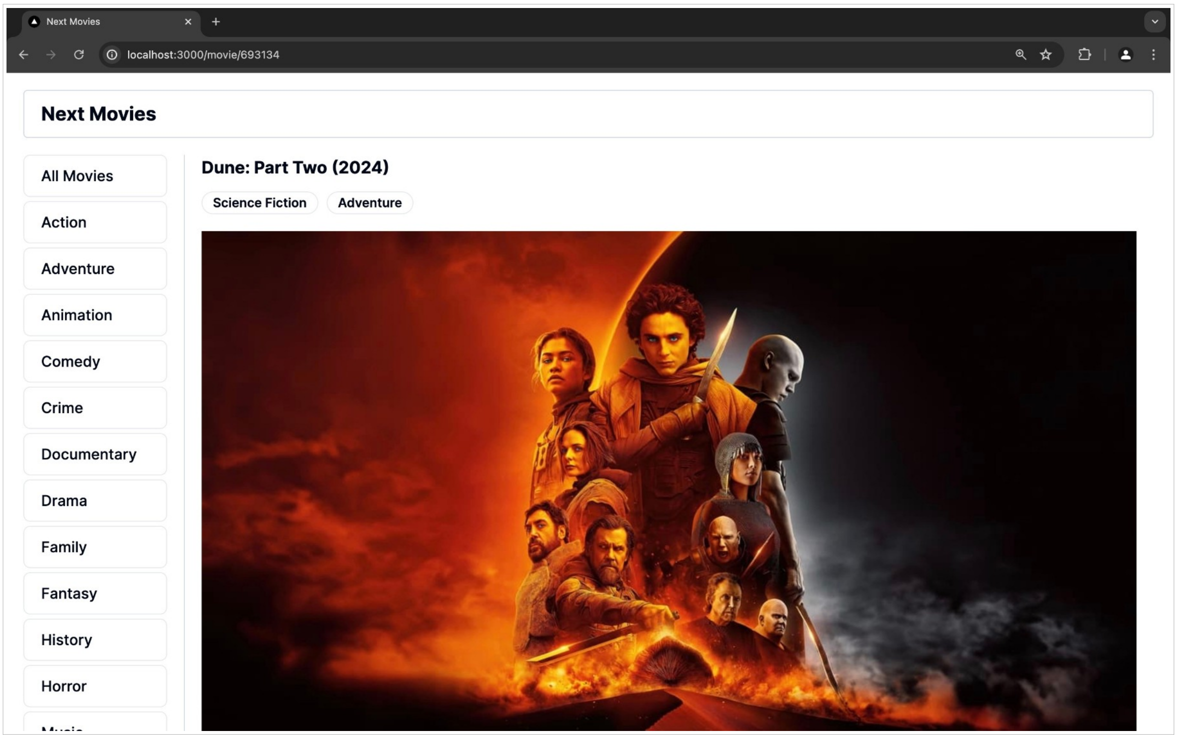
Next.js - app/movie/[id]/page.js

```
import { Badge } from "@/components/ui/badge";

const token = process.env.TMDB_TOKEN;

async function fetchMovie(id) {
  const res = await fetch(`https://api.themoviedb.org/3/movie/${id}`, {
    headers: {
```

```
      Authorization: `Bearer ${token}`,
    },
  });

  return await res.json();
}

export default async function Movie({ params }) {
  const movie = await fetchMovie(params.id);

  const cover = "http://image.tmdb.org/t/p/w1280";

  return (
    <>
      <h2 className="font-bold">
        {movie.title}
        <span className="ml-1">
          ({movie.release_date.split("-")[0]})
        </span>
      </h2>

      <div className="mb-4 mt-2">
        {movie.genres.map(genre => {
          return (
            <Badge
              key={genre.id}
              variant="outeline"
              className="mr-2">
              {genre.name}
            </Badge>
          );
        })}
      </div>

      <img src={cover + movie.backdrop_path} />
```

```
      <p className="mt-3">{movie.overview}</p>
    </>
  );
}
```

ဒီတစ်ခါ `id` ပေးပြီးတော့ ရုပ်ရှင်တစ်ခုထဲကို Fetch လုပ်လိုက်တာပါ။ Genre Name တွေကို Badge လေး တွေနဲ့ပြပေးထားပါတယ်။ ရုပ်ရှင်တစ်ခုဟာ Genres နှစ်ခုသုံးခုဖြစ်နိုင်ပါတယ်။ စမ်းကြည့်လို့ရပါပြီ။ Movie List ထဲက နှစ်သာရာတစ်ခုကို နှိပ်ကြည့်ရင် အခုလိုရလဒ်ကို ရမှာဖြစ်ပါတယ်။



ပိုစတာပုံလေးကို မပြတော့ဘဲ Cover ပုံကြီးကို အပြည့်ပြအောင်လုပ်ထားပါတယ်။ ပိုပြီး ပြည့်စုံသွားအောင် သရုပ်ဆောင်စာရင်းပါ ထည့်ပြလိုက်ချင်ပါတယ်။ ဒါကြောင့် `components/Persons.js` မှာ အခုလို ရေးပေးပါ။

Next.js - app/components/Persons.js

```
const token = process.env.TMDB_TOKEN;

async function fetchCasts(id) {
  const res = await fetch(
    `https://api.themoviedb.org/3/movie/${id}/credits`, {
      headers: {
        Authorization: `Bearer ${token}`,
      },
    }
  );

  return (await res.json()).cast;
}

export default async function Persons({ movie }) {
  const casts = await fetchCasts(movie.id);

  const profile = "http://image.tmdb.org/t/p/w185";

  return (
    <div className="flex gap-4 flex-row flex-wrap">
      {casts.map(cast => {
        return (
          <div
            key={cast.id}
            className="w-[180px] bg-gray-100 text-center
                       flex flex-col justify-between">

            {cast.profile_path ? (
              <img src={profile + cast.profile_path} />
            ) : (
              <div></div>
            )}
```

```
              <div className="p-2">
                <div className="text-sm">{cast.name}</div>
                <span className="text-sm text-gray-500">
                  {cast.character}
                </span>
              </div>
            </div>
          );
        })}
      </div>
  );
}
```

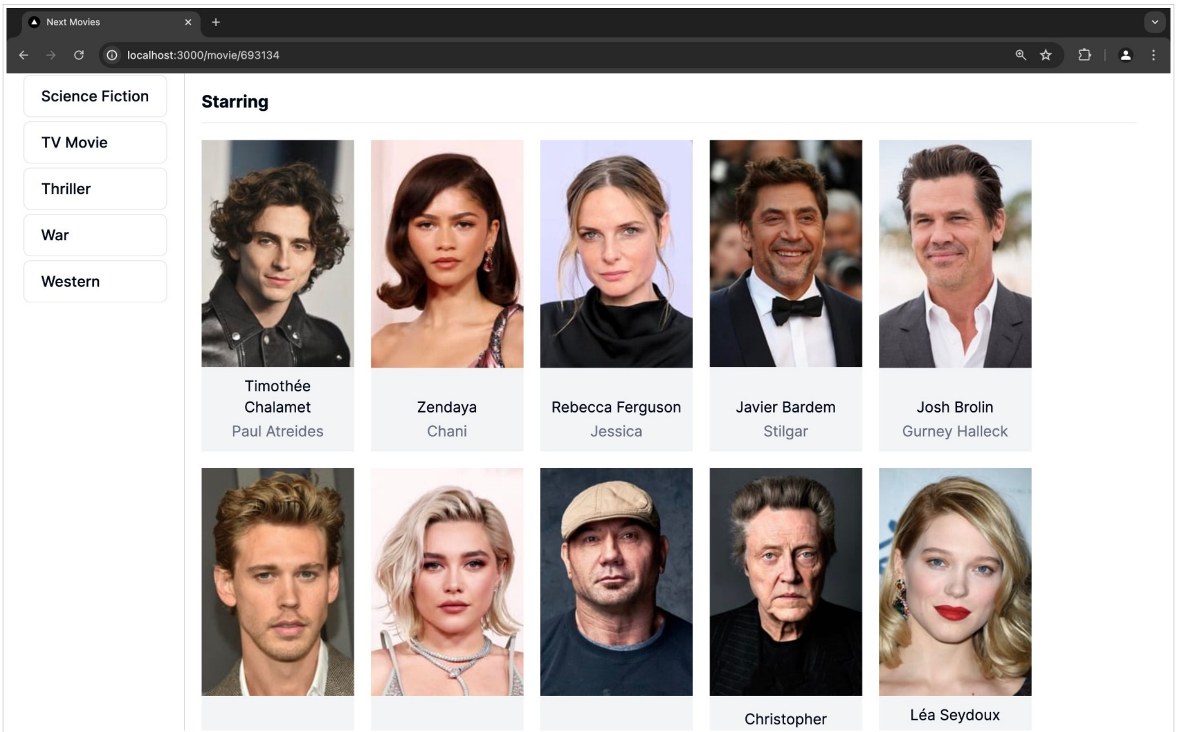
သူက `movie` ကို Prop အနေနဲ့ လက်ခံထားပါတယ်။ ပြီးတဲ့အခါ Movie ID ကိုသုံးပြီး Casts သရုပ်ဆောင်စာရင်းကို Data Fetch လုပ်ပါတယ်။ ရလာတဲ့ လူစာရင်းကို ပြထားတာပါ။ ပုံတွေနဲ့ တွဲပြဖို့အတွက် Profile ပုံကိုသုံးပါတယ်။ သူလည်း Movies တုန်းကလိုပဲ Profile ပုံမရှိရင် `<div>` အလွတ်တစ်ခုပြပေးဖို့ ထည့်စစ်ထားပါတယ်။

ဒီ Component ကို စောစောက Movie Page မှာ အခုလိုထည့်ပေးလိုက်ပါ။

`Next.js - app/movie/[id]/page.js`

```
<div className="mt-5">
  <h3 className="font-bold border-b mb-4 pb-2">Starring</h3>
  <Persons movie={movie} />
</div>
```

ရလဒ်က အခုလိုဖြစ်သွားပါလိမ့်မယ်။



အတော်စုံသွားပါပြီ။ သရုပ်ဆောင် တစ်ဦးချင်းစီကို အသေးစိတ် နှိပ်ကြည့်လို့ရအောင် လုပ်မယ်ဆို အလွယ်တကူ ထည့်လုပ်လို့ ရနိုင်ပါတယ်။ မလုပ်တော့ပါဘူး။ ကိုယ့်ဘာသာ TMDB API Documentation ကို တွဲကြည့်ပြီး စမ်းကြည့်လိုက်ပါ။

နောက်ဆုံးလုပ်ဆောင်ချက်အနေနဲ့ Search လုပ်ဆောင်ချက်လေး ထည့်ကြပါမယ်။ ဒါကြောင့် `search/page.js` မှာ အခုလို ရေးပေးပါ။

Next.js - app/search/page.js

```
import Movies from "@/components/Movies";

const token = process.env.TMDB_TOKEN;

async function fetchSearch(query) {
  const res = await fetch(
    `https://api.themoviedb.org/3/search/movie?query=${query}`,
    {
```

```
      headers: {
        Authorization: `Bearer ${token}`,
      },
    }
  );

  return await res.json();
}

export default async function Search({ searchParams }) {
  const search = await fetchSearch(searchParams.q);

  return (
    <>
      <h3 className="font-bold border-b mb-4 pb-2">
        Search: {searchParams.q}
      </h3>
      <Movies movies={search.results} />
    </>
  );
}
```

Search API ကို Fetch လုပ်လိုက်တာပါ။ ထူးခြားချက်အနေနဲ့ Component မှာ လက်ခံတာ params မဟုတ်တော့ပါဘူး။ `searchParams` ဖြစ်သွားပါတယ်။ `searchParams` ကို URL Query Parameter တွေ လက်ခံဖို့သုံးရပါတယ်။ ဥပမာ - URL က `search?q=dune` ဆိုရင် `searchParams` မှာ `q=dune` ဆိုတဲ့ Value ရှိနေမှာပါ။ Search လုပ်ဆောင်ချက်အတွက် Input Component လေး ထည့်သုံးချင်လို့ အခုလို Install လုပ်ပေးပါ။

```
npx shadcn-ui@latest add input
```

ပြီးတဲ့အခါ `components/Header.js` မှာ Search Form တစ်ခုနဲ့ Function တစ်ခု ဖြည့်စွက်ပေးလိုက်ပါ။

Next.js - app/components/Header.js

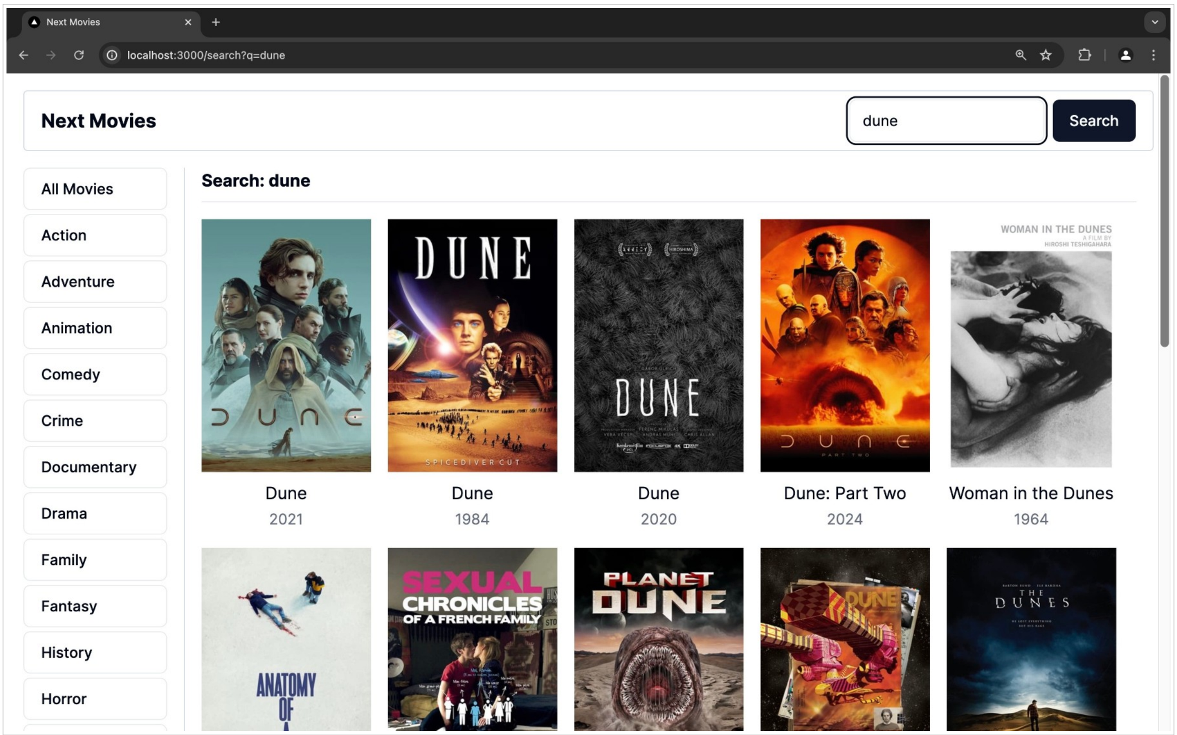
```
import Link from "next/link";
import { redirect } from "next/navigation";
import { Input } from "@/components/ui/input";
import { Button } from "@/components/ui/button";

export default function header() {
  async function search(formData) {
    "use server";

    const q = formData.get("q");
    redirect(`/search?q=${q}`);
  }

  return (
    <nav className="flex justify-between items-center border
                    rounded  border-slate-30 px-4 py-2 my-4">

      <h1 className="font-bold text-lg">
        <Link href="/">Next Movies</Link>
      </h1>

      <form action={search} className="flex gap-2">
        <Input
          type="text"
          name="q"
          placeholder="Search Movies"
        />
        <Button type="submit">Search</Button>
      </form>
    </nav>
  );
}
```

ဒီမှာထူးခြားချက်ရှိနေပါတယ်။ `<form>` တစ်ခုရဲ့ အခြေခံအလုပ်လုပ်ပုံက Server Submit လုပ်တာပါ။ `action` မှာ လက်ခံအလုပ်လုပ်မယ့် Server URL ပေးရပါတယ်။ နမူနာမှာ `action` အတွက် Server URL မပေးဘဲ `search` Function ကိုပေးထားပါတယ်။ ဒါကြောင့် Form ကို Submit လုပ်လိုက်ရင် Server URL ကို Submit လုပ်မသွားဘဲ ပေးထားတဲ့ Function အလုပ်လုပ်သွားပါလိမ့်မယ်။ ဒီသဘောကို Server Action လို့ခေါ်ပါတယ်။ ရေးထားတဲ့ကုဒ်မှာကြည့်ရင် သိပ်မရှုပ်လှပေမယ့် ဒီလိုအလုပ်လုပ်နိုင်အောင် နောက်ကွယ်မှာ ကျယ်ပြန့်တဲ့ သဘောသဘာဝတွေကို အသုံးပြုထားကြရတာပါ။

`<form>` ဟာ User Input တွေကို လက်ခံပြီး User ခလုပ်နှိပ်မှ အလုပ်လုပ်တဲ့ Client Component ဖြစ်ပါတယ်။ Form Action အနေနဲ့ ချိတ်ဆက်ပေးမယ့် Function ဟာ Server-side Function ဖြစ်ဖို့လိုပါတယ်။ ဒါကြောင့် `search` Function ထဲမှာ `"use server"` လို့ထည့်ရေးပေးထားရတာကို သတိပြုပါ။ အဲ့ဒီလို ထည့်မရေးရင် အလုပ်မလုပ်ပါဘူး။

Next.js က Server-side Function တစ်ခုကိုတွေ့ရင် အဲ့ဒီ Function ကို Server-side မှာ Run ပြီး Client ဘက်က ဆက်သွယ်လာမယ့်အချိန်ကို စောင့်ကြည့်နေပါတယ်။ Client က Submit လုပ်လိုက်တဲ့အချိန်မှာ စောင့်ကြည့်နေတဲ့ Server-side Function ကအလုပ်လုပ်ပေးသွားမှာပါ။ ဒါကြောင့် နောက်ကွယ်ကကိစ္စတွေက ကျယ်ပြန့်တယ်လို့ ပြောတာပါ။

နမူနာမှာ `redirect` ကို Import လုပ်ထားတာလည်း သတိပြုပါ။ Server Action Function က `formData` ကို လက်ခံအလုပ်လုပ်ပေးပါတယ်။ `formData` နဲ့အတူပါလာတဲ့ Input `q` ရဲ့ Value ကို `search` Page ရဲ့ URL Query အဖြစ်ထည့်ပေးပြီး Redirect လုပ်ထားပါတယ်။ ဒါကြောင့် Form ကို Submit လုပ်လိုက်ရင် Search Page ကို ရောက်သွားမှာ ဖြစ်ပါတယ်။ စမ်းကြည့်ရင် အခုလိုတွေ့ရပါလိမ့်မယ်။



ဒီလိုတွေရေးမယ့်အစား `<form action="/search" method="get">` လို့ရေးပေးလိုက်ရင်လည်း တူညီတဲ့ပုံစံကို အလုပ်လုပ်နိုင်ပေမယ့် Form Submit ကြောင့် Page Reload ဖြစ်သွားမှာပါ။ အခု Server Action ရေးနည်းကို သုံးထားတဲ့အတွက် Page Reload မဖြစ်တော့သလို နောင်လိုအပ်ရင် Server-side မှာ လုပ်စရာရှိတဲ့ တခြားအလုပ်တွေလည်း Server Action Function ထဲမှာ ထည့်လုပ်လို့ရသွားမှာပါ။

ဒီလောက်ဆိုရင် နမူနာလုပ်ကြည့်ချင်တဲ့ လုပ်ဆောင်ချက်တွေစုံသွားပါပြီ။ ထပ်ဖြည့်မယ်ဆိုရင်တော့ လုပ်လို့ရတာတွေ အများကြီး ကျန်ပါသေးတယ်။ ရုပ်ရှင်တွေသာမက တီဗွီစီရီးတွေလည်း ထည့်လို့ရပါတယ်။ တီဗွီစီးရီးဆိုရင် Episode List ပါ တွဲပြရပါမယ်။ သရုပ်ဆောင်တွေသာမက ရိုက်ကူးသူ ထုတ်လုပ်သူတွေပါ ထည့်ပြလို့ရပါတယ်။ Rating တွေ Review တွေ ထည့်ပြလို့ ရနိုင်ပါတယ်။ Review တွေရေးလို့ရအောင် လုပ်ပေးလို့ ရနိုင်ပါတယ်။ Trailer ဗွီဒီယိုတွေ တွဲပြလို့ရနိုင်ပါတယ်။ State တွေ Database တွေနဲ့ တွဲပြီး User ရဲ့ Favorite တွေ Watch List တွေထည့်လို့ရနိုင်ပါတယ်။ လက်တွေ့လေ့ကျင့်ရင်း ကိုယ့်ဘာသာ စမ်းပြီးဖြည့်စွက်ကြည့်ဖို့ တိုက်တွန်းပါတယ်။

နိဂုံးချုပ်

JavaScript နည်းပညာတွေကို Front-end, Back-end နဲ့ Web Development ပိုင်းမှာသာမက၊ Desktop, Mobile, Microservice စသည်ဖြင့် နေရာတိုင်းမှာ သူမပါရင်မပြီးသလောက် ကျယ်ကျယ်ပြန့်ပြန့် အသုံးပြုကြသလို၊ အသစ်အဆန်း အပြောင်းအလဲတွေလည်း များပြားမြန်ဆန်လှ ရှိပါတယ်။ ဒီသဘောကို လေ့လာသူများလည်း သိရှိကြပြီးသားပါ။

ပုံစံစုံမှုကွဲတွေ အများကြီးရှိပြီး အမြဲပြောင်းလဲနေလို့ အမီလိုက်သင့်တာကို မြတ်ခြေမပြတ် အမီလိုက်ဖို့လိုပေမယ့် ဘာရွေးရမှန်းမသိ မရေမရာတွေဖြစ်ကုန်လို့လည်း မဖြစ်ပါဘူး။ ဒီစာအုပ်မှာ ဖော်ပြခဲ့တဲ့ နည်းပညာတွေဟာ စာရေးသူအနေနဲ့ နည်းပညာအလားလာတွေ၊ လက်ရှိလိုအပ်ချက်တွေပေါ်မှာ ချင့်ချိန်ပြီး၊ လေ့လာသူများ ဦးစားပေးအာရုံစိုက် လေ့လာစေချင်တဲ့ အကြောင်းအရာတွေကို စုစည်းဖော်ပြလိုက်တာပါ။

ထည့်သွင်းဖော်ပြနိုင်ခြင်းမရှိတဲ့ ကျန်နည်းပညာတွေ၊ မူကွဲတွေကိုလည်း ပူးတွဲလေ့လာနိုင်ရင်တော့ ကောင်းပါတယ်။ လေ့လာစရာတွေများလွန်းလို့ အခုချက်ခြင်း ထည့်မလေ့လာနိုင်သေးရင်တောင် Follow Up လုပ်ပြီး စောင့်ကြည့်နေသင့်တဲ့ နည်းပညာတွေအကြောင်း ထည့်ပြောချင်ပါတယ်။

Language နေရာမှာ JavaScript အစား TypeScript ကို အစားထိုးအသုံးပြုနိုင်မယ်ဆိုရင် အမှားနည်းပြီး အရည်အသွေးပိုကောင်းတဲ့ ကုဒ်တွေကို ရရှိနိုင်ပါတယ်။ JavaScript ရပြီးသားသူအတွက် အခက်အခဲသိပ်မရှိဘဲ လေ့လာအသုံးပြုလို့ရနိုင်တဲ့အတွက် လေ့လာထားသင့်ပါတယ်။

TypeScript – https://www.typescriptlang.org/

JavaScript ကို လုံးဝမဟုတ်တဲ့ Language တွေပူးတွဲလေ့လာချင်တယ်ဆိုရင်တော့ Go, Rust နဲ့ Zig တို့ကို လေ့လာနိုင်ပါတယ်။ Performance ပိုကောင်းတဲ့ Lower-Level Language တွေပါ။ JavaScript Ecosystem မှာလည်း ဒီနည်းပညာတွေကို အသုံးပြုဖန်တီးထားတဲ့ Tools တွေတဖြည်းဖြည်း ပိုများလာနေပါတယ်။

Runtime အနေနဲ့ ဒီစာအုပ်အစပိုင်းမှာလည်း ထည့်သွင်းဖော်ပြခဲ့ပြီးဖြစ်ပါတယ်။ Node ကို အဓိကအသုံးပြုကြပေမယ့် ပိုပြီး Modern ဖြစ်တဲ့ Runtime အဖြစ် Deno နဲ့ Bun ကို ထည့်သွင်းလေ့လာ စောင့်ကြည့်သင့်ပါတယ်။ TypeScript တို့ JSX တို့လို လုပ်ဆောင်ချက်တွေကို Third-Party နည်းပညာတွေ မလိုဘဲ Runtime က တိုက်ရိုက် Run ပေးနိုင်နေပါပြီ။

Deno – https://deno.com

Bun – https://bun.sh

Package Manager အကြောင်းလည်း အစပိုင်းမှာ ထည့်သွင်းဖော်ပြခဲ့ပြီးဖြစ်ပါတယ်။ NPM ကို အဓိကထား အသုံးပြုခဲ့ပေမယ့် yarn နဲ့ pnpn ကိုထည့်သွင်းလေ့လာသင့်ပါတယ်။ Module Bundler နဲ့ Build Tool နည်းပညာဘက်မှာ Vite ကို အဓိကအသုံးပြုခဲ့ပါတယ်။ ဆက်စပ်နည်းပညာအဖြစ် Turbopack လို နည်းပညာမျိုးကိုလည်း ထည့်သွင်းလေ့လာသင့်ပါတယ်။ Next.js ကိုတီထွင်တဲ့ Vercel ကပဲ တီထွင်ထားတဲ့ နည်းပညာပါ။

Turbopack – https://turbo.build/pack

ဒီစာအုပ်မှာနမူနာအနေနဲ့ ဖော်ပြခဲ့တဲ့ ပရောဂျက်တွေမှာ Field တွေအများကြီးနဲ့ စီမံအလုပ်လုပ်ရတဲ့ Form တွေမပါခဲ့ပါဘူး။ ဒါကြောင့် Form နဲ့ပက်သက်တဲ့ သီးခြားနည်းပညာမသုံးဘဲ ကိုယ့်ဘာသာဘဲ State တွေနဲ့ စီမံခဲ့ကြပါတယ်။ React Hook Form တို့ Tenstack Form တို့လို နည်းပညာတွေကို ထည့်သွင်းလေ့လာသင့်ပါတယ်။

React Hook Form – https://www.react-hook-form.com

Tenstack Form – https://tanstack.com/form/latest

State Management နဲ့ပက်သက်ရင် အရင်က Redux လိုနည်းပညာမျိုးကို အဓိကနည်းပညာတစ်ခု အနေနဲ့ ထည့်သွင်းလေ့လာကြရပါမယ်။ အခုနောက်ပိုင်း Context တို့ React Query တို့လို နည်းပညာတွေက State နဲ့ပက်သက်တဲ့ အလုပ်အတော်များများကို လုပ်ပေးနိုင်သွားလို့ သီးခြားနည်းပညာ အသုံးနည်းလာကြပါတယ်။ ဒါကြောင့် ဒီစာအုပ်မှာ ထည့်မလေ့လာခဲ့ကြပါဘူး။ Redux လိုပုံစံမျိုးအလုပ်လုပ်ပြီး အခုနောက်ပိုင်း လူစိတ်ဝင်စားမှု ပိုများနေတာက Zustand ဖြစ်ပါတယ်။

Zustand – https://zustand-demo.pmnd.rs

Back-end Framework အဖြစ် Express ကို အဓိကထား အသုံးပြုခဲ့ပါတယ်။ တခြားရွေးချယ်စရာတွေ အများကြီးရှိနေပေမယ့် Express လိုပုံစံမျိုးကို ပိုပြီးတော့ ခေတ်မီသွားတာကတော့ Fastify ဖြစ်ပါတယ်။

Fastify – https://fastify.dev

ဒီစာအုပ်မှာ Prisma ကို ORM နည်းပညာအနေနဲ့ ထည့်သွင်းလေ့လာခဲ့ကြပါတယ်။ မူကွဲအနေနဲ့ Drizzle လို့ခေါ်တဲ့နည်းပညာလည်း ရှိပါတယ်။ နှစ်မျိုးစလုံးကို လူကြိုက်များကြပါတယ်။

Drizzle – https://orm.drizzle.team

ဒီစာအုပ်မှာ Testing နည်းပညာတွေကိုထည့်သွင်းမလေ့လာခဲ့ကြပါဘူး။ အရည်အသွေးပိုကောင်းတဲ့ ကုဒ်တွေရဖို့ဆိုရင် အနည်းဆုံး Unit Testing လောက်တော့ ရေးဖို့လိုပါတယ်။ Jest, Vitest နဲ့ React Testing Library တို့ကို ထည့်သွင်းလေ့လာထားသင့်ပါတယ်။ Jest နဲ့ Vitest က ပုံစံဆင်တူပါတယ်။ React မှ မဟုတ်ဘဲ မည်သည့် JavaScript ပရောဂျက်မှာမဆို အသုံးပြုနိုင်ပါတယ်။ React Testing Library ကတော့ React Component တွေကို Test လုပ်ပေးတဲ့ကုဒ်တွေ ရေးဖို့အသုံးဝင်ပါတယ်။ Jest တို့ Vitest တို့နဲ့ တွဲသုံးရပါတယ်။

Jest – https://jestjs.io

Vitest – https://vitest.dev

React Testing Library – https://testing-library.com

အထက်ကနည်းပညာတွေကို ကုဒ်ကို ကုဒ်နဲ့ပဲ ပြန်စမ်းတဲ့ နည်းပညာတွေပါ။ Component တွေကို မျက်မြင် အမျိုးမျိုး နှိုင်းယှဉ်စမ်းသပ်ဖို့အတွက် အသုံးဝင်တဲ့ Storybook ခေါ် နည်းပညာလည်း ရှိပါသေးတယ်။

Storybook – https://storybook.js.org

React အတွက် Animation Library အဖြစ် Framer Motion ကို ထည့်သွင်းလေ့လာသင့်ပြီး React Native အတွက်ဆိုရင် React Native Reanimated တို့ကို ထည့်သွင်းလေ့လာသင့်ပါတယ်။

Framer Motion – https://www.framer.com/motion/

React Native Reanimated – https://docs.swmansion.com/react-native-reanimated/

တခြားပူးတွဲစောင့်ကြည့်သင့်တဲ့ UI Framework တွေ Data Fetching နည်းပညာတွေ၊ Third-Party Authentication နည်းပညာတွေအကြောင်းကို သူ့နေရာနဲ့သူ ထည့်ပြောခဲ့ပြီးသားမို့လို့ ထပ်မပြောတော့ပါဘူး။ ဒါတွေအများကြီး ထည့်ပြောလို့လည်း စိတ်ရှုပ်မသွားပါနဲ့။ JavaScript နည်းပညာတွေက ဒီလိုပါပဲ။ ကိုယ့်အတွက် အရေးကြီးတာတွေကို ပိုအာရုံစိုက်ပြီး ကျန်ဆက်စပ်နည်းပညာတွေကိုတော့ ဒီလိုပဲ လုပ်ရင်းသင်၊ သင်ရင်းလုပ် အမြဲမပြတ် ဆက်လေ့လာသွားကြရမှာပဲဖြစ်ပါတယ်။

အားလုံးပဲ ထူးချွန်အောင်မြင်တဲ့ Developer တွေဖြစ်ကြပါစေ၊ အစစအရာရာ အဆင်ပြေကြပါစေလို့ ဆုတောင်းရင်း ဒီစာအုပ်ကို ဒီနေရာမှာပဲ နိဂုံးချုပ်လိုက်ပါတယ်။

အိမောင် (Fairway)

၁၅.၇.၂၀၂၄ ရက်နေ့တွင် ရေးသားပြီးစီးသည်။

စာရေးသူ၏ကိုယ်ရေးအကျဉ်း

အမည်ရင်း အိမောင် ဖြစ်ပြီး မကွေးတိုင်းဒေသကြီး သရက်မြို့ဇာတိဖြစ်သည်။ (၂၀၀၀) ပြည့်နှစ်တွင် သရက်မြို့ အ.ထ.က (၁) မှ တက္ကသိုလ်ဝင်တန်းစာမေးပွဲအောင်မြင်ပြီး (၂၀၀၁) ခုနှစ်မှစတင်ကာ ပညာဆက်လက် သင်ယူနိုင်ရန် ရန်ကုန်မြို့သို့ ပြောင်းရွှေ့ အခြေချခဲ့သည်။ (၂၀၀၃) ခုနှစ်တွင် ဒဂုံတက္ကသိုလ် လူ့စွမ်းအားအရင်းအမြစ်ဌာနမှ ပေးအပ်သည့် ကွန်ပျူတာနည်းပညာ ဒီပလိုမာဘွဲ့ (Diploma in Computer Studies) ကို ရရှိခဲ့ပြီး ထိုအချိန်မှစတင်၍ Software ရေးသားခြင်း လုပ်ငန်းများကို လုပ်ကိုင်လာခဲ့သည်။ (၂၀၀၆) ခုနှစ်မှ စတင်ကာ Web နည်းပညာအခြေပြု Software များကို စတင်ရေးသားခဲ့ပြီး ယနေ့အချိန်ထိ Web Developer တစ်ဦးအဖြစ် ဆက်လက်ရပ်တည် လုပ်ကိုင်နေသည်။

ကွန်ပျူတာနည်းပညာများကို အလေးထားလေ့လာနေ၍ ဆက်လက်တက်ရောက်ခြင်း မပြုဖြစ်ပဲ ရပ်နားထားခဲ့သည့် အဝေးသင်တက္ကသိုလ် (ဥပဒေပညာအထူးပြု) ကို (၂၀၀၉) ခုနှစ်တွင် ပြန်လည် တက်ရောက်ခဲ့ပြီး (၂၀၁၃) ခုနှစ်တွင် အောင်မြင်၍ ဥပဒေပညာဘွဲ့ (LL.B) ကိုရရှိခဲ့သည်။

(၂၀၀၉) ခုနှစ်တွင် Durosoft အမည်ဖြင့် Web Development လုပ်ငန်းကို ဦးဆောင်တည်ထောင်ခဲ့သည်။ ပြည်တွင်းပြည်ပမှ အပ်နှံကြသည့် Outsource Project များကို အဓိကထား ဆောင်ရွက်ခဲ့သည်။ ထို့နောက် (၂၀၁၁) ခုနှစ်တွင် Outsource Project များ ဆောင်ရွက်နေရာမှ မိမိတို့ကိုယ်ပိုင် Software Product များကိုသာ ရေးသား တော့မည် ဟူသည့် ရည်ရွယ်ချက်ဖြင့် Durosoft အား Fairway Web ဟူ၍ အမည်ပြောင်းကာ ဆက်လက် လုပ်ကိုင်ခဲ့သည်။ (၂၀၁၆) ခုနှစ်မှစတင်၍ Fairway Web အား Fairway Technology ဟု အမည်ပြောင်းကာ Software များ ရေးသားခြင်း၊ နည်းပညာဝန်ဆောင်မှုပေးခြင်းနှင့် သင်တန်းများ ပို့ချခြင်းတို့ကို ဆောင်ရွက်လျှက် ရှိပါသည်။ ဤစာရေးသားနေစဉ်တွင် Fairway Technology ၏ Managing Partner အဖြစ် တာဝန်ထမ်းဆောင်လျှက်ရှိပါသည်။

(၂၀၁၃) ခုနှစ်ဇွန်လတွင် **Professional Web Developer** စာအုပ် **ပထမအကြိမ်** ကို ရေးသားပြုစု ဖြန့်ချီခဲ့ပြီး စတုတ္ထအကြိမ်ထိ ဆက်တိုက်တည်းဖြတ် ပုံနှိပ်ထုတ်ဝေခဲ့သည်။ (၂၀၁၄) ခုနှစ်ဇွန်လတွင် **Ubuntu - သင့်အတွက် Linux** စာအုပ်ကို ရေးသားပြုစု ဖြန့်ချီခဲ့ပြီး၊ (၂၀၁၅) ခုနှစ် စက်တင်ဘာလတွင် **Rockstar Developer** စာအုပ်ကို ရေးသားပြုစု ဖြန့်ချီခဲ့ပါသည်။

(၂၀၂၀-၂၀၂၁) ခုနှစ်ကာလများတွင် **Bootstrap** လိုတိုရှင်း၊ **JavaScript** လိုတိုရှင်း၊ **PHP** လိုတိုရှင်း၊ **Laravel** လိုတိုရှင်း၊ **React** လိုတိုရှင်း၊ **API** လိုတိုရှင်း နှင့် **Bitcoin** လိုတိုရှင်း ဟူသော စာအုပ် (၇) အုပ်ကို ဆက်တိုက် ရေးသားဖြန့်ချီခဲ့သည်။ (၂၀၂၁) ခုနှစ် ဒီဇင်ဘာလတွင် **Professional Web Developer 2022** စာအုပ်ကိုရေးသား ဖြန့်ချီခဲ့ပြီး (၂၀၂၃) ခုနှစ် မတ်လတွင် **Professional Web Developer 2023** အမည်ဖြင့် ထပ်မံတည်းဖြတ် ဖြန့်ချီခဲ့ပါသည်။ ယခု **Rockstar Developer 2025** စာအုပ်ကို ဆက်လက်ဖြန့်ချီနိုင်ခဲ့ပြီ ဖြစ်သည့်အတွက် စုစုပေါင်း ရေးသားပြုစုထားသည့် နည်းပညာစာအုပ်ပေါင်း (၁၂) အုပ် ရှိသွားခဲ့ပြီ ဖြစ်ပါသည်။

(၂၀၁၀) ခုနှစ်မှ စတင်ကာ Web Development ဆိုင်ရာ သင်တန်းများကို ပို့ချလာခဲ့ပြီး၊ ယခုအခါ Fairway Technology သင်တန်းကျောင်း၏ ကျောင်းအုပ်ကြီးအဖြစ် နည်းပညာ သင်တန်းများကို ဦးဆောင်ပို့ချနေပါသည်။

ဇနီးဖြစ်သူ နှင်းဝေလွင်၊ သမီးဖြစ်သူ စုရတနာမောင် တို့နှင့်အတူ ရန်ကုန်မြို့တွင် အခြေချ နေထိုင်လျှက် ရှိသည်။ စာရေးသူအား ဆက်သွယ်လိုပါက eimg@fairwayweb.com အီးမေးလ်လိပ်စာ (သို့မဟုတ်) စာရေးသူ၏ Website ဖြစ်သော eimaung.com တွင် ဖော်ပြထားသည့် ဆက်သွယ်ရန် အချက်အလက်များ အတိုင်း ဆက်သွယ်နိုင်ပါသည်။