API

လို - တို - ရှင်း

အိမောင်

Fairway

© Copyright 2020, **Ei Maung**

Fairway Technology. All right reserved.

မာတိကာ

အခန်း (၁) – API ဆိုသည်မှာ

ကွန်ပျူတာပရိုဂရမ်တွေကို **Software** နဲ့ **Service** ဆိုပြီးတော့ အုပ်စု (၂) စု ခွဲကြည့်ကြရအောင်။

Software ဆိုတဲ့ထဲမှာ System Software, Desktop Solution, Web Application, Mobile App စသဖြင့် အမျိုးမျိုးရှိသလို၊ လုပ်ငန်းသုံး Software နဲ့ လူသုံး Software ဆိုပြီးတော့လည်း ကွဲပြားနိုင်ပါသေးတယ်။ ဘယ်လိုပဲ ကွဲပြားနေပါစေ Software လို့ပြောရင် အသုံးပြုသူ လူဖြစ်တဲ့ **User** က ထိတွေ့အသုံးပြုလို့ ရတဲ့ အရာတွေလို့ ဆိုနိုင်ပါတယ်။ ဒီစာအုပ်မှာ ပြောချင်တာက အဲ့ဒီ Software တွေအကြောင်း မဟုတ်ပါဘူး။ Service တွေအကြောင်း ပြောမှာပါ။

Service ဆိုတာကတော့ လူဖြစ်တဲ့ User က ထိတွေ့အသုံးပြုမှာ မဟုတ်ဘဲ၊ အခြားကွန်ပျူတာပရိုဂရမ်တွေက အသုံးပြုမယ့် အရာတွေပါ။ ပရိုဂရမ် A က ပရိုဂရမ် B ကို ဆက်သွယ်အသုံးပြုပြီး အလုပ်လုပ်နေပြီဆိုရင် ပရိုဂရမ် B ဟာ Service ဖြစ်သွားပါပြီ။ သူကို လူကသုံးတာ မဟုတ်ဘဲ အခြားပရိုဂရမ်က ဆက်သွယ်ပြီး သုံးနေတာမို့လို့ပါ။

ဒီတော့ Service တစ်ခုဖန်တီးဖို့ဆိုရင် အရေးပါလာတာက ဆက်သွယ်ရေးနည်းပညာပါ။ တစ်ခြား ပရိုဂရမ်တွေက ဆက်သွယ်ပြီး အသုံးပြုနိုင်ဖို့ဆိုရင် ဆက်သွယ်ရေးနည်းပညာတစ်ခုကို ကြားခံလိုအပ်ပါတယ်။ HTTP, FTP, POP/SMTP, XMPP စသဖြင့် ဆက်သွယ်ရေး နည်းပညာတွေ အမျိုးမျိုး ရှိပါတယ်။ သူတို့ရဲ့ အပေါ်မှာ XML–RPC, SOAP စသဖြင့် နောက်ထပ်ဆက်သွယ်ရေး နည်းပညာတွေ ရှိကြပါသေးတယ်။ လိုတို ရှင်း စာအုပ်ရဲ့ထုံးစံအတိုင်း အကျယ်တွေတော့ ချဲ့မနေတော့ပါဘူး။ လိုရင်းပဲပြောပါမယ်။ ဒီလိုနည်းပညာ အမျိုးမျိုးရှိနေတဲ့အထဲက ကနေ့ခေတ်မှာ Service တွေဖန်တီးဖို့ အကျယ်ပြန့်ဆုံး အသုံးပြုတဲ့ ဆက်သွယ်ရေးနည်းပညာ ကတော့ HTTP ဖြစ်ပါတယ်။ HTTP ဟာ Web Technology တစ်ခုဖြစ်လို့ HTTP အသုံးပြုထားတဲ့ Service တွေကို **Web Service** လို့လည်း ခေါ်ကြပါတယ်။

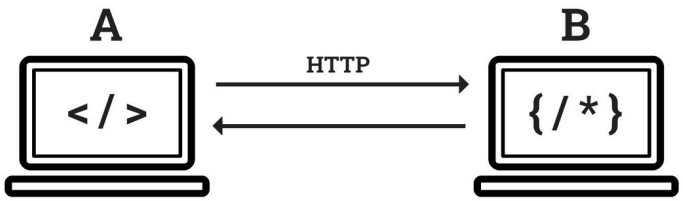
API ဆိုတာ မူရင်းအတိုင်းဆိုရင် Application Program Interface ရဲ့ အတိုကောက်ဖြစ်ပြီးတော့၊ Service များကို ရယူအသုံးပြုရန် သတ်မှတ်ထားသောနည်းလမ်း လို့ ပြောလို့ရပါတယ်။ သူများပေးထားတဲ့ Service ကို ကိုယ်က ရယူအသုံးပြုချင်ရင် သူသတ်မှတ်ထားတဲ့ API ကို ကိုယ်ကသိပြီး သတ်မှတ်ချက်အတိုင်း ရယူအသုံးပြုရပါမယ်။ ကိုယ်က Service တွေ ဖန်တီးပြီး ပေးချင်တာဆိုရင် ကိုယ့်ဘက်က အသုံးပြုနည်း API ကို သတ်မှတ်ပေးရပါတယ်။ ဒီစာအုပ်မှာ အဲ့ဒီလို Service တွေ တည်ဆောက်ပုံ၊ API သတ်မှတ်ပုံတွေကို လေ့လာသွားကြမှာပါ။ ထပ်ပြောပါမယ်။ Service က လက်တွေ့အလုပ်လုပ်တဲ့ ပရိုဂရမ်ဖြစ်ပြီးတော့၊ API က အဲ့ဒီ Service ကို ရယူအသုံးပြုလိုတဲ့အခါ အသုံးပြုရတဲ့နည်းလမ်း ဖြစ်ပါတယ်။

HTTP ကို အသုံးပြုထားတဲ့ Service တွေမှာ URL လိပ်စာတွေကို API အနေနဲ့ အသုံးပြုရတယ်လို့ အလွယ်မှတ်နိုင်ပါတယ်။ ဒါကြောင့် Service တစ်ခုကို ဆက်သွယ်အသုံးပြုလိုရင် သတ်မှတ်ထားတဲ့ URL ကို သိရပါတယ်။ Products တွေလိုချင်ရင် `/products` ဆိုတဲ့ URL သုံးရမယ်လို့ သတ်မှတ်ထားရင် အဲ့ဒီ `/products` ဆိုတဲ့ URL ကို API လို့သဘောထားပြီး အခြားပရိုဂရမ်တွေက အသုံးပြုရမှာပါ။

တစ်ကယ်တော့ မျက်စိထဲမှာ မြင်တွေ့ရတဲ့ URL ကို ဥပမာပြုပြီး ပြောလိုက်ပေမယ့် ပိုပြီးတော့ တိတိကျကျ ပြောရရင် **HTTP Request** တွေကို API အနေနဲ့ အသုံးပြုတယ်လို့ပြောမှ ပြည့်စုံ မှန်ကန်ပါလိမ့်မယ်။ HTTP Request တွေမှာ Request Method တွေ Request Headers တွေ URL တွေ ပေါင်းစပ်ပါဝင်ပါတယ်။ URL ဆိုတာ HTTP Request ရဲ့ အစိတ်အပိုင်းတစ်ခုသာ ဖြစ်ပါတယ်။ ဒီအကြောင်းတွေကို နောက်အခန်းမှာ ဆက်လက်လေ့လာကြပါမယ်။

အခန်း (၂) – HTTP Request

HTTP ဟာ Communication Protocol တစ်ခုဖြစ်ပါတယ်။ ပရိုဂရမ်နှစ်ခုအကြား အပြန်အလှန် ဆက်သွယ်ကြတဲ့အခါ နှစ်ဦးနှစ်ဘက် လိုက်နာအသုံးပြုရမယ့် နည်းလမ်းတွေကို သတ်မှတ်ပေးထားတာပါ။ ဆက်သွယ်မှု စတင်ပြုလုပ်သူကို Client လို့ ခေါ်ပြီး၊ လက်ခံတုံ့ပြန်သူကို Server လို့ ခေါ်ပါတယ်။ Client နဲ့ Server တို့ကို အသုံးပြု တည်ဆောက်ထားကြတဲ့ နည်းပညာတွေ တူစရာမလိုပါဘူး။ နှစ်ဦးလုံးက HTTP ကို အသုံးပြုမယ်ဆိုရင် အပြန်အလှန် ဆက်သွယ်အလုပ်လုပ်နိုင်ပါပြီ။ ဥပမာ – Client ကို JavaScript နဲ့ ဖန်တီးပြီး Server ကို PHP နဲ့ ဖန်တီးထားပေမယ့်၊ HTTP ရဲ့အကူအညီနဲ့ ဆက်သွယ်အလုပ်လုပ်နိုင်တဲ့ သဘောပါ။



HTTP ကိုအသုံးပြုကြတဲ့ထဲက ထင်ရှားမြင်သာတဲ့ Client ပရိုဂရမ်တွေကတော့ ကျွန်တော်တို့ နေစဉ်သုံးနေတဲ့ Chrome, Firefox, Edge စတဲ့ Web Browser တွေပါ။ ဒီပရိုဂရမ်တွေက Google, Facebook စတဲ့ Server တွေကို HTTP အသုံးပြုပြီး ဆက်သွယ်ပေးနိုင်ကြပါတယ်။ Client ဘက်က ဆက်သွယ်မှု စတင်ပြုလုပ်တဲ့ လုပ်ငန်းကို Request လုပ်တယ်လို့ ခေါ်ပြီး၊ Server ဘက်က ပြန်လည်တုံ့ပြန်မှု ပြုလုပ်တဲ့ လုပ်ငန်းကိုတော့ Response ပြန်တယ်လို့ ခေါ်ပါတယ်။

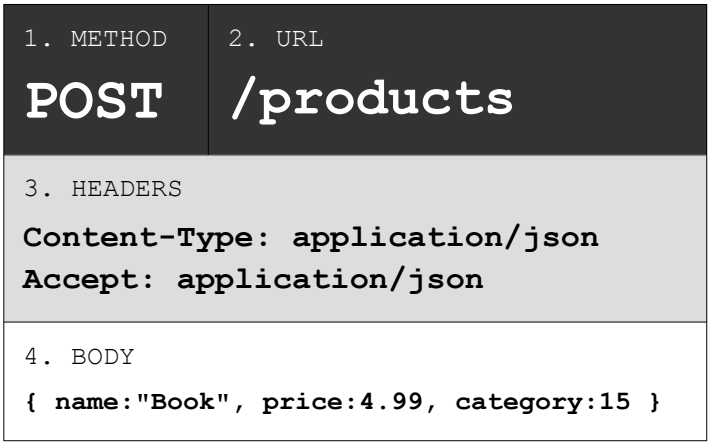
HTTP အကြောင်းကို Professional Web Developer စာအုပ်မှာလည်း ထည့်သွင်း ဖော်ပြထားလို့ လေ့လာကြည့်နိုင်ပါတယ်။ အဲ့ဒီစာအုပ်မှာ အထွေထွေ Web Development လုပ်ငန်းငန်းတွေ လုပ်ဖို့အတွက်

ရည်ရွယ်ရေးသား ဖော်ပြထားတာပါ။ ဒီစာအုပ်မှာတော့ Service နဲ့ API တွေ တည်ဆောက်နိုင်ဖို့အတွက် ရှုထောင့်နည်းနည်းပြောင်းပြီး ဖော်ပြသွားမှာဖြစ်ပါတယ်။

Requests တွေအကြောင်း လေ့လာတဲ့အခါ မှတ်ရလွယ်အောင် အပိုင်း (၄) ပိုင်း ခွဲပြောချင်ပါတယ်။

1. Request Method
2. Request URL
3. Request Headers
4. Request Body

Request တစ်ခုရဲ့ ဖွဲ့စည်းပုံကို အခုလို မျက်စိထဲမှာ မြင်ကြည့်ကြရအောင်။ အပေါ်မှာပြောထားတဲ့ အချက် (၄) ချက် စုဖွဲ့ပါဝင်တဲ့ Package လေးတစ်ခုလို့ မြင်ကြည့်ရမှာပါ။



Client က Server ကို ဆက်သွယ်မှု စတင်ပြုလုပ်တော့မယ်ဆိုရင် ဒီ Package လေးကို အထုပ်လိုက် လှမ်းပို့ပြီးတော့ ဆက်သွယ်ရတယ်လို့ ခေါင်းထဲမှာ ပုံဖော်ကြည့်ပါ။ METHOD ကတော့ ဆက်သွယ်မှုကို ပြုလုပ်ရတဲ့ ရည်ရွယ်ချက်ပါ။ Data လိုချင်လို့လား။ Data ပို့ချင်တာလား။ Data ပြင်စေချင်တာလား။ စသဖြင့်ပါ။ တစ်ကယ်တော့ Data လို့ မခေါ်ပါဘူး။ Resource လို့ခေါ်ပါတယ်။ အခေါ်အဝေါ်ကြောင့် ခေါင်းမစားပါနဲ့။ Resource ကတော့ အခေါ်အဝေါ်အမှန်ပါ။ ဒါပေမယ့် Data, Resource, Content, Document, Object စ သဖြင့် အမျိုးမျိုး ခေါ်ကြပါတယ်။ လိုရင်းကတော့ အတူတူပါပဲ။

URL ကတော့ ဆက်သွယ်ရယူလိုတဲ့အချက်အလက်ရဲ့ တည်နေရာပါ။ HEADERS ကတော့ ဆက်သွယ်မှု ဆိုင်ရာ အချက်အလက်များ ဖြစ်ပါတယ်။ ဘယ်လိုပုံစံနဲ့ လိုချင်တာလဲ။ ဘာကြောင့်လိုချင်တာလဲ။ စတဲ့ အချက်အလက်တွေပါ။ BODY ကတော့ Client ဘက်က ပေးပို့လိုတဲ့ အချက်အလက်တွေ ဖြစ်ပါတယ်။ ပေးပို့လိုတဲ့ Data မရှိတဲ့အခြေအနေမှာ Body မပါတဲ့ Request တွေဆိုတာလည်း ရှိနိုင်ပါတယ်။

အဲ့ဒီထဲကမှ အခုစတင် လေ့လာကြရမှာကတော့ Method တွေ အကြောင်းပဲ ဖြစ်ပါတယ်။

Request Methods

HTTP နည်းပညာက သတ်မှတ်ပေးထားတဲ့ Request Methods (၉) မျိုးရှိပါတယ်။ တစ်ခုချင်းစီမှာ သူ့အဓိပ္ပါယ်နဲ့သူပါ။ အရေးကြီးပါတယ်။ ဂရုစိုက်ပြီးလေ့လာပေးပါ။

1. **GET** – Client က Server ထံကနေ အချက်အလက်တွေရယူလိုတဲ့အခါ **GET** Method ကို အသုံးပြုရပါတယ်။

2. **POST** – Client က Server ထံ အချက်အလက်တွေပေးပို့လိုတဲ့အခါ **POST** Method ကို အသုံးပြုရပါတယ်။ ဒီလိုပေးပို့လိုက်တဲ့အတွက် Server မှာ အချက်အလက်သစ်တွေ ရောက်ရှိသွားတာ ဖြစ်နိုင်သလို၊ ရှိပြီးသား အချက်အလက်တွေ ပြောင်းလဲသွားတာ၊ ပျက်သွားတာမျိုးလည်း ဖြစ်နိုင်ပါတယ်။ ရိုးရိုး Web Application တွေမှာ **POST** ကို အချက်အလက်သစ် ပေးပို့ဖို့ရော၊ ပြင်ဆင်ဖို့ရော၊ ပယ်ဖျက်ဖို့ပါ အသုံးပြုကြပါတယ်။ ဒါပေမယ့် API မှာတော့ **POST** Method ကို အချက်အလက်သစ်တွေ ပေးပို့ဖို့အတွက်သာ အသုံးပြုကြလေ့ ရှိပါတယ်။ ပြင်ဆင်တဲ့လုပ်ငန်းနဲ့ ပယ်ဖျက်တဲ့လုပ်ငန်းအတွက် တစ်ခြားပိုသင့်တော်တဲ့ Method တွေကို သုံးကြပါတယ်။

3. **PUT** – Client က Server မှာရှိတဲ့ အချက်အလက်တွေကိုအစားထိုးစေလိုတဲ့အခါ **PUT** Method ကိုအသုံးပြုရပါတယ်။ ပေးလိုက်တဲ့အချက်အလက်နဲ့ မူလရှိနေတဲ့ အချက်အလက်ကို အစားထိုးလိုက်မှာပါ။ Server မှာရှိတဲ့ အချက်အလက်တွေ ပြောင်းလဲသွားမှာဖြစ်လို့ **PUT** Method ကို Update လုပ်ချင်တဲ့အခါ အသုံးပြုရတယ်လို့ ဆိုကြပါတယ်။

4. **PATCH** – Client က Server မှာရှိတဲ့ အချက်အလက်တစ်စိတ်တစ်ပိုင်းကိုပြောင်းစေလိုတဲ့အခါ **PATCH** Method ကို အသုံးပြုရပါတယ်။ ဒါကြောင့် **PATCH** ကိုလည်း အချက်အလက်တွေ Update လုပ်ချင်တဲ့အခါ အသုံးပြုနိုင်ပါတယ်။ **PUT** နဲ့ **PATCH** ရဲ့ ကွဲပြားမှုကို နားလည်ဖို့ လိုပါတယ်။ နှစ်ခုလုံးက Update လုပ်တာချင်း တူပေမယ့် သဘောသဘာဝ ကွဲပြားပါတယ်။ နဂိုရှိနေတဲ့ အချက်အလက်ကို၊ ပေးလိုက်တဲ့ အချက်အလက်သစ်နဲ့ အစားထိုးခြင်းအားဖြင့် ပြောင်းစေလိုရင် **PUT** ကိုသုံးပြီး၊ နဂိုရှိနေတဲ့ အချက်အလက်မှာ တစ်စိတ်တစ်ပိုင်းပဲ ရွေးထုတ်ပြင်ပေးစေလိုရင် **PATCH** ကို သုံးရတာပါ။

5. **DELETE** – Client က Server မှာရှိတဲ့ အချက်အလက်တွေ ပယ်ဖျက်စေလိုတဲ့အခါ **DELETE** Method ကို အသုံးပြုရပါတယ်။

6. **OPTIONS** – Client နဲ့ Server ကြား သဘောတူညီချက်ယူတဲ့ ဆက်သွယ်မှုတွေကို **OPTIONS** Method နဲ့ ပြုလုပ်ရပါတယ်။ သဘောတူညီချက်ယူတယ်ဆိုတာ ဥပမာ Client က ဆက်သွယ်ခွင့်ရှိရဲ့လား လှမ်းမေးတဲ့အခါ Server က Username, Password လိုတယ်လို့ ပြန်ပြောတာမျိုးပါ။ အချက်အလက် အမှန်တစ်ကယ်ပေးပို့တဲ့ Request မဟုတ်ဘဲ အကြို အမေးအဖြေ လုပ်တဲ့ Preflight Request တွေအတွက် သုံးတဲ့သဘောပါ။

7. **HEAD** – **GET** Method နဲ့ အတူတူပါပဲ။ ကွာသွားတာကတော့ Server က Response ပြန်ပေးတဲ့အခါ Headers ပဲပေးပါ၊ Body မလိုချင်ပါဘူးလို့ ပြောလိုက်တာပါ။ အပေါ်ကပုံမှာ ပြထားတဲ့ Request မှာ Headers နဲ့ Body ရှိသလိုပဲ၊ Server ကပြန်ပေးမယ့် Response မှာလည်း Headers နဲ့ Body ရှိပါတယ်။ **GET** က Headers ရော Body ပါ အကုန်လိုချင်တဲ့သဘောဖြစ်ပြီး **HEAD** ကတော့ Headers တွေပဲ လိုချင်တယ်ဆိုတဲ့သဘော ဖြစ်ပါတယ်။

8. **CONNECT** – ဒီ Method ကတော့ API မှာ မသုံးကြပါဘူး။ ရှိမှန်းသိအောင်သာ ထည့်ပြောထားထာပါ။ HTTP Proxy တွေအတွက် အဓိကသုံးပါတယ်။

9. **TRACE** – Client ပို့လိုက်တဲ့အတိုင်းပဲ Server က ပြန်ပို့ပေးစေလိုတဲ့အခါ **TRACE** Method ကို သုံးရပါတယ်။ အဆင်ပြေရဲ့လား ပေးပို့စမ်းသပ်ကြည့်တဲ့ သဘောမျိုးပါ။

ဒီ Request Method တွေကို အသုံးပြုလိုက်ခြင်းအားဖြင့် အလိုအလျောက် လိုချင်တဲ့လုပ်ဆောင်ချက်ကို ရသွားမှာ မဟုတ်ပါဘူး။ Service ကို ဒီဇိုင်းလုပ်မယ့်သူက ဒါတွေကို သိထားပြီး ပေးပို့လာတဲ့ Request Method နဲ့အညီ အလုပ်လုပ်ပေးနိုင်အောင် ဒီဇိုင်းလုပ်ရမှာပါ။ ဒီအတိုင်း အတိအကျ မလုပ်ရင်ရော မရဘူးလား။ ရပါတယ်။ ဥပမာ - အချက်အလက်သစ်တွေ တည်ဆောက်လိုရင် **`POST`** ကို သုံးရမယ်လို့ HTTP က ပြောထားပါတယ်။ ကိုယ်က အဲ့ဒီအတိုင်း လိုက်မလုပ်ဘဲနဲ့ **`GET`** နဲ့ပို့လာတာကိုပဲ အချက်အလက်သစ် တည်ဆောက်ဖို့ သတ်မှတ်မယ်ဆိုရင်လည်း ကိုယ့် Service က အလုပ်တော့ လုပ်နေမှာပါပဲ။ မှန်ကန်စနစ်ကျတဲ့ Service တော့ဖြစ်မှာ မဟုတ်ပါဘူး။ Service နဲ့ API ဒီဇိုင်း ကောင်းမကောင်းဆိုတာ ဒါတွေက စကားပြောသွားမှာပါ။

Request Headers

Request Headers တွေအကြောင်း ဆက်ကြည့်ကြပါမယ်။ HTTP က အသုံးပြုဖို့ သတ်မှတ်ပေးထားတဲ့ Headers တွေကတော့ အများကြီးပါ။ လိုက်ရေကြည့်တာ Request, Response အပါအဝင် Headers အားလုံးပေါင်း (၁၀၀) ကျော်ပါတယ်။ အဲ့ဒီထဲက၊ အတွေ့ရများတဲ့ Request Headers တွေက ဒါတွေပါ။

- **`Accept`**
- `Accept-Encoding`
- **`Authorization`**
- `Cache-Control`
- **`Content-Type`**
- `Cookie`
- `ETag`
- `If-Modified-Since`
- `Referer`
- `User-Agent`

Service တွေဖန်တီးတဲ့အခါမှာ ကိုယ်တိုင်တိုက်ရိုက် မကြာခဏ စီမံဖို့လိုနိုင်တာက `Accept`, `Content-Type` နဲ့ `Authorization` တို့ပါ။ ကျန်တာတွေက အရေးတော့ကြီးပါတယ်။ ဒါပေမယ့် ကိုယ်တိုင် စီမံဖို့လိုချင်မှလိုပါလိမ့်မယ်။ တစ်ချို့က Client ပရိုဂရမ်ကို Run တဲ့ Browser က လုပ်ပေးသွားပါလိမ့်မယ်။ တစ်ချို့ကိုတော့ အသင့်သုံး Library တွေရဲ့အကူအညီယူလိုက်လို့ ရနိုင်ပါတယ်။ ဖြစ်နိုင်မယ်ဆိုရင် ဒါတွေကို အသေးစိတ် အကုန်ပြောချင်ပေမယ့် သွားချင်တဲ့ လမ်းကြောင်းကနေ ဘေးနည်းနည်း ရောက်သွားမှာစိုးလို့ လိုမယ့်အပိုင်းလေးတွေပဲ အရင်ရွေး ကြည့်ကြရအောင်ပါ။

`Content-Type` Header ကနေစပြောပါမယ်။ `Content-Type` Header မှာ Client က ပေးပို့မယ့် Request Body ရဲ့ Content အမျိုးအစားကို သတ်မှတ်ပေးရမှာပါ။ Content Type တွေ ဒီလို အမျိုးမျိုးရှိနိုင်ပါတယ်။

- `image/jpeg`
- `image/png`
- `image/svg+xml`
- `text/plain`
- `text/html`
- `application/javascript`
- **`application/json`**
- `application/xml`
- **`application/x-www-form-urlencoded`**

ဥပမာတစ်ချို့ ရွေးထုတ်ပေးတာပါ။ အဲ့ဒီလို Content Type (MIME Type လို့လည်းခေါ်ပါတယ်) ပေါင်း (၆၀၀) ကျော်တောင် ရှိပါတယ်။ မနည်းမနောပါ။ အဲ့ဒီထဲကမှ API အတွက် အသုံးအများဆုံး ဖြစ်မှာကတော့ `application/json` နဲ့ `application/x-www-form-urlencoded` တို့ ဖြစ်ပါတယ်။

`Accept` Header ကတော့ ပြောင်းပြန်ပါ။ Client ကပြန်လိုချင်တဲ့ Content Type ကို သတ်မှတ်ဖို့အတွက် အသုံးပြုရတာပါ။ ဟိုးအပေါ်က Request တစ်ခုမှာပါတဲ့အရာတွေကို Package တစ်ခုလို့ မြင်ကြည့်ပါဆိုပြီး ပေးထားတဲ့နမူနာကို ပြန်လေ့လာကြည့်ပါ။ `Content-Type` က `application/json` ဖြစ်နေသလို `Accept` ကလည်း `application/json` ဖြစ်နေတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။ အဓိပ္ပါယ်ကတော့ Client ပေးပို့မှာ JSON ဖြစ်သလို Client က ပြန်လိုချင်တာလည်း JSON ပဲ လိုချင်တယ်လို့ ပြောလိုက်တာပါ။

လက်တွေ့အသုံးပြုရမယ်လို့ပြောတဲ့အထဲမှာပါတဲ့ `Authorization` Header အကြောင်းကိုတော့ သက်ဆိုင်ရာအခန်း ရောက်တော့မှပဲ ပြောပြပါတော့မယ်။ အခု Request Body အကြောင်း နည်းနည်းဆက် ကြည့်ချင်ပါတယ်။

Request Body

Client ပရိုဂရမ်နဲ့ Server ပရိုဂရမ်တို့ တည်ဆောက်ထားတဲ့ နည်းပညာတွေ မတူလည်းဘဲ HTTP ကို သုံးတာချင်းတူရင် အပြန်အလှန် ဆက်သွယ်အလုပ်လုပ်နိုင်တယ်လို့ ပြောခဲ့ပါတယ်။ ဒီနေရာမှာ ပြောစရာရှိလာတာက Data Format ပါ။ HTTP ကို သုံးကြတာချင်းတူလို့ အပြန်အလှန် ဆက်သွယ်နိုင်တယ် ဆိုပေမယ့် သုံးတဲ့ Data Format မတူရင် သူပေးတဲ့ Data ကို ကိုယ်နားမလည်၊ ကိုယ်ပေးတဲ့ Data ကို သူနားမလည် ဖြစ်နိုင်ပါတယ်။

HTTP ကတော့ ဘယ်လို Data Format (Content Type) ကို သုံးရမယ်လို့ ပုံသေကန့်သတ် မထားပါဘူး။ Support လုပ်တဲ့ Content Type ပေါင်း (၆၀၀) ကျော်ထဲက ကြိုက်တာကို သုံးပြီး ဆက်သွယ်လို့ရပါတယ်။ ဒါပေမယ့် ကြိုက်တာသုံးလို့ရတယ်ဆိုတိုင်း တစ်ယောက်တစ်မျိုး သုံးလို့မဖြစ်ပါဘူး။ ဘုံတူညီနဲ့ Format တစ်ခုကို ရွေးချယ် အသုံးပြုကြဖို့လိုပါတယ်။ အရင်ကတော့ အချက်အလက်ဖွဲ့စည်းပုံ တင်းကြပ်စနစ်ကျတဲ့ XML ကို ဘုံတူညီတဲ့ Data Exchange Format အနေနဲ့ အသုံးပြုကြပါတယ်။ ကနေ့ခေတ်မှာတော့ JSON ကိုသာ အဓိကထားပြီး အသုံးပြုကြပါတော့တယ်။ တစ်ခြား Content Type တစ်ခုခုကို နှစ်ဘက်ညှိပြီး သုံးကြမယ်ဆိုရင်လည်း ဖြစ်တော့ဖြစ်နိုင်ပါတယ်။ ဒါပေမယ့် JSON ဟာ ဖွဲ့စည်းပုံစနစ်ကျတယ်၊ ပေါ့ပါးတယ်၊ အသုံးပြုရလည်း လွယ်ကူတယ်၊ ပြီးတော့ ကွန်ပျူတာစနစ် အတော်များများကလည်း နားလည်တဲ့အတွက် JSON ကိုသာ ရွေးချယ်အသုံးပြုလာကြခြင်း ဖြစ်ပါတယ်။

Client က Data ပေးပို့တဲ့အခါ JSON Format ကို အသုံးပြုသင့်သလို့ Server က ပြန်ပို့တဲ့အခါမှာလည်း JSON Format ကိုသာ သုံးကြရမှာ ဖြစ်ပါတယ်။ တနည်းအားဖြင့် Request Body ဟာ JSON Format ဖြစ်ရမှာ ဖြစ်ပြီးတော့ Response Body ဟာလည်း JSON Format ဖြစ်ရမှာပဲ ဖြစ်ပါတယ်။

JSON အပြင်ဖြည့်စွက်မှတ်သားသင့်တဲ့ Format ကတော့ URL Encode Format ဖြစ်ပါတယ်။ သူ့ကို Request Body အဖြစ်ရံဖန်ရံခါ သုံးကြပါတယ်။ URL Encode Format ဟာ ရိုးရိုး Plain Text တစ်မျိုး ဖြစ်ပါတယ်။ ဒါပေမယ့် `Key1=Value1&Key2=Value2` ဆိုတဲ့ ဖွဲ့စည်းပုံနဲ့ သုံးပေးရပါတယ်။ ဥပမာ –

```
name=John%20Doe&age=22&email=john%40gmail.com
```

ပေးထားတဲ့နမူနာကို သေချာကြည့်လိုက်ရင် သူမှာ `name`, `age` နဲ့ `email` ဆိုတဲ့ တန်ဖိုးသုံးခု ပါဝင်ပါ

တယ်။ Key နဲ့ Value ကို = နဲ့ ပိုင်းခြားပြီး၊ တန်ဖိုးတစ်ခုနဲ့တစ်ခုကို & သင်္ကေတနဲ့ပိုင်းခြားလို့ ဒီ Operator နှစ်ခုကို အထူးပြုမှတ်သားရပါမယ်။ တန်ဖိုးတွေထဲမှာ Space နဲ့ Special Character တွေကို အသုံးပြုခွင့်မရှိပါဘူး။ အသုံးပြုလိုရင် သတ်မှတ်ထားတဲ့နည်းအတိုင်း Encode လုပ်ပေးရပါတယ်။ `%20` ဆိုတာ Space ဖြစ်ပါတယ်။ ဒါကြောင့် `John%20Doe` ကို Decode ပြန်လုပ်လိုက်ရင် John Doe ကိုရမှာ ဖြစ်ပါတယ်။ `%40` ကတော့ @ ဖြစ်ပါတယ်။ ဒါကြောင့် `john%40gmail.com` ကို Decode ပြန်လုပ်လိုက်ရင် john@gmail.com ကို ရမှာပါ။

တစ်ကယ်တော့ သတ်မှတ်ထားတဲ့ URL Encode\Decode တန်ဖိုးပြောင်းတဲ့အလုပ်ကို ကိုယ်တိုင်လုပ်စရာ မလိုပါဘူး။ အသင့်သုံးလို့ရတဲ့ နည်းပညာတွေရှိပါတယ်။ ဒါကြောင့် ဒီ Encode တန်ဖိုးတွေကြောင့် မျက်စိမ ရှုပ်ရအောင် နောက်ပိုင်းနမူနာတွေမှာဆိုရင် Encode လုပ်ပြီးတော့ ဖော်ပြမှာမဟုတ်တော့ဘဲ၊ ရိုးရိုးပဲ ဖော်ပြသွားတော့မှာပါ။ ဒီလိုပါ –

```
name=John Doe&age=22&email=john@gmail.com
```

ဒီလိုရိုးရိုးအမြင်အတိုင်း ဖော်ပြထားပေမယ့်၊ တစ်ကယ်တမ်း အလုပ်လုပ်တဲ့ အခါမှာတော့ Space အပါအဝင် Special Character တွေကို သတ်မှတ်ထားတဲ့ နည်းလမ်းအတိုင်း Encode လုပ်ပြီးတော့ အလုပ်လုပ်သွားပါလား ဆိုတာကို လေ့လာသူက သိထားဖို့ပဲ လိုပါတယ်။ ကိုယ်တိုင်လိုက်လုပ်ပေးစရာ တော့ လိုခဲပါတယ်။ တစ်ခါတစ်ရံ မဖြစ်မနေ ကိုယ်တိုင် Encode/Decode လုပ်ပေးဖို့ လိုတယ်ဆိုရင်လည်း သက်ဆိုင်ရာ Language မှာအသင့်ပါတဲ့ လုပ်ဆောင်ချက်တွေကို သုံးနိုင်ပါတယ်။ ဥပမာ – JavaScript မှာ `encodeURIComponent()` ကိုသုံးပြီးတော့ ရိုးရိုး String ကို URL Encode ပြောင်းနိုင်သလို၊ `decodeURIComponent()` ကိုသုံးပြီးတော့ String ပြန်ပြောင်းနိုင်ပါတယ်။

အခု စာအနေနဲ့ ဖတ်ရှုလေ့လာခဲ့တဲ့ သဘောသဘာဝတွေကို ရှုထောင့်တစ်မျိုးပြောင်း လေ့လာပြီးသား ဖြစ် သွားအောင် ကုဒ်နမူနာလေးတစ်ချို့နဲ့လည်း ဆက်လက်ဖော်ပြပါဦးမယ်။

Request Code Sample

ဒီကုဒ်နမူနာတွေက လက်တွေ့ လိုက်ရေးဖို့ မဟုတ်သေးပါဘူး။ စာနဲ့လေ့လာခဲ့တဲ့ သဘောသဘာဝကို ပိုပြီး မြင်သွားအောင် ကုဒ်နမူနာအနေတဲ့ ထပ်ပြပေးတဲ့သဘောပါ။ ကုဒ်တွေကို လိုက်ဖတ်ပြီး လေ့လာကြည့်ပါ။

```
$.ajax({
     url: "/products/",
     type: "POST",
     contentType:"application/x-www-form-urlencoded",
     data: "name=Book&price=8.99",
     success: function() {
          // do something
     }
});
```

ဒါဟာ jQuery ရဲ့ `ajax()` Function ကို အသုံးပြုပြီး Request ပေးပို့ပုံ ပေးပို့နည်း ဖြစ်ပါတယ်။ `url`, `type`, `contentType`, `data` စသဖြင့် Request တစ်ခုမှာ ပါဝင်ရမယ့်အရာတွေကို ပြည့်ပြည့်စုံစုံ သတ်မှတ်ပေးရတာကို တွေ့နိုင်ပါတယ်။ jQuery က HTTP အကြောင်း သေချာမသိတဲ့သူတွေလည်း အသုံးပြုရတာ လွယ်စေချင်လို့ ထင်ပါတယ်၊ အသုံးအနှုံးတွေကို ပြောင်းထားပါတယ်။ `type` ဆိုတာ METHOD ကို ပြောတာပါ။ `data` ဆိုတာ BODY ကို ပြောတာပါ။ ဒီကုဒ်ကိုပဲ နည်းနည်းပြင်လိုက်လို့ ရပါသေးတယ်။

```
$.ajax({
     url: "/products/",
     type: "POST",
     data: { name: "Book", price: 8.99 },
     success: function() {
          // do something
     }
});
```

`contentType` ဖြုတ်လိုက်တာပါ။ `contentType` မပါရင် Default က URL Encode ပဲမို့လို့ပါ။ ပြီးတော့ `data` ကိုကြည့်ပါ။ URL Encode ဆိုပေမယ့် JSON အနေနဲ့ ပေးလို့ရပါတယ်။ jQuery က JSON ကို URL Encode ဖြစ်အောင် သူ့ဘာသာပြောင်းပေးသွားမှာ မို့လို့ပါ။

နောက်ထပ်ကုဒ်နမူနာတစ်ခုကြည့်လိုက်ပါဦး။

```
fetch("/products", {
    method: "POST",
    headers: {
        "Content-type": "application/json",
        "Accept": "application/json"
    },
    body: JSON.stringify({ name: "Book", price: 8.99 })
});
```

ဒါကတော့ ES6 `fetch()` ကိုအသုံးပြုပြီး Request ပေးပို့လိုက်တာပါ။ သူကတော့ HTTP အသုံးအနှုံး အမှန်အတိုင်းပဲ အသုံးပြုပေးရပါတယ်။ URL ကို ပထမ Parameter အနေနဲ့ ပေးရပြီး ကျန်တဲ့ Method, Headers, Body စတာတွေက နောက်ကလိုက်ရတာပါ။ နမူနာမှာ HEADERS ရဲ့ `Content-type` ကို JSON လို့ သတ်မှတ်ထားတာကို တွေ့နိုင်ပါတယ်။ ဒီနေရာမှာ `Accept` Header ကိုလည်း သတိပြုပါ။ တစ်ချို့ Server တွေက Data Format နှစ်မျိုးသုံးမျိုး Support လုပ်ကြပါတယ်။ အဲ့ဒီလို Support လုပ်တဲ့ အခါ Client က `Accept` Header ကိုသုံးပြီး ကိုယ်လိုချင်တဲ့ Format ကို ပြောပြလို့ရတဲ့သဘောပါ။

BODY အတွက် JSON Data ကိုပေးထားပါတယ်။ ဒီလိုပေးတဲ့အခါ ရေးထုံးက JSON ရေးထုံးနဲ့ ရေးရပေမယ့် ပို့တဲ့အခါ String အနေနဲ့ ပို့ရလို့ `JSON.stringify()` Function ရဲ့အကူအညီနဲ့ String ပြောင်းထားတာကို တွေ့ရနိုင်ပါတယ်။ ဒါကိုလည်း သေချာ သတိပြုပါ။ Content Type ကို JSON လို့ပြောထားတဲ့အတွက် ပို့တဲ့ Format က JSON Format မှန်ရပါတယ်။ ဒါပေမယ့် HTTP က တစ်ကယ်ပို့တဲ့အခါ String အနေနဲ့ပဲ ပို့တာပါ။ ဒါကြောင့် JSON Format နဲ့ ရေးထားတဲ့ String ကို ပို့တာ ဖြစ်ရမှာပါ။

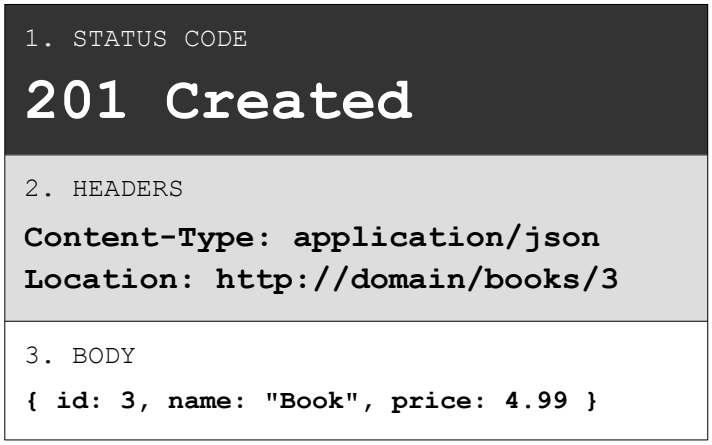
ဒီလောက်ဆိုရင် HTTP Request တွေအကြောင်း စုံသင့်သလောက် စုံသွားပါပြီ။ နောက်တစ်ခန်းမှာ Response တွေအကြောင်း ဆက်ကြည့်ကြရအောင်။

အခန်း (၃) – HTTP Response

HTTP Request တွေကို မှတ်ရလွယ်အောင် Method, URL, Header, Body ဆိုပြီး (၄) ပိုင်းမှတ်နိုင်တယ်လို့ ပြောခဲ့ပါတယ်။ HTTP Response တွေကိုတော့ (၃) ပိုင်း မှတ်ပေးပါ။

1. Status Code
2. Response Headers
3. Response Body

Request မှာတုံးက ပေးပို့တဲ့အခါ Package လေးတစ်ခုအနေနဲ့ Method, URL, Headers, Body တွေကို စုဖွဲ့ပြီးတော့ ပို့တယ်လို့ ပြောခဲ့ပါတယ်။ Server က ပြန်လည်ပေးပို့တဲ့ Response တွေကလည်း ဒီလိုပုံစံ Package လေးနဲ့ပဲ ထုပ်ပိုးပြီး ပြန်လာတယ်လို့ မြင်ကြည့်နိုင်ပါတယ်။



Request တုံးက Method တွေကို ဦးစားပေး ကြည့်ခဲ့ပါတယ်။ Response အတွက်တော့ Status Code တွေကို ဦးစားပေးပြီး လေ့လာကြရမှာပါ။

Status Codes

Status Code တွေကို အုပ်စု (၅) စုခွဲပြီး မှတ်နိုင်ပါတယ်။

- **1XX** – လက်ခံရရှိကြောင်း အသိပေးခြင်း
- **2XX** – ဆက်သွယ်မှု အောင်မြင်ခြင်း
- **3XX** – တည်နေရာ ပြောင်းလဲခြင်း
- **4XX** – Client ကြောင့်ဖြစ်သော Error
- **5XX** – Server ကြောင့်ဖြစ်သော Error

Status Code တွေဟာ အမြဲတမ်း (၃) လုံးတွဲပဲလာပါတယ်။ တစ်ခြားပုံစံမလာပါဘူး။ 1 နဲ့စတဲ့ Status Code တွေဟာ ဆက်သွယ်မှုကို လက်ခံရရှိကြောင်း အကြောင်းပြန်တဲ့ ကုဒ်တွေပါ။ အလုပ်မလုပ်သေးပါဘူး၊ အသိပေးတဲ့ အဆင့်ပဲ ရှိပါသေးတယ်။ 2 နဲ့စတဲ့ Status Code တွေကတော့ အများအားဖြင့်ဆက်သွယ်မှုအောင်မြင်သလို သက်ဆိုင်ရာလုပ်ငန်းလည်း အောင်မြင်တယ်လို့ အဓိပ္ပါယ်ရတဲ့ ကုဒ်တွေပါ။ 3 နဲ့စတဲ့ Status Code တွေကတော့ အများအားဖြင့် Client လိုချင်တဲ့အချက်အလက်ရဲ့ တည်နေရာပြောင်းသွားကြောင်း အသိပေးတဲ့ Code တွေပါ။ 4 နဲ့ စတဲ့ Status Code တွေကတော့ ဆက်သွယ်မှု မအောင်မြင်တဲ့အခါ Error အနေနဲ့ ပေးမှာပါ။ မအောင်မြင်ရခြင်းအကြောင်းရင်းက Client ရဲ့ အမှားကြောင့်ပါ။ 5 နဲ့စတဲ့ Status Code တွေကလည်း ဆက်သွယ်မှု မအောင်မြင်တဲ့အခါ Error အနေနဲ့ ပေးမှာပါပဲ။ ဒါပေမယ့် ဒီတစ်ခါတော့ မအောင်မြင်တာ Server ရဲ့ အမှားကြောင့်ပါ။

Status Code ပေါင်း (၅၀) ကျော်ရှိလို့ တစ်ခုမကျန်အကုန်မဖော်ပြတော့ပါဘူး။ အရေးကြီးတဲ့ Code တွေကိုပဲ ရွေးထုတ်ဖော်ပြသွားပါမယ်။ အရေးကြီးပါတယ်။ ကိုယ့် Service နဲ့ API ဘက်က သင့်တော်မှန်ကန်တဲ့ Status Code ကို ပြန်ပေးနိုင်ဖို့ လိုအပ်တဲ့အတွက် Code တွေရဲ့ အဓိပ္ပါယ်ကို သတိပြုမှတ်သားပေးပါ။

200 OK – ဆက်သွယ်မှုကိုလက်ခံရရှိပြီး လုပ်ငန်းအားလုံး အောင်မြင်တဲ့အခါ ဒီ Code ကိုပြန်ပို့ပေးရမှာပါ။ Request Method ဘာနဲ့ပဲလာလာ အောင်မြင်တယ်ဆိုရင် သုံးနိုင်ပါတယ်။

201 Created – ဆက်သွယ်မှုကိုလက်ခံရရှိပြီး အချက်အလက်သစ် တည်ဆောက်အောင်မြင်တဲ့အခါ ပြန်ပို့ပေးရမှာပါ။ အများအားဖြင့် **POST** သို့မဟုတ် **PUT** Method နဲ့လာတဲ့ Request တွေကို တုံ့ပြန်ဖို့ပါ။

202 Accepted – ဆက်သွယ်မှုကို အောင်မြင်စွာလက်ခံရရှိတယ်၊ လုပ်စရာရှိတာ ဆက်လုပ်ထားလိုက်မယ်လို့ ပြောတာပါ။ လက်ခံရရှိကြောင်းသက်သက်ပဲ အကြောင်းပြန်လိုတဲ့အခါ သုံးနိုင်ပါတယ်။

204 No Content – ဆက်သွယ်မှုကို လက်ခံရရှိတယ်၊ အောင်မြင်တယ်၊ ဒါပေမယ့် ပြန်ပို့စရာအချက်အလက် မရှိတဲ့အခြေအနေမျိုးမှာ သုံးနိုင်ပါတယ်။ ဥပမာ **DELETE** Method နဲ့လာတဲ့ Request မျိုးပါ။ ဖျက်လိုက်တယ်၊ အောင်မြင်တယ်၊ ဒါပေမယ့် ဘာမှပြန်မပို့တော့ဘူး ဆိုတဲ့သဘောပါ။

---

301 Move Permanently – အချက်အလက်ရဲ့ တည်နေရာပြောင်းသွားကြောင်း အသိပေးဖို့ သုံးပါတယ်။ Redirect ဆိုတဲ့သဘောပါ။ ဥပမာ - `/items` ကိုလိုချင်တယ်လို့ Client က Request လုပ်လာပေမယ့် `/items` ကမရှိဘူး၊ `/products` ပဲ ရှိတယ်ဆိုရင် 301 ကို Location Header နဲ့တွဲပြီး ပြန်ပို့နိုင်ပါတယ်။ ဥပမာ ဒီလိုပါ။

Request

```
GET /items
Content-Type: application/json
```

Response

```
301 Move Permanently
Location: http://domain/products
```

307 Temporary Redirect – နံပါတ်ကျော်ပြီး `307` ကို ပြောလိုက်တာ သဘောသဘာဝ တူလို့ပါ။ တည်နေရာ ပြောင်းသွားကြောင်း ပြောတာပါပဲ။ Redirect အတွက် သုံးပါတယ်။ ကွာသွားတာက `301` ဆို ရင် အပြီးပြောင်းတာ နောက်ပြန်မလာနဲ့တော့လို့ အဓိပ္ပါယ်ရပြီး `307` ဆိုရင်တော့ ခဏပြောင်းတာ၊ နောက်ပြန်လာချင် လာခဲ့ ဆိုတဲ့သဘောမျိုး အဓိပ္ပါယ်ရပါတယ်။

304 Not Modified – ဒါကအများအားဖြင့် Cache အတွက်အသုံးပါတယ်။ ရိုးရိုး Web Application တွေမှာ ဒီ Status Code ကို အမြဲလိုလိုတွေ့ရပေမယ့် API မှာတော့ အသုံးနည်းပါတယ်။ Client က Request လုပ်တဲ့အခါမှာ `If-Modified-Since` Header ကိုသုံးပြီးတော့ အပြောင်းအလဲရှိမှပေးပါလို့ ပြောလို့ရတယ်။ ဒါဆိုရင် Server က စစ်ကြည့်ပြီးတော့ အပြောင်းအလဲမရှိရင် Data ကို ပြန်မပေးတော့ဘဲ `304 Not Modified` လို့ ပြောလိုက်လို့ ရတဲ့သဘောပါ။

---

400 Bad Request – Client ရဲ့ Request က မပြည့်စုံရင် (သို့မဟုတ်) တစ်ခုခု မှားနေရင် ပြန်ပေးတဲ့ Error Code ပါ။

401 Unauthorized – Client က Request လုပ်လာပေမယ့်၊ အဲ့ဒီနေရာကိုဝင်ခွင့်မရှိတဲ့အခါ ဒီ Code ကို Error အနေနဲ့ပေးရပါတယ်။

Security မှာ Authentication နဲ့ Authorization ဆိုပြီး နှစ်မျိုးရှိပါတယ်။ လေ့လာစမှာ ဒီနှစ်ခုကို ခပ်ဆင် ဆင်ဖြစ်နေလို့ ရောကြပါတယ်။ မျက်စိလည်ကြပါတယ်။ အကျဉ်းချုပ်ပြောရရင် Authentication ဆိုတာ ဝင်ခွင့်ရှိမရှိစစ်တာပါ။ Login ဝင်လိုက်ရင် Authenticate ဖြစ်သွားတယ်၊ ဝင်ခွင့်ရသွားတယ် ဆိုပါတော့။ Authorization ဆိုတာကတော့ လုပ်ခွင့်ရှိမရှိစစ်တာပါ။ ဝင်ခွင့်ရှိရင်တောင် လုပ်ခွင့်က ရှိချင်မှ ရှိမှာပါ။ Authenticate ဖြစ်နေပေမယ့် Authorize ဖြစ်ချင်မှ ဖြစ်တာပါ။ ဒီလိုရောတတ်တဲ့ကိစ္စကိုမှ ထပ်ပြီးမျက်စိ လည်စရာ ဖြစ်သွားနိုင်တာကော့ `401 Unauthorized` မှာ အသုံးအနှုံး Unauthorized လို့ ပါနေပေ မယ့် တစ်ကယ်တမ်း Authenticate လိုတဲ့နေရာမှာ သုံးရတာပါတဲ့။ Authorize လိုတဲ့နေရာတွေအတွက် `403 Forbidden` ကို သုံးရပါတယ်။

403 Forbidden – Client က Request လုပ်လာပေမယ့် သူမှာ အဲ့ဒီအလုပ်ကို လုပ်ခွင့်မရှိတဲ့အခါ `403 Forbidden` ကို Error Code အနေနဲ့ ပြန်ပေးရပါတယ်။

404 Not Found – မရှိတဲ့အရာတစ်ခုကို Client က Request လုပ်ယူဖို့ကြိုးစားတဲ့အခါ ပေးရတဲ့ Error Code ဖြစ်ပါတယ်။ ဒီဟာကတော့ ကျွန်တော်တို့ အင်တာနက်ပေါ်မှာ မကြာခဏ တွေ့ဖူးနေကြပါ။ Request URL မှားနေတာ (သို့မဟုတ်) အရင်က မှန်ပေမယ့် အခုဖျက်ထားလိုက်လို့ မရှိတော့တာမျိုးမှာ ပေးရမှာပါ။

405 Method Not Allowed – တစ်ချို့ URL တွေကို အသုံးပြုခွင့်ရှိတဲ့ Method နဲ့ မရှိတဲ့ Method တွေ ခွဲပြီးသတ်မှတ်ထားမယ်ဆို သတ်မှတ်ထားလို့ ရပါတယ်။ ဥပမာ `/users` ဆိုတဲ့ URL အတွက် `GET` နဲ့ `POST` လက်ခံပေမယ့် `DELETE` လက်မခံဘူး ဆိုကြပါစို့။ Client က `/users` ကို `DELETE` Method နဲ့ Request လုပ်လာတဲ့အခါ `405` ကို ပြန်ပေးနိုင်ပါတယ်။

409 Conflict – Database Table တစ်ခုထဲမှာ Record အသစ်တစ်ကြောင်း ထည့်လိုက်တယ်ဆိုရင် Auto-Increment နဲ့ ID ပြန်ရလေ့ ရှိပါတယ်။ Auto-Increment မသုံးရင်လည်း တစ်ခြားနည်းလမ်း တစ်ခုခုနဲ့ Unique ဖြစ်တဲ့ Key/ID တစ်ခုခုကို ပြန်ရလေ့ ရှိပါတယ်။ ဒါကြောင့် အသစ်ထည့်လိုရင် ID တွေ Key တွေမပေးရပါဘူး။ ဒါကြောင့် အသစ်တည်ဆောက်ပါလို့ပြောတဲ့ Request တွေမှာ ID ပေးလာတဲ့အခါ လက်မခံသင့်ပါဘူး။ အဲ့ဒီလို ID ပေးပြီး အသစ်ဆောက်ခိုင်းနေရင် `409 Conflict` ကို Error Code အနေနဲ့ ပြန်ပေးလေ့ရှိပါတယ်။

415 Unsupported Media Type – Client က `Accept` Header နဲ့ သူလိုချင်တဲ့ Response Body Format ကို ပြောလို့ရတယ်လို့ ပြောခဲ့ပါတယ်။ အကယ်၍ Client က လိုချင်တဲ့ Format ကို Server က Support မလုပ်ရင် `415` ကို Error Code အနေနဲ့ ပြန်ပေးရပါတယ်။

418 I"m a Teapot – ဟာသသဘောနဲ့ ထည့်ထားတဲ့ Error Code ပါ။ ဟာသဆိုပေမယ့် တစ်ကယ်ရှိပါတယ်။ Client က Server ကို ကော်ဖီတစ်ခွက်ပေးပါလို့ Request လုပ်လာခဲ့ရင် `418` ကို Error Code အနေနဲ့ ပြန်ပေးရတာပါ။ Teapot မို့လို့လက်ဘက်ရည်ပဲရမယ်၊ ကော်ဖီမရဘူးလို့ ပြောလိုက်တဲ့ သဘောပါ။

429 Too Many Requests – Server တွေမှာ လက်ခံနိုင်တဲ့ Request အရေအတွက် အကန့်အသတ် ရှိကြပါတယ်။ ဥပမာ – တစ်မိနစ်မှာ Client တစ်ခုဆီက Request အကြိမ် (၆၀) ပဲ လက်ခံမယ်ဆိုတာမျိုး ပါ။ သတ်မှတ်ထားတဲ့ Request အကြိမ်ရေကျော်သွားပြီဆိုရင် 429 ကို Error အနေနဲ့ ပြန်ပေးနိုင်ပါတယ်။

---

500 Internal Server Error – Client Request က အဆင်ပြေပေမယ့် Server ဘက်မှာ Error ဖြစ်နေတဲ့အခါ ပြန်ပေးရတဲ့ Code ပါ။ ဘာ Error မှန်းမသိတဲ့အခါ (သို့မဟုတ်) ဘာ Error လည်း မပြောပြ ချင်တဲ့အခါ 500 ကို ပြန်ပေးကြပါတယ်။

502 Bad Gateway – Server တွေဟာ အချင်းချင်း ဆက်သွယ်ပြီးတော့လည်း အလုပ်လုပ်ဖို့ လိုတတ် ပါတယ်။ Client Request ကို လက်ရှိ Server က လက်ခံရရှိပေမယ့် လိုအပ်လို့ နောက် Server ကို ထပ်ဆင့် ဆက်သွယ်တဲ့အခါ အဆင်မပြေဘူးဆိုရင် 502 ကို ပြန်ပေးလေ့ရှိကြပါတယ်။

503 Service Unavailable – Server တွေမှာ တစ်ချိန်တည်း တစ်ပြိုင်းတည်း Concurrent လက်ခံအလုပ်လုပ်နိုင်တဲ့ ပမာဏအကန့်အသတ်ရှိပါတယ်။ ဥပမာ – Client အခု (၂၀) ကိုပဲ တစ်ပြိုင်တည်း လက်ခံ အလုပ်လုပ်နိုင်တယ် ဆိုတာမျိုးပါ။ တစ်ပြိုင်တည်း ဆက်သွယ်တဲ့ Client က သတ်မှတ် အရေအတွက် ကျော်သွားတဲ့အခါ 503 ကို ပြန်ပေးကြပါတယ်။

မှတ်စရာနည်းနည်းများတယ်လို့ ဆိုနိုင်ပေမယ့်၊ အခက်ကြီးတော့လည်း မဟုတ်ပါဘူး။ ကြိုးစားမှတ်သား ထားပေးပါ။ API ဒီဇိုင်းမှာ Request Method တွေနဲ့ Status Code တွေဟာ အရေးအကြီးဆုံးအစိတ်အပိုင်း တွေလို့ ဆိုနိုင်ပါတယ်။

ဆက်လက်ပြီး Response Header တွေအကြောင်းလေ့လာကြပါမယ်။

Response Headers

Response Headers တွေထဲမှာ `Content-Type` နဲ့ `Location` တို့ကိုပဲ အရင်ကြည့်ရမှာပါ။ Request Headers အကြောင်း ပြောခဲ့တုံးကနဲ့ သဘောသဘာဝ အတူတူပါပဲ။ `Content-Type` Header ကိုအသုံးပြုပြီးတော့ Response Body ရဲ့ Content Type ကို သတ်မှတ်ပေးရမှာပါ။ Response Body က လည်း၊ Request Body လိုပဲ JSON ပဲ ဖြစ်ရမှာမို့လို့ `application/json` ကိုပဲ အသုံးပြုရမှာပါ။ ဒါပေမယ့် Request Body မှာလို URL Encoded ကိုတော့ Response မှာ မသုံးကြပါဘူး။ ဒါကြောင့် Response Body ရဲ့ Content Type အဖြစ် JSON တစ်မျိုးတည်းပဲ သုံးရမှာလို့ ဆိုနိုင်ပါတယ်။

`Location` Header ကိုတော့ အထူးသဖြင့် POST နဲ့လာတဲ့ Request တွေအတွက် သုံးပါတယ်။ ဒီလိုပါ –

Request

```
POST /products
Content-Type: application/json
{ name: "Book", price: 4.99 }
```

Response

```
201 Created
Content-Type: application/json
Location: http://domain/products/3
{ id: 3, name: "Book", price: 4.99 }
```

Request က POST နဲ့လာတဲ့အတွက် Request Body ကိုသုံးပြီး အချက်အလက်သစ်တစ်ခု ဖန်တီးပေးလိုက်ပါတယ်။ ဒီလိုဖန်တီးလိုက်တဲ့အတွက် ထွက်လာတဲ့ ID တန်ဖိုးက နမူနာအရ 3 ပါ။ ဒါကြောင့် Response မှာ Status Code အနေနဲ့ `201 Created` ကိုသုံးပြီးတော့ Headers မှာ `Location` ကိုသုံးပြီး ဖန်တီးလိုက်တဲ့ Record ကို ပြန်ကြည့်ချင်ရင် ကြည့်လို့ရတဲ့ URL ကို တွဲပေးလိုက်တာပါ။

တစ်ခြား Headers တွေလည်း သူနေရာနဲ့သူအရေးကြီးပေမယ့် Request တုံးကလိုပဲ လက်တွေ့အသုံးပြုရမယ့် Headers တွေကိုပဲ ရွေးကြည့်ချင်တဲ့ သဘောပါ။

- `Access-Control-Allow-Credentials`
- `Access-Control-Allow-Origin`
- `Access-Control-Allow-Methods`
- `Access-Control-Allow-Headers`
- `Cache-Control`
- `Content-Encoding`
- `Content-Length`
- **`Content-Type`**
- `Expires`
- `Server`
- `Last-Modified`
- **`Location`**
- `Set-Cookie`
- `WWW-Authenticate`
- `X-Rate-Limit-Limit *`
- `X-Rate-Limit-Remaining *`
- `X-Rate-Limit-Reset *`

ဒီထဲက `Access-Control-*` နဲ့ စတဲ့ Headers တွေအကြောင်းကိုတော့ CORS လို့ခေါ်တဲ့ နည်းပညာ အကြောင်း ရောက်တော့မှ ထည့်ပြောပါမယ်။ `X-Rate-*` နဲ့ စတဲ့ Headers တွေကတော့ Standard Headers တွေ မဟုတ်ကြပါဘူး၊ ဒါပေမယ့် API ဒီဇိုင်းအတွက် အသုံးများကြတဲ့ Custom Headers တွေပါ။ နောက်တစ်ခန်းကျော်မှာ ဒီအကြောင်း ထည့်ပြောသွားမှာပါ။

Response Body

Response Body ဟာ JSON Format ဖြစ်ရမှာပါ။ ဒီနေရာမှာ Data ကို ပုံစံနှစ်မျိုးနဲ့ ပြန်ပေးလို့ရပါတယ်။ အရှိအတိုင်းပြန်ပေးလို့ရသလို Data Envelope နဲ့ထည့်ပြီးတော့လည်း ပြန်ပေးလို့ရပါတယ်။ ဥပမာ - စာအုပ်စာရင်းတစ်ခု JSON Array အနေနဲ့ ရှိတယ်ဆိုပါစို့။ ဒါကို အရှိအတိုင်း ပြန်ပေးမယ်ဆိုရင် သူ့ရဲ့ ဖွဲ့စည်းပုံက ဒီလိုဖြစ်နိုင်ပါတယ်။

```
[
    { id: 1, title: "React", price: 4.99 },
    { id: 2, title: "Laravel", price: 4.99 },
    { id: 3, title: "API", price. 4.99 }
]
```

အရှိအတိုင်း၊ ဒီအတိုင်းမပေးပဲ Data Envelope နဲ့ အုပ်ပြီးတော့ အခုလိုလည်း ပြန်ပေးလို့ရပါတယ်။

```
{
     data: [
          { id: 1, title: "React", price: 4.99 },
          { id: 2, title: "Laravel", price: 4.99 },
          { id: 3, title: "API", price. 4.99 }
     ]
}
```

သတိထားကြည့်ပါ ကွာပါတယ်။ Data Envelope နဲ့ ထည့်ပေးတဲ့အတွက် ဘာထူးသွားလည်းဆိုတော့ တစ်ခြား ဆက်စပ်အချက်အလက် (Meta information) တွေကိုပါ တွဲထည့်ပေးလို့ ရနိုင်သွားပါတယ်။ ဒီလိုပါ –

```
{
     success: true,
     total: 3,
     data: [
          { id: 1, title: "React", price: 4.99 },
          { id: 2, title: "Laravel", price: 4.99 },
          { id: 3, title: "API", price. 4.99 }
     ]
}
```

Data နဲ့အတူ တစ်ခြားဆက်စပ်အသုံးဝင်နိုင်တဲ့ အချက်အလက်တစ်ချို့ပါ ပါဝင်သွားတာပါ။ အကယ်၍ Error တွေဘာတွေ ရှိခဲ့ရင်လည်း ဒီလိုပုံစံ ဖြစ်သွားနိုင်ပါတယ်။

```
{
     success: false,
     errors: {
          code: 123,
          message: "Cannot get book list"
     }
}
```

ဒီတော့ Response Body မှာ Data ကို ဒီအတိုင်းပြန်ပေးမယ့်အစား Data Envelope နဲ့ပြန်ပေးတာက ပိုကောင်းတယ်လို့ ဆိုကြသူတွေ ရှိပါတယ်။ ဒါပေမယ့် တစ်ချို့ကလည်း ဒါဟာ Standard နဲ့မညီဘူးလို့ ဆိုကြပါတယ်။ Meta Information တွေကို တွဲပြီးတော့ ဖော်ပြချင်ရင်၊ Data Envelope နဲ့စုထည့်မှ မဟုတ်ပါဘူး၊ Headers မှာ Custom Header အနေနဲ့ ထည့်ပေးလိုက်လို့လည်း ရတာပဲလို့ ဆိုကြပါတယ်။ ဥပမာ – နမူနာပေးထားတဲ့အထဲကဆိုရင် `success` တို့ `errors.code` တို့ဆိုတာ လိုတောင်မလိုပါဘူး။ Status Code တွေ ရှိနေတာပဲ။ Status Code ကိုကြည့်လိုက်ယုံနဲ့ သိနေရပြီလေ။ ကျန်တဲ့ `errors.message` လိုဟာမျိုး သီးခြား ပေးဖို့လိုအပ်ရင် `X-Error-Message` ဆိုပြီးတော့ Custom Header အနေနဲ့ ထည့်ပေးလိုက်လို့ ရနိုင်ပါတယ်။ Data ထဲမှာ သွားရောပေးစရာမလိုပါဘူးလို့ ပြောကြပါတယ်။ ဒါကတော့ API Developer တွေကြားထဲမှာ ရှိနေကြတဲ့ မတူကွဲပြားတဲ့အမြင်ပါ။

လက်တွေ့မှာ Header တွေထက်စာရင် Data Envelope ထဲက အချက်အလက်တွေက ပိုပြီးတော့ အသုံးပြုရ လွယ်လို့ Data Envelope နဲ့ပဲ ပိုပြီးတော့ အသုံးများကြပါတယ်။ Data Envelope ထဲမှာ ဘာတွေပါရမှာလဲဆိုတာကိုတော့ နောက်တစ်ခန်းကျော်ကျတော့မှ ဆက်ကြည့်ကြပါမယ်။

အခန်း (၄) – RESTful API

REST ဆိုတာ Representational State Transfer ရဲ့ အတိုကောက်ဖြစ်ပြီး၊ Service တွေဖန်တီးတဲ့အခါမှာ လိုက်နာဖို့ သတ်မှတ်ထားတဲ့ Architecture Design လမ်းညွှန်ချက်ဖြစ်ပါတယ်။ Roy Fielding လို့ခေါ်တဲ့ ကွန်ပျူတာသိပ္ပံပညာရှင်တစ်ဦးက Ph.D စာတမ်းအဖြစ် (၂၀၀၀) ပြည့်နှစ်မှာ တင်သွင်းခဲ့တာပါ။ ဒါပေမယ့် သူက (၁၉၉၄) ခုနှစ်လောက်တည်းက ဒီနည်းစနစ်တွေကိုသုံးပြီးတော့ HTTP ကို ပူးပေါင်းတီထွင်ပေးခဲ့တာပါ။ ဒါကြောင့် REST က Architecture ဖြစ်ပြီး HTTP က အဲ့ဒီ Architecture ကိုအသုံးပြုထားတဲ့ နည်းပညာလို့ ဆိုနိုင်ပါတယ်။

Service တွေရဲ့သဘောသဘာဝကို ပထမဆုံးအခန်းမှာ လိုတိုရှင်း ပြောခဲ့ပြီးဖြစ်ပါတယ်။ Service တွေနဲ့ ပက်သက်ရင် **RESTful Web Service** လို့ခေါ်တဲ့ အသုံးအနှုန်းတစ်ခုရှိပါတယ်။ ဆိုလိုတာက စနစ်ကျတဲ့ Service တစ်ခုဖြစ်ဖို့ဆိုရင် RESTful ဖြစ်ရမယ်၊ REST Architecture Design လမ်းညွှန်ချက်များနဲ့ ကိုက်ညီရမယ် ဆိုတဲ့ အဓိပ္ပါယ်ပါ။ တစ်ကယ်တော့ HTTP ကိုအသုံးပြုတဲ့ Service ဖြစ်တာနဲ့တင် RESTful Service ဖြစ်ပြီးနေပြီလို့ ဆိုနိုင်ပါတယ်။ HTTP ကိုယ်တိုင်က REST ကို လိုက်နာပြီး တီထွင်ထားတာ မို့လို့ပါ။

RESTful Services

ဒါပေမယ့် HTTP ကို သုံးနေတာပဲ ဆိုယုံလောက်နဲ့ ကျေနပ်လို့မရသေးပါဘူး။ REST ရဲ့ သဘော သဘာဝတွေကို ကိုယ်တိုင်နားလည် ကျင့်သုံးပေးဖို့လည်း လိုအပ်ပါသေးတယ်။ မဟုတ်ရင် HTTP သုံးထားပေမယ့် REST သတ်မှတ်ချက်များနဲ့ မကိုက်ညီတာမျိုး ဖြစ်နေနိုင်ပါတယ်။ စနစ်မကျတော့တာမျိုး ဖြစ်နိုင်ပါတယ်။ ဒါကြောင့် REST ရဲ့ သဘောသဘာဝ အနှစ်ချုပ်ကို အခုလိုမှတ်ပါ။

Client နဲ့ Server တို့အကြား အချက်အလက်တွေ ဖလှယ်တဲ့အခါ အဲ့ဒီအချက်အလက်တွေဟာ **Representable** ဖြစ်ရပါမယ်။ ဆိုလိုတာက အပြန်အလှန်ဖလှယ်ကြတဲ့ အချက်အလက်တွေကို ကွန်ပျူတာစနစ်တွေက နားလည်အလုပ်လုပ်နိုင်သလို၊ လူတွေကလည်း ဖတ်ရှုနားလည်နိုင်ရမယ် ဆိုတဲ့ အဓိပ္ပါယ်ပါ။

ဥပမာ –

`/products/123/reviews` ဆိုတဲ့ URL ကို ကွန်ပျူတာစနစ်ဖြစ်တဲ့ HTTP Server က နားလည်အလုပ် လုပ်နိုင်သလို၊ လူတစ်ယောက်က ဖတ်ကြည့်ရင်လည်း ဘာကိုဆိုလိုတယ်ဆိုတာ နားလည်နိုင်စေပါတယ်။ `/xyz987/a(12)3+b45?q=678#rtyez` ဆိုရင်တော့ ကွန်ပျူတာက နားလည် အလုပ်လုပ်ပေးနိုင်တဲ့ URL ဖြစ်ကောင်းဖြစ်ပေမယ့် လူကတော့ ဖတ်ရှုနားလည်မှာ မဟုတ်တော့ပါဘူး။

URL တင်မကပါဘူး။ ပြန်သုံးသပ်ကြည့်မယ်ဆိုရင် Request Methods တွေ၊ Response Status Code တွေ Headers တွေအားလုံးဟာလည်း ကွန်ပျူတာစနစ်တွေသာမက လူတွေဖတ်ရှု နားလည်နိုင်စွမ်းရှိတဲ့ သတ်မှတ်ချက်တွေ ဖြစ်တယ်ဆိုတာကို တွေ့ရမှာပါ။ ဒါဟာ HTTP က မူလကတည်းက Representable ဖြစ်နေခြင်းဖြစ်တယ်လို့ နားလည်ရမှာဖြစ်ပြီး အဲ့ဒီလို Representable ဖြစ်နေတာကို ကိုယ့်ရဲ့အသုံးပြုပုံအမှားကြောင့် မပျက်စီးစေဖို့ လိုအပ်ပါတယ်။

နောက်ထပ် REST ရဲ့ အရေးပါတဲ့ သဘောသဘာဝကတော့ **Stateless** ဖြစ်ရမယ်ဆိုတဲ့ သတ်မှတ်ချက်ပါတယ်။ HTTP က Stateless ဖြစ်ပြီးသားပါ။ Client က Server ကို ဆက်သွယ်တဲ့အခါ Server က အကြောင်းပြန်ပါတယ်။ ဒါပေမယ့် အဲ့ဒီဆက်သွယ်မှုကို မှတ်မထားပါဘူး။ ဒါကြောင့် နောက်တစ်ကြိမ် ဆက်သွယ်တဲ့အခါ Server က ဆက်သွယ်မှု အသစ်တစ်ခုအနေနဲ့ပဲ လက်ခံအလုပ်လုပ်သွားမှာပါ။ အရင်ကလာဖူးတယ်၊ အဆင်ပြေရဲ့လား၊ စသဖြင့် အဲ့ဒါမျိုးတွေ တစ်ခုမှမရှိဘဲ ဘယ်နှစ်ကြိမ် ဆက်သွယ်မှုကို ပြုလုပ်လာပါစေ တစ်ခါမှ မလာဖူးတဲ့၊ သူစိမ်းတစ်ယောက်လိုပဲ အမြဲဆက်ဆံ တုံ့ပြန်သွားမှာ ဖြစ်ပါတယ်။ ဒီသဘောသဘာဝကို Stateless ဖြစ်တယ်လို့ ဆိုတာပါ။ ဒီလို Stateless ဖြစ်ခြင်းကြောင့်ရလာမယ့် အားသာချက်ပေါင်းများစွာ ရှိပါတယ်။ အမြင်သာဆုံးကတော့ Client အနေနဲ့ အရင်ဆက်သွယ်မှုတွေနဲ့ ပြန်လည်ချိန်ညှိနေစရာ မလိုတော့ဘဲ၊ အသစ်တစ်ခုကဲ့သို့ အမြဲဆက်သွယ်နိုင်တဲ့အတွက် ဆက်သွယ်ရ လွယ်ကူရိုးရှင်းသွားခြင်းပဲ ဖြစ်ပါတယ်။

REST ဆိုတာ Ph.D စာတမ်းကြီးတစ်စောင်ဖြစ်လို့ ကျယ်ပြန့်လှပါတယ်။ ဒါပေမယ့် လက်တွေ့အသုံးပြုဖို့ အတွက်ဆိုရင်တော့ ဒီနှစ်ချက်ကို မှတ်သားထားရင် လုံလောက်ပါပြီ။ (၁) ဆက်သွယ်ပေးပို့တဲ့ အချက်အလက်တွေဟာ ကွန်ပျူတာသာမက လူကပါ ဖတ်ရှုနားလည်နိုင်စွမ်း ရှိရမယ်။ (၂) ဆက်သွယ်မှုတွေဟာ Stateless ဖြစ်နေရမယ်။ ဒါပါပဲ။ ဒီနှစ်ချက်ကို နားလည်လိုက်နာပြီး HTTP ရဲ့ မူလသတ်မှတ်ချက်တွေကို မှန်မှန်ကန်ကန် အသုံးပြုသွားမယ်ဆိုရင် ကျွန်တော်တို့ တည်ဆောက်မယ့် Service တွေဟာ စနစ်ကျတဲ့ RESTful Service တွေ ဖြစ်နေမှာပဲ ဖြစ်ပါတယ်။

RESTful API

API ဆိုတာဟာ Service ကို ဆက်သွယ်အသုံးပြုရန်နည်းလမ်း လို့အပေါ်မှာ ရှင်းခဲ့ပါတယ်။ Web Service တစ်ခုကို ဆက်သွယ်အသုံးပြုလိုတဲ့အခါ URL အပါအဝင် Request တွေကို အသုံးပြုရပါတယ်။ ဒါကြောင့် API ဒီဇိုင်းတစ်ခု RESTful ဖြစ်ဖို့ဆိုတာ ဒီ URL နဲ့ Request တွေရဲ့ဖွဲ့စည်းပုံကို စနစ်ကျအောင် သတ်မှတ်တာပါပဲ။ ဒီလိုသတ်မှတ်တဲ့အခါ မူလ HTTP ရဲ့ သဘောသဘာဝကနေ သွေဖီခြင်းမရှိစေဘဲ ကိုယ့်နည်းကိုယ်ဟန်နဲ့ စိတ်တိုင်းကျ သတ်မှတ်လို့ရနိုင်ပါတယ်။ ဒါပေမယ့် ကနေ့ခေတ်မှာ ဒီလိုကိုယ့်နည်းကိုယ်ဟန်နဲ့ တစ်ကျောင်းတစ်ပုဒ်ဆန်း ထွင်ပြီးလုပ်နေစရာ မလိုတော့ပါဘူး။ လူတိုင်းလိုလိုက လက်ခံပြီး ကျင့်သုံးနေတဲ့ နည်းတွေ ရှိနေပါပြီ။ ဒီနည်းတွေကို ဆက်ပြောသွားမှာပါ။

API URL နဲ့ပက်သက်ရင် အခေါ်အဝေါ် (၃) ခုကို အရင်မှတ်ထားဖို့ လိုပါမယ်။ **Resource**, **Action** နဲ့ **ID** ပါ။ Resource ဟာ ဆောင်ရွက်မယ့် လုပ်ငန်းရင်းမြစ်ဖြစ်ပါတယ်။ Data ဖြစ်ပါတယ်။ ဥပမာအားဖြင့်၊ Products, Customers, Users, Students, Books, Records, Places စသဖြင့် ဖြစ်နိုင်ပါတယ်။ Action ကတော့ အဲ့ဒီ Resource Data ပေါ်မှာ သက်ရောက်မယ့် လုပ်ငန်းတွေပါ။ ဥပမာအားဖြင့် Create Product, View Customers, Update User, Delete Student, Increase Book, Change Record, Cancel Place စသဖြင့် ဖြစ်နိုင်ပါတယ်။ ID ကတော့ စုဖွဲ့ပြီးရှိနေတဲ့ Resource Data Collection ထဲက Specific Resource တစ်ခုကို တိတိကျကျညွှန်းဖို့ အတွက် သတ်မှတ်ထားတဲ့ Unique Key ကိုပြောတာပါ။

API URL တွေ သတ်မှတ်ဖို့အတွက် ဒီ (၃) ချက်ကို စုဖွဲ့တည်ဆောက်ရမှာပါ။ ဒီလိုတည်ဆောက်တဲ့အခါ နားလည်လိုက်နာရမယ့် စည်းမျဉ်းတွေ ရှိပါတယ်။ နည်းနည်းလည်းများပါမယ်။ ဒါပေမယ့် သူလည်းပဲ အရေးကြီးတဲ့ ကိစ္စတစ်ခုမို့လို့ သေသေချာချာဂရုပြုပြီး လေ့လာမှတ်သားပေးပါ။

1. ပထမဆုံးစည်းမျဉ်းကတော့ API URL တွေမှာ စာလုံးအသေးတွေချည်းပဲသုံးရမယ် ဆိုတဲ့အချက် ဖြစ်ပါတယ်။ ဥပမာ -

/products	/articles	/books	/users

2. ဒုတိယအနေနဲ့ Resource အမည်က Plural (အများကိန်း) ဖြစ်ရပါမယ်။ အပေါ်ကဥပမာကိုပြန်ကြည့်ပါ။ Plural Noun တွေကိုပဲသုံးပြီး နမူနာပေးထားတာကို တွေ့ရနိုင်ပါတယ်။

3. Resource အမည်မှာ Special Character တွေနဲ့ Space တွေ ထည့်မသုံးရပါဘူး။ Space လိုအပ်တဲ့အခါ Dash (-) နဲ့တွဲပေးနိုင်ပါတယ်။ ဥပမာ -

/junior-engineers	/language-books	/travel-records	/management-jobs

4. API URL မှာ Action တွေမထည့်ရပါဘူး။ Action အစား သင့်တော်တဲ့ Request Method ကို အသုံးပြုရပါမယ်။ ဒီတော့ Resource, Action, ID ရယ်လို့ (၃) မျိုးရှိပေမယ့် API URL မှာ Resource နဲ့ ID ပဲ ပါရမယ်ဆိုတဲ့ သဘောပါ။

	/products	**/products/123**
GET	Get all products.	Get product with id 123
POST	Create a product	
PUT	Update many products at once	Update product with id 123
PATCH	Update many products at once	Update product with id 123
DELETE	Delete many products	Delete product with id 123

အပေါ်ကဇယားကွက်မှာ URL က နှစ်ခုတည်း ဆိုပေမယ့် Request Method (၅) မျိုးနဲ့ ပေါင်းလိုက်တဲ့အခါ လုပ်ဆောင်ချက် (၉) မျိုးရတာကို တွေ့ရနိုင်ပါတယ်။ လေ့လာကြည့်လိုက်ပါ။ PUT နဲ့ PATCH ရဲ့ ကွဲပြားပုံ ကိုရှင်းပြခဲ့ပြီး ဖြစ်ပါတယ်။ ပြီးတော့ POST နဲ့ အချက်အလက်သစ်တည်ဆောက်တဲ့အခါ ID ပေးပြီး တည်ဆောက်ရင် Conflict ဖြစ်လို့ အဲ့ဒီလုပ်ဆောင်ချက်တစ်ခုချန်ထားခဲ့တာလည်း သတိပြုပါ။ ဒီနည်းနဲ့ Action တွေကို URL မှာ ထည့်စရာမလိုတော့ပါဘူး။

တစ်ချို့ Action တွေကတော့ Request Method သက်သက်နဲ့ အဓိပ္ပါယ် မပေါ်လွင်လို့ မထည့်မဖြစ် ထည့်ဖို့လိုတာတွေ ရှိနိုင်ပါတယ်။ ဥပမာ - `increase`, `toggle`, `tag`, `verify`, `undo` စသဖြင့်ပါ။ ဒါမျိုးလိုအပ်လာရင်တော့ URL မှာ Action ထည့်နိုင်ပါတယ်။ ဥပမာ -

`/product/increase/123`	`/items/toggle/4`	`/users/verify`	`/tasks/undo/5`

5. **Sub Resource** ဆိုတာလည်း လိုအပ်တတ်ပါသေးတယ်။ လိုအပ်ရင် ထည့်လို့ရပါတယ်။ ဥပမာ - Product ရဲ့ Reviews တွေကို လိုချင်တယ်။ Student ရဲ့ Grades ကိုလိုချင်တယ်။ Article ရဲ့ Comments တွေကို လိုချင်တယ်၊ စသဖြင့် ရှိတတ်ပါတယ်။ အဲ့ဒါဆိုရင် URL ရဲ့ ဖွဲ့စည်းပုံက ဒီလို ဖြစ်သွားမှာပါ။

`/products/123/reviews`	`/students/4/grades`	`/articles/5/comments`

နောက်ကလိုက်တဲ့ `reviews`, `grades`, `comments` တွေဟာလည်း Resource တွေပါပဲ၊ ဒါပေမယ့် Main Resource တော့ မဟုတ်ပါဘူး။ Sub Resource တွေပါ။

6. ရှေ့ကနေ `/api` ဆိုတဲ့ Prefix နဲ့ URL တွေကို စပေးချင်ရင် ပေးလို့ရပါတယ်။ ရိုးရိုး URL နဲ့ API URL ကို ကွဲပြားသွားစေလို့ ပေးသင့်ပါတယ်။ မပေးရင်လည်း ရတော့ရပါတယ်။ ကိုယ့်လိုအပ်ချက် ပေါ်မှာ မူတည်ပါတယ်။

7. URL မှာ API Version ပါသင့်ပါတယ် (ဥပမာ `/api/v1`)။ API ဆိုတာ ပေးထားပြီးရင် မပြင်သင့်တော့ပါဘူး။ ပြင်လိုက်လို့ ပြောင်းသွားတဲ့အခါ ဒီ API ကို သုံးနေသူ Client အတွက် ပြဿနာရှိနိုင်ပါတယ်။ ဒါကြောင့် ပြင်ဖို့လိုတယ် ဆိုရင်လည်း ပြင်ပြီနောက်မှာ API Version သစ်အနေနဲ့ ထပ်တိုးပေးရမှာ ဖြစ်ပါတယ်။ ဥပမာ -

`/api/`**`v1`**`/products/123`	`/api/`**`v2`**`/articles/4/comments`	`/api/`**`v3`**`/user/verify/5`

8. Sorting, Filter နဲ့ Paging လုပ်ဆောင်ချက်တွေ အတွက် **URL Query** တွေကို သုံးရပါတယ်။ ဒီအပိုင်းကတော့ တစ်ယောက်နဲ့တစ်ယောက် အကြိုက်မတူသလို Recommendation တွေလည်း ကွဲပြားကြပါတယ်။ ဒါကြောင့် အခုဖော်ပြမယ့်နည်းကို Recommendation များစွာထဲက တစ်ခုလို့ သဘော ထားသင့်ပါတယ်။

   Sorting နဲ့ပတ်သတ်တဲ့ သတ်မှတ်ချက်တွေကို အခုလိုသတ်မှတ်ပေးနိုင်ပါတယ်။

`/products?sort[price]=1`	`/students?sort[name]=1&sort[age]=-1`

   URL Query Operator ဖြစ်တဲ့ ? ကို သုံးလိုက်တာပါ။ ပထမဥပမာမှာ Products တွေကို `price` နဲ့ Sorting စီပြီး လိုချင်တဲ့သဘောကို သတ်မှတ်ပေးလိုက်တာပါ။ ဒုတိယဥပမာ မှာတော့ Students တွေကို `name` နဲ့လည်းစီမယ် `age` နဲ့လည်းစီမယ်ဆိုတဲ့သဘောနဲ့ နှစ်ခုပေး ထားတာကို သတိပြုပါ။ ပြီးတော့ `age` အတွက် Value က `-1` ဖြစ်ပါတယ်။ ပြောင်းပြန် စီမယ် Descending ဆိုတဲ့ သဘောပါ။ ရှိးရှိးစီရင် `1` ကိုသုံးပြီး ပြောင်းပြန်စီရင် `-1` ကိုသုံးတယ် ဆိုတဲ့ ဒီနည်းဟာ ရှေ့ဆက်လေ့လာမယ့် MongoDB ကနေလာတဲ့ နည်းဖြစ်ပါတယ်။

   Filter ကနည်းနည်းတော့ ရှုပ်ပါတယ်။ ပေးထားတဲ့နမူနာကိုသာ တိုက်ရိုက်လေ့လာကြည့်ပါ။

```
/products?filter[name]=Book&filter[price][$lt]=9
```

   Grater Than, Less Than, Not စတဲ့ Filter Operator တွေအတွက် `$gt]` `[$gte]` `[$lt]` `[$lte]` `[$not]` စသဖြင့် လေးထောင့် ကွင်းလေးတွေနဲ့ ရေးကြပါတယ်။ ဒါလည်းပဲ MongoDB ကနေလာတဲ့ နည်းပဲ ဖြစ်ပါတယ်။ အခုအခြေမှာ ရှုပ်နေပေမယ့် လက်တွေ့အသုံးပြုတဲ့ အခါ ဒီနည်းတွေကိုသုံးတဲ့အတွက် နောက်ပိုင်း တော်တော်ရှင်းပြီး အဆင်ပြေတယ်ဆိုတာကို တွေ့ ရပါလိမ့်မယ်။

Paging ကတော့နှစ်မျိုးရှိပါတယ်။ Client ဘက်က Paging ကို သတ်မှတ်ခွင့် ပြုလို့ရသလို Server ဘက်ကပဲ သတ်မှတ်ပေးလို့လည်း ရပါတယ်။ Client ဘက်က Paging သတ်မှတ်ခွင့်ပြုချင်ရင် API URL ရဲ့ ဖွဲ့စည်းပုံက အခုလိုဖြစ်နိုင်ပါတယ်။ ရိုးရိုး Database Query တွေမှာသုံးတဲ့ `limit` ရေးထုံးမျိုးကိုပဲ ပြန်သုံးလိုက်တာပါ။

`/products?skip=5&limit=10`	`/students?skip=10&limit=20`

Client ဘက်က ဘယ်ကစမလဲ ရွေးလို့ရသလို၊ တစ်ကြိမ်မှာ ဘယ်နှစ်ခုလိုချင်သလဲပါရွေးလို့ရသွားမှာပါ။ နောက်တစ်နည်းကတော့ Paging Options တွေကို Server ဘက်က ပုံသေသတ်မှတ်ထားတာမျိုးပါ။ ဒီလိုပါ။

`/products?page=2`	`/students?page=3`

Client ဘက်ကလိုချင်တဲ့ အရေအတွက်တွေဘာတွေ ပေးခွင့်မရှိတော့ပါဘူး။ လိုချင်တဲ့ Page Number ကိုပဲပေးရတဲ့ သဘောပါ။ Page တစ်ခုမှာ Record ဘယ်နှစ်ခုရှိမလဲဆိုတာမျိုးကတော့ Server ကသတ်မှတ်မှာပါ။

Filter, Sorting နဲ့ Paging ပေါင်းသုံးထားတဲ့ ဥပမာလေးတစ်ခုကိုလည်း ဖော်ပြပေးလိုက်ပါတယ်။

```
/products?filter[category]=4&sort[name]=1&sort[price]=-1&page=2
```

`category` က ကိုလိုချင်တယ်။။ `name` နဲ့ Sorting စီမယ်။ `price` နဲ့လည်း ပြောင်းပြန်ထပ်စီဦးမယ်။ Page 2 ကို လိုချင်တယ်လို့ ပြောလိုက်တာပါ။

ဒီလို Sorting, Filter, Paging လုပ်ဆောင်ချက်တွေ အကုန်လုံးကို ကိုယ့် API က မဖြစ်မနေ ပေးရမယ်လို့ မဆိုလိုပါဘူး။ ပေးချင်တယ်ဆိုရင် ဘယ်လိုပေးရမလဲဆိုတဲ့ နည်းကိုသာပြောတာပါ။ လက်တွေ့မှာ ကိုယ့် API ဘက်က ဘယ်လိုလုပ်ဆောင်ချက်တွေ ပေးမလဲဆိုတာ ကိုယ်တိုင် ရွေးချယ်သတ်မှတ်ရမှာ ဖြစ်ပါတယ်။

ဒီထက်ပိုပြီး နည်းနည်းအသေးစိတ်ချင်သေးတယ်ဆိုရင် Client ဘက်က လိုချင်တဲ့ Fields တွေကို ရွေးယူလို့ရအောင်လည်း ပေးလို့ရပါတယ်။ ဒီလိုပါ။

`/products?fields=name,price,description`	`/students?fields=name,age,grade`

ပထမနမူနာမှာ Products တွေကိုလိုချင်တာပါ။ ဒါပေမယ့် Products ရဲ့ ရှိသမျှ Field အားလုံးကို မယူဘဲ `name, price` နဲ့ `description` ပဲလိုချင်တယ်လို့ ရွေးပေးလိုက်တာပါ။ ဒုတိယ နမူနာကလည်း အတူတူပါပဲ။ Students ရဲ့အချက်အလက်တွေထဲကမှ `name, age` နဲ့ `grade` ကိုပဲ လိုချင်တယ်ဆိုတဲ့ အဓိပ္ပါယ်ပါ။

ဒီလောက်ဆိုရင် RESTful API URL တွေရဲ့ ဖွဲ့စည်းပုံပိုင်းမှာ တော်တော်လေး ပြည့်စုံသွားပါပြီ။ ခေါင်းလည်း နည်းနည်းမူးသွားကြပြီ ထင်ပါတယ်။ မှတ်စရာ နည်းနည်းများတဲ့ အတွက်ကြောင့် မှတ်ရလွယ်အောင် နံပါတ်စဉ်တပ်ပြီး ဖော်ပြပေးခဲ့ပါတယ်။ အားလုံး (၈) ပိုင်းရှိပါတယ်။

စာတွေများလို့ စိတ်တော့ ညစ်မသွားပါနဲ့။ ဒီနှိုင်းပိုင်း ကောင်းမွန်စနစ်ကျတဲ့ Service တွေ API တွေဖြစ်စေဖို့အတွက် နောက်ထပ်အရေးကြီးတဲ့ အပိုင်းတစ်ခုကို ဆက်လက်လေ့လာ ကြရပါဦးမယ်။ အဲ့ဒါကတော့ Response တွေရဲ့ ဖွဲ့စည်းပုံအကြောင်းပဲ ဖြစ်ပါတယ်။

အခန်း (၅) – Response Structure

ဒီအခန်းမှာတော့ အထူးသဖြင့် Response Body ရဲ့ ဖွဲ့စည်းပုံနဲ့အတူ Client နဲ့ Server အပြန်အလှန် ပေးပို့ကြမယ့် Request / Response တွေရဲ့ ဖွဲ့စည်းပုံအကြောင်းကို ပြောပြသွားမှာပါ။ ဟိုးအပေါ်မှာ Response Body ဟာ JSON Format ဖြစ်ရပြီးတော့ Data Envelope ကိုအသုံးပြုနိုင်တယ်လို့ ပြောထားပါတယ်။

JSON Format ဖြစ်ဖို့က လွယ်ပါတယ်။ ကိုယ်တောင် ဘာမှလုပ်စရာ မလိုပါဘူး။ Language တိုင်းလိုလိုမှာ Array ကို JSON ပြောင်းပေးနိုင်တဲ့ လုပ်ဆောင်ချက်တွေ၊ Object ကို JSON ပြောင်းပေးနိုင်တဲ့ လုပ်ဆောင်ချက်တွေ ပါပြီးသားပါ။

ဒါကြောင့် ဒီနေရာမှာ Data Envelope ရဲ့ဖွဲ့စည်းပုံကို နားလည်ဖို့ကပိုအရေးကြီးပါတယ်။ Data Envelope ရဲ့ ဖွဲ့စည်းပုံက တစ်ယောက်နဲ့တစ်ယောက် အကြိုက်မတူကြသလို၊ ပေးကြတဲ့ Recommendation တွေလည်းမတူကြပါဘူး။ ဒါပေမယ့် အသေးစိတ်တွေမှာသာ ကွာသွားတာပါ၊ အခြေခံမူသဘောတွေကတော့ အတူတူပါပဲ။ ဒါကြောင့်မို့လို့ အခုဖော်ပြမယ့်နည်းတွေကို လေ့လာထားပြီး ကျန်အသေးစိတ်ကိုတော့ ကိုယ့်လိုအပ်ချက်နဲ့အညီ ချိန်ညှိပြီး သုံးသွားလို့ ရနိုင်ပါတယ်။

ပထမဆုံးအနေနဲ့ Response Envelope ထဲမှာ `meta`, `data`, `links` နဲ့ `errors` လို့ခေါ်တဲ့ ပင်မအစိတ်အပိုင်း (၄) ခု ပါလေ့ရှိတယ်လို့ မှတ်သားထားပေးပါ။ ဥပမာ - `GET` Method ကို သုံးပြီး `/students` ကို Request ပြုလုပ်ခဲ့တယ်ဆိုရင် အခုလိုဖြစ်နိုင်ပါတယ်။

Request

```
GET /students
Content-Type: application/json
```

Response

```
200 OK
Content-Type: application/json

{
    meta: {
        total: 15
    },
    data: [
        { ... }, { ... }, { ... }, { ... }, { ... }
    ],
    links: {
        self: "http://domain/students",
        students: "http://domain/students",
        classes: "http://domain/classes",
        grades: "http://domain/students/grades"
    }
}
```

အထက်က Response Body နမူနာမှာ `meta`, `data` နဲ့ `links` တို့ပါပါတယ်။ `meta` မှာ စုစုပေါင်း Record အရေအတွက်ကို `total` နဲ့တွဲထည့်ပေးထားပါတယ်။ `data` ကတော့ Client လိုချင်တဲ့ Student စာရင်းကို JSON Array အနေနဲ့ပေးထားတာပါ။ `errors` မပါပါဘူး။ Error မရှိတဲ့သဘောပါ။ တစ်ချို့က `errors` နဲ့ `data` နှစ်ခုထဲက တစ်ခုပဲပါရမယ်။ `errors` ပါရင် `data` မပါရဘူး။ `data` ပါရင် `errors` မပါရဘူး လို့ပြောကြပါတယ်။ ဒါသဘာဝကျပါတယ်။ ဒါကြောင့် `errors` နဲ့ `data` နှစ်ခုရှိပေမယ့် နှစ်ခုထဲကတစ်ကြိမ်မှာ တစ်ခုသာ ပါဝင်ရမှာပဲ ဖြစ်ပါတယ်။

`links` မှာတော့ အသုံးဝင်တဲ့ API URL တွေကို တန်းစီပြီးတော့ ပေးထားတဲ့ သဘောပါ။ ဒီအတွက် **HATEOAS** လို့ခေါ်တဲ့ REST ရဲ့ဆက်စပ်နည်းပညာတစ်ခုရှိပါတယ်။ Hypermedia as the Engine of Application State ရဲ့အတိုကောက်ပါ။ ဒါနဲ့ပက်သက်ပြီး နည်းနည်းတော့ အငြင်းပွားကြပါတယ်။ လိုတယ်လို့ ပြောတဲ့သူရှိသလို၊ မလိုဘူးလို့ ပြောတဲ့သူလည်း ရှိပါတယ်။ မလိုဘူးလို့ ဘာကြောင့် ပြောသလဲဆိုတော့၊ လက်တွေ့မှာ ဒီ Links တွေကိုအားကိုးလို့ မပြည့်စုံပါဘူး။ API Documentation ကိုသွားကြည့်ရမှာပဲမို့လို့

Response Body မှာ ထည့်ပေးနေစရာမလိုဘူးလို့ ပြောကြတာပါ။ တစ်ကယ်တမ်း လက်တွေ့မှာလည်း အမြဲတမ်း ပြည့်စုံအောင်လိုက်ထည့်ပေးဖို့ မလွယ်တဲ့အတွက် အများအားဖြင့် Pagination ပိုင်းကလွဲရင် `links` ကို သိပ်ပြီးတော့ မသုံးကြပါဘူး။ ဒါကြောင့် နောက်ပိုင်းမှာ `links` ကို အမြဲမသုံးတော့ဘဲ လိုအပ်မှပဲ သုံးပါတော့မယ်။

`GET` Method နဲ့ `/studetns/:id` ဆိုတဲ့ API URL ကို Request ပြုလုပ်ခဲ့ရင်တော့ ဖွဲ့စည်းပုံက အခုလိုပုံစံ ဖြစ်နိုင်ပါတယ်။

Request

```
GET /students/3
Content-Type: application/json
```

Response

```
200 OK
Content-Type: application/json

{
    meta: {
        id: 3
    },
    data: { ... }
}
```

ဒီတစ်ခါတော့ `meta` မှာ `id` ကို ထည့်ပေးထားပါတယ်။ တစ်ချို့လည်း `total` တို့ `id` တို့လို `meta` Data တွေကို `meta` ထဲမှာ တစ်ဆင့်ခံ မထည့်တော့ပဲ တိုက်ရိုက်ပေးတတ်ကြပါတယ်။ ဒီလိုပါ။

```
{
    id: 3,
    data: { ... }
}
```

နည်းလမ်းတွေ ရောပြောနေလို့ ခေါင်းမစားပါနဲ့။ အထက်မှာပြောခဲ့သလို အသေးစိတ်လေးတွေပဲ ကွာသွားတာပါ။ အခြေခံမူသဘောက အတူတူပဲမို့လို့ ကြိုက်တဲ့နည်းကို သုံးနိုင်ပါတယ်။ လက်ရှိနမူနာမှာ `data` ကတော့ JSON Object တစ်ခုတည်း ဖြစ်သွားပါပြီ။ Array မဟုတ်တော့ဘူးဆိုတာကိုလည်း သတိပြုပါ။

အကယ်၍များ Error ဖြစ်ခဲ့ရင်တော့ Response က ဒီလိုဖြစ်နိုင်ပါတယ်။

Response

```
404 Not Found
Content-Type: application/json

{
     meta: {
          id: 3
     },
     errors: {
          message: "Student with id 3 doesn't exists."
     }
}
```

သို့မဟုတ် ဒီလိုလည်း ဖြစ်နိုင်ပါတယ်။

Response

```
401 Unauthorized
Content-Type: application/json

{
     meta: {
          id: 3
     },
     errors: {
          message: "Unauthorize access."
     },
     links: {
          self: "http://domain/studnets/3",
          login: "http://domain/login"
     }
}
```

နမူနာမှာ Client အတွက် အသုံးဝင်နိုင်တဲ့အတွက် `links` ကိုထည့်ပေးထားတာကို သတိပြုပါ။ Sorting တွေ Filter တွေ Paging တွေ အကုန်ရောပါတဲ့ နမူနာလေး တစ်ခုလောက် ထပ်ပေးပါဦးမယ်။

Request

```
GET /students?filter[grade]=7&sort[name]=1&page=2
Content-Type: application/json
```

Response

```
200 OK
Content-Type: application/json

{
    meta: {
          filter: { grade: 7 },
          sort: { name: 1 },
          page: 2,
          total: 25,
          limit: 5
    },
    data: [
          { ... }, { ... }, { ... }, { ... }, { ... }
    ],
    links: {
         self: "/students?filter[grade]=7&sort[name]=1&page=2",
         first: "/students?filter[grade]=7&sort[name]=1&page=1",
          last: "/students?filter[grade]=7&sort[name]=1&page=5",
          next: "/students?filter[grade]=7&sort[name]=1&page=3",
          prev: "/students?filter[grade]=7&sort[name]=1&page=1"
    }
}
```

ဒီနမူနာမှာတော့ `meta`, `links` အစုံပါသွားပါပြီ။ ပါဖို့လည်းလိုပါတယ်။ ဒီတော့မှ Client က Pagination နဲ့ပက်သက်တဲ့ လုပ်ဆောင်ချက်တွေကို Data နဲ့အတူ တစ်ခါတည်း အတွဲလိုက် သိရမှာပါ။ အခုလိုသာ Data Envelope ထဲမှာစနစ်တကျ အချက်အလက် ပြည့်ပြည့်စုံစုံ ပြန်ပေးမယ်ဆိုရင် Client အတွက် အသုံးပြုရတာ အရမ်းအဆင်ပြေသွားမှာပါ။

တစ်ချို့ API URL တွေက နည်းနည်းရှုပ်လို့ Client ဘက်က မှားနိုင်ပါတယ်။ အဲ့ဒီလိုမှားပြီဆိုရင်တော့ ဒီလို Error ပေးနိုင်ပါတယ်။

Response

```
400 Bad Request
Content-Type: application/json

{
     errors: {
          message: "Unable to parse parameters."
     }
}
```

ဆက်လက်ပြီးတော့ `POST` Method နဲ့ Request ပြုလုပ်လာတဲ့အခါ အပြန်အလှန်သွားလေ့ရှိကြတဲ့ နမူနာလေးတစ်ခု ဖော်ပြပေးပါမယ်။

Request

```
POST /students
Content-Type: application/json

{
     name: "Tom", age: 12, grade: 6
}
```

Response

```
201 Created
Content-Type: application/json
Location: /students/9

{
     meta: {
          id: 9
     },
     data: {
          id: 9, name: "Tom", age: 12, grade: 6
     }
}
```

POST Method နဲ့လာလို့ အချက်အလက်သစ် ထည့်သွင်းပေးရတဲ့အတွက် Response မှာ Status Code အနေနဲ့ `201 Created` ကို ပြန်ပေးထားပါတယ်။ ထည့်သွင်းလိုက်တဲ့အတွက် ရလာတဲ့ ID ကို Response မှာ ပြန်ထည့်ပေးထားတာကို သတိပြုပါ။ ဒီနည်းနဲ့ Client က `POST` လုပ်ပြီးတာနဲ့ အချက်အလက်သစ် သိမ်းပေးသွားယုံသာမက၊ Auto ID ကိုပါ တစ်ခါတည်း ပြန်ရသွားမှာ ဖြစ်ပါတယ်။

Response မှာ `Location` Header ကိုသုံးပြီး ထည့်သွင်းလိုက်တဲ့ အချက်အလက်ကို ပြန်ကြည့်လို့ရတဲ့ API URL ကိုပါ ပြန်တွဲထည့်ပေးထားတာကို သတိပြုပါ။ `links` နဲ့ Body ထဲမှာ ထည့်ရင်လည်း ရနိုင်ပါတယ်။ HTTP Standard အရဆိုရင်တော့ `201` နဲ့အတူ `Location` ကိုတွဲပေးရတယ်ဆိုတဲ့ သတ်မှတ်ချက်ရှိလို့ ဒီနေရာမှာတော့ Location Header နဲ့တွဲပေးတာက ပိုစနစ်ကျမှာပါ။

အကယ်၍ပေးတဲ့အချက်အလက်မပြည့်စုံရင် `400 Bad Request` ကိုပြန်ပေးနိုင်ပါတယ်။ ထည့်ခွင့်မရှိဘဲ လာထည့်နေတာဆိုရင် `401 Unauthorized` နဲ့ `403 Forbidden` တို့ကို သင့်တော်သလို ပြန်ပေးနိုင်ပါတယ်။ အကယ်၍ ID ပေးပြီး ထည့်ခိုင်းနေရင်တော့ `409 Conflict` ကို ပြန်ပေးရမှာပါ။ ဒီလိုပါ။ ဒီသဘောသဘာဝတွေကို Status Code တွေအကြောင်း ပြောတုံးက ရှင်းပြခဲ့ပြီး ဖြစ်ပါတယ်။

Request

```
POST /students/8
Content-Type: application/json

{
    name: "Tom", age: 12, grade: 6
}
```

Response

```
409 Conflict
Content-Type: application/json

{
    errors: {
        message: "Incorrect request. Giving ID."
    }
}
```

အချက်အလက်တွေကို `PUT` နဲ့ပေးပို့လာတဲ့အခါ နဂို Data ကို ပေးလာတဲ့ Data နဲ့ အစားထိုးပြီး ပြင်ပေးရမှာပါ။ `PATCH` နဲ့ပေးပို့လာရင်တော့ တစ်စိတ်တစ်ပိုင်း ပြင်ပေးရမှာပါ။ ဥပမာ – မူလဒီလိုပုံစံမျိုးနဲ့ Data ရှိနေတယ်လို့ သဘောထားပါ။

```
{ id: 8, name: "Tom", age: 12, grade: 6 }
```

အခဲ့ဒါကို `PATCH` နဲ့ Request လုပ်ပြီးပြင်မယ်ဆိုရင် ဒီလိုဖြစ်မှာပါ။

Request

```
PATCH /students/8
Content-Type: application/json

{
    name: "Tommy"
}
```

ပေးလိုက်တဲ့ Request Body မှာ `name` တစ်ခုပဲပါတာကို သတိပြုပါ။ Update လုပ်တဲ့အလုပ် လုပ်ပြီးနောက် ပြန်ရမယ့် Response က ဒီလိုပုံစံဖြစ်မှာပါ။

Response

```
200 OK
Content-Type: application/json

{
    meta: {
        id: 8
    },
    data: {
        id: 8, name: "Tommy", age: 12, grade: 6
    }
}
```

မူလ Data မှာ ရှိနေတဲ့အချက်အလက်တွေထဲက `name` တစ်ခုတည်းကို ရွေးပြီးပြင်ပေးသွားတာပါ။ ကျန်အချက်အလက်တွေကိုတော့ မူလအတိုင်းပဲ ဆက်ထားပေးပါတယ်။ အကယ်၍အလားတူ လုပ်ဆောင်ချက်ကိုပဲ `PUT` နဲ့ Request လုပ်ခဲ့မယ်ဆိုရင် ဒီလိုဖြစ်မှာပါ။

Request

```
PUT /students/8
Content-Type: application/json

{
    name: "Tommy"
}
```

Response

```
200 OK
Content-Type: application/json

{
     meta: {
          id: 8
     },
     data: {
          id: 8, name: "Tommy"
     }
}
```

နဂိုရှိတဲ့ Data တွေကို ပေးလိုက်တဲ့ Data နဲ့အစားထိုးလိုက်တာမို့လို့ နဂို Data တွေမရှိတော့တာကို သတိပြုကြည့်ပါ။ ဒီလိုနှစ်လမ်းနှစ်မျိုးရဲ့ ကွဲပြားပုံကို သဘောပေါက်ဖို့လိုပါတယ်။ ကိုယ့် Service နဲ့ API ဘက်က နှစ်မျိုးလုံးကို ကွဲပြားအောင် Implement လုပ်ပေးထားနိုင်ရင်တော့ အကောင်းဆုံးပါပဲ။

ဖြည့်စွက်မှတ်သားသင့်တဲ့ ထူးခြားချက်အနေနဲ့ မရှိတဲ့ Data ကို `PUT/PATCH` တို့နဲ့ ပြင်ဖို့ကြိုးစားရင် အသစ်ထည့်ပေးရမယ်လို့လည်း တစ်ချို့ကပြောကြပါတယ်။ မဖြစ်မနေ အသစ်ထည့်ပေးရမယ်လို့ မဆိုလိုပါဘူး။ ဆန္ဒရှိရင်ထည့်ပေးနိုင်တယ်ဆိုတဲ့ သဘောပါ။ မထည့်ချင်ရင်လည်း `404 Not Found` ကို ပြန်ပေးလိုက်လို့လည်း ရပါတယ်။ မရှိတဲ့ Data ကို ပြင်ဖို့ကြိုးစားလို့ အသစ်အနေနဲ့ ထည့်ပေးလိုက်ချင်တယ်ဆိုရင်တော့ ဒီလိုဖြစ်မှာပါ။

Request

```
PUT /students/9
Content-Type: application/json

{
     name: "Mary", age: 12, grade: 6
}
```

Response

```
202 Accepted
Content-Type: application/json

{
     meta: {
          id: 9
     },
     data: {
          id: 9, name: "Mary", age: 12, grade: 6
     }
}
```

200 OK တို့ 201 Created တို့ကို မသုံးဘဲ၊ 202 Accepted ကိုသုံးတာသတိပြုပါ။ အဓိပ္ပါယ်က၊ ပေးပို့တာ မမှန်ပေမယ့် လက်ခံပေးလိုက်တယ်ဆိုတဲ့ အဓိပ္ပါယ်ပါ။ Status Code မှာတင် လုပ်ပေးလိုက်တဲ့ အလုပ်ရဲ့ အဓိပ္ပါယ် ပေါ်လွင်သွားစေတာပါ။ ဒါကြောင့် Request Method တွေ Status Code တွေကို HTTP က သတ်မှတ်ထားတဲ့အတိုင်း မှန်ကန်အောင် သုံးပေးရတယ်လို့ ပြောတာပါ။ အားလုံးက သူ့နေရာနဲ့ သူ အဓိပ္ပါယ်ရှိပြီး စနစ်ကျနေစေမှာပါ။

ဆက်လက်ပြီး DELETE Request တွေလာတဲ့အခါ တုံ့ပြန်ပုံကို ဖော်ပြပေးပါမယ်။ ဒီလိုပါ။

Request

```
DELETE /students/8
```

Request

```
204 No Content
```

ဒါပါပဲ၊ အရမ်းရှင်းပါတယ်။ Request မှာ Body မပါသလို၊ Response မှာလည်း Body မပါပါဘူး။ ဘာမှမပါလို့ ကိစ္စမရှိပါဘူး။ 2XX Status Code ဖြစ်ကတည်းက လုပ်ငန်းအောင်မြင်တာ ပေါ်လွင်နေပါပြီ။ 204 လို့ပြောတဲ့အတွက် Content မရှိတော့လို့ ပြန်မပေးတာ ဖြစ်တဲ့အတွက် Delete ဖြစ်သွားတာလည်း ပေါ်လွင်ပါတယ်။ အကယ်၍ ဒီနည်းကို မကြိုက်ဘူးဆိုရင်လည်း ရိုးရိုး 200 OK နဲ့ပဲ meta မှာ messages ပေးပြီး အခုလိုတုံ့ပြန်နိုင်ပါတယ်။

Request

```
DELETE /students/8
```

Response

```
200 OK
Content-Type: application/json

{
    meta: {
        id: 8,
        message: "Successfully deleted."
    }
}
```

ဒီလောက်ဆိုရင် Response Structure နဲ့ပက်သက်ပြီး တော်တော်ပြည့်စုံသွားပါပြီ။ ဖြည့်စွက်ချက်အနေနဲ့ Data Envelope ထဲမှာ `relationships` နဲ့ `included` ဆိုပြီးတော့လည်း ဆက်စပ်အချက်အလက်တွေကို ထည့်ကြလေ့ရှိတယ်ဆိုတာကို မှတ်သားပေးပါ။

Relationship Data

Data ဟာ Database တွေကနေ လာလေ့ရှိပါတယ်။ Database ထဲမှာ သိမ်းတဲ့အခါ Table Relationship တွေရှိနိုင်ပါတယ်။ အဲ့ဒီလိုရှိနေတဲ့ Relationship Data တွေကို တစ်ခါတည်း ထည့်ပေးချင်ရင် ပေးလို့ရအောင်လို့ပါ။ ပြီးတော့ တစ်ချို့ Data တွေမှာ သူနဲ့တွဲသုံးဖို့ လိုအပ်နိုင်တဲ့ ဆက်စပ် Data တွေ ရှိတတ်ပါတယ်။ အဲ့ဒီလို ရှိတဲ့အခါ `included` ဆိုပြီးတော့ တစ်ခါတည်း တွဲပေးလိုရင် ပေးနိုင်တဲ့ သဘောမျိုးပါ။ ဒီလိုပါ။

```
{
    meta: {
        id: 1
    },
    data: {
        attributes: {
            title: "API Book",
            author: "Ei Maung",
            price: 4.99,
            category_id: 2
        },
```

```
            relationships: {
                  category: {
                        id: 2,
                        name: "Technology"
                  }
            }
      },
      included: {
            author: {
                  name: "Ei Maung",
                  bio: " ... "
            }
      }
}
```

မူလ Data တွေက `attributes` ဖြစ်သွားပြီး Relationship Data တွေက `relationships` ထဲကို ရောက်သွားတာပါ။

Table Relationship စနစ်ကို မသုံးတဲ့ MongoDB လို NoSQL Database တွေကိုလည်း သုံးတတ်ကြပါသေးတယ်။ အဲ့ဒီလို NoSQL Database တွေကနေ လာတာဆိုရင်တော့ Data တွေရဲ့ ဖွဲ့စည်းပုံကတော့အခုလို ဖြစ်နိုင်ပါတယ်။

```
{
      meta: {
            id: 1
      },
      data: {
            title: "API Book",
            author: "Ei Maung",
            price: 4.99,
            category: {
                  id: 2,
                  name: "Technology"
            }
      },
      included: {
            author: {
                  name: "Ei Maung",
                  bio: " ... "
            }
      }
}
```

ပါတဲ့ Data တွေက အတူတူပါပဲ။ ဘာကွာသွားတာလဲဆိုတော့ `attributes` တွေ `relationships` တွေ မပါတော့ဘဲ၊ အချက်အလက်အားလုံးကို တစ်စုတစ်စည်းထဲ တွဲထည့်ပေးလိုက်တာပဲ ကွာသွားတာပါ။ ကိုယ့် Response မှာ Relationship Data တွေ ထည့်ပေးဖို့လိုရင် ဒီနှစ်နည်းထဲက ကြိုက်တဲ့နည်းကို သုံးနိုင်ပါတယ်။

Rate Limiting

Request / Response တွေရဲ့ ဖွဲ့စည်းပုံနဲ့ ပက်သက်ရင် နောက်ဆုံးတစ်ခုအနေနဲ့ မှတ်သားသင့်တာကတော့ Rate Limit နဲ့ ပက်သက်တဲ့ Headers တွေပါ။ API တွေမှာ Rate Limit ထားရပါတယ်။ Rate Limit မထားရင်၊ Client တွေက အထိမ်းအကွပ်မရှိ Request ဝိုင်းလုပ်ကြတဲ့အခါ API Server ကို Abuse လုပ်သလိုဖြစ်ပြီး Server က မနိုင်ဝန်ထမ်းရပါလိမ့်မယ်။ ဒါကြောင့် Client တစ်ခုကို တစ်မိနစ်မှာ Request အကြိမ် (၆၀) သာလုပ်ခွင့်ရှိတယ်ဆိုတာမျိုးပါ။ ဥပမာအနေနဲ့ ပြောတာပါ။ ဘယ်လောက် ကန့်သတ်မလဲဆိုတာကတော့ ကိုယ့် Server ရဲ့ Capacity တို့ ကိုယ့် Service ရဲ့ Processing Power လိုအပ်ချက်တို့ ကိုယ့် User တွေရဲ့ လိုအပ်ချက်တို့နဲ့ ချင့်ချိန်ပြီး သတ်မှတ်ထားရမှာပါ။

API ဘက်က ဒီလိုကန့်သတ်ထားကြောင်း Client ကို ထည့်သွင်းအသိပေးနိုင်ပါတယ်။ ဒီအတွက် Custom Header (၃) ခုကို မှတ်သားသင့်ပါတယ်။ `X-Rate-Limit-Limit`, `X-Rate-Limit-Remaining` နဲ့ `X-Rate-Limit-Reset` ပါ။

ဒါတွေက Standard HTTP Response Headers တွေ မဟုတ်ကြပါဘူး။ API ဖန်တီးသူဘက်က ကိုယ့် အစီအစဉ်နဲ့ကိုယ် Client သိစေဖို့ ထည့်ပြောပေးတဲ့ Custom Headers တွေပါ။ `X-Rate-Limit-Limit` ကိုသုံးပြီး တစ်မိနစ်မှာ Request အကြိမ်ရေ ဘယ်လောက် ကန့်သတ်ထားသလဲ အသိပေးနိုင်ပါတယ်။ `X-Rate-Limit-Remaining` ကို သုံးပြီးတော့ အကြိမ်ရေ ဘယ်လောက်ကျန်သေးလဲ အသိပေးနိုင်ပါတယ်။ `X-Rate-Limit-Reset` ကိုသုံးပြီးတော့ နောက်အချိန် ဘယ်လောက်ကြာတဲ့အခါ Limit ကို Reset ပြန်လုပ်ပေးမလဲဆိုတာကို ပြောပြနိုင်ပါတယ်။ ဒီလိုပုံစံ ဖြစ်မှာပါ။

Response

```
200 OK
Content-Type: application/json
X-Rate-Limit-Limit: 60
X-Rate-Limit-Remaining: 58
X-Rate-Limit-Reset: 30
```

ဒီ Headers အရ၊ တစ်မိနစ်မှာ Request အကြိမ်အရေအတွက်ကို (၆၀) လို့ ကန့်သတ်ထားပြီး နောက်ထပ် (၅၈) ကြိမ် ကျန်သေးတယ်ဆိုတဲ့ အဓိပ္ပါယ်ရပါတယ်။ စက္ကန့် (၃၀) အကြာမှာ Reset ပြန်လုပ်မှာဖြစ်လို့ နောက်စက္ကန့် (၃၀) ကြာတဲ့အခါ၊ မူလသတ်မှတ်ချက်အတိုင်း အကြိမ် (၆၀) ပြန်ဖြစ်သွားမယ်ဆိုတဲ့ အဓိပ္ပါယ်ပါ။

ဒီလောက်ဆိုရင် API ဒီဇိုင်းပိုင်းနဲ့ပက်သက်ပြီး Client ဘက်ခြမ်း၊ API ဘက်ခြမ်း နဲ့ Server ဘက်ခြမ်း သိသင့်တာတွေ စုံသလောက် ရှိသွားပါပြီ။ ဒီလိုမျိုးသဘောသဘာဝတွေကို ကိုယ့်ဘက်သိပြီဆိုရင် လက်တွေ့မှာ ကြိုက်တဲ့ Language နဲ့ ကြိုက်တဲ့ Framework ကိုသုံးပြီး API တွေ ဖန်တီးလို့ရပါတယ်။ PHP မှာဆိုရင် Laravel လို Full-fledged Framework တွေရှိသလို Slim လို့ API Framework တွေလည်း ရှိပါတယ်။ Ruby မှာဆိုရင်လည်း Rails လို Framework ကြီးတွေရှိသလို Sinatra လို Framework လေးတွေလည်း ရှိပါတယ်။ Python မှာဆိုရင်လည်း Django လို့ Framework ကြီးတွေရှိသလို Flask လို Framework လေးတွေ ရှိပါတယ်။ JavaScript မှာဆိုရင်တော့ ExpressJS လို မူလအစဉ်အလာ Framework တွေရှိသလို NextJS တို့လို ခေတ်ပေါ် Server-side Rendering နည်းပညာတွေ ရှိပါတယ်။ ကြိုက်တဲ့နည်းပညာကိုသုံးပါ။ ဟို Language နဲ့မှ ဖန်တီးလို့ရတယ်၊ ဒီ Framework နဲ့မှ အဆင်ပြေတယ် ဆိုတာမျိုး မရှိတော့ပါဘူး။ ကိုယ်ကျွမ်းကျင်ရာ၊ နှစ်သက်ရာ နည်းပညာကို ရွေးချယ်ပြီး၊ အခုသိရှိလေ့လာခဲ့ကြတဲ့ သဘောသဘာဝတွေနဲ့ ပေါင်းစပ်လိုက်မယ်ဆိုရင် API တွေကို ဖန်တီးလို့ ရပြီပဲဖြစ်ပါတယ်။

ဒီစာအုပ်မှာတော့ အခုသိရှိလေ့လာခဲ့ကြတဲ့ သဘောသဘာဝတွေကို လက်တွေ့အသုံးချကြည့်တဲ့ သဘောနဲ့ ExpressJS ကို အဓိကထားအသုံးပြုပြီး နမူနာတွေ ဆက်လက်ရေးသားဖော်ပြပေးသွားမှာ ဖြစ်ပါတယ်။

အခန်း (၆) – MongoDB

ရှေ့ပိုင်းမှာ API ဒီဇိုင်းနဲ့ပက်သက်ပြီး သိသင့်တာတွေ ပြောလို့ပြီးသွားပါပြီ။ အခုအခဲ့ဒီ ဗဟုသုတွေကို လက်တွေ့အသုံးချပြီး ဆက်လက်လေ့လာနိုင်ဖို့ ပရောဂျက်လေးတစ်ခုလုပ်ကြည့်ပါမယ်။ အခုချိန်ကစပြီးတော့ ဖတ်ယုံမဖတ်တော့ဘဲ၊ အဆင့်လိုက် တစ်ပြိုင်တည်း လိုက်လုပ်ကြည့်ပါလို့ တိုက်တွန်းပါတယ်။ ဒါမှ တစ်ခါတည်းရပြီး တစ်ခါတည်း မှတ်မိသွားမှာပါ။

အခုဒီစာအုပ်ကိုရေးတဲ့အချိန်မှာ Covid-19 ကပ်ရောဂါ ဖြစ်နေချိန်ဆိုတော့၊ အချိန်ကာလနဲ့ ကိုက်ညီတဲ့ ပရောဂျက်လေးတစ်ခု လုပ်ကြမှာပါ။ ခရီးသွားမှတ်တမ်းပရောဂျက်လေးပါ။ သိပ်ကြီးကြီးကျယ်ကျယ်ကြီး မဟုတ်သလို၊ ခက်လည်းမခက်ပါဘူး။ လူတစ်ယောက်ခရီးသွားတော့မယ်ဆိုရင် သူ့ရဲ့အမည်၊ မှတ်ပုံတင် အမှတ် စတဲ့အချက်အလက်နဲ့အတူ ဘယ်ကနေ၊ ဘယ်ကိုသွားမှာလည်း၊ စီးသွားမယ့် ကားနံပါတ်က ဘယ်လောက်လဲ စသဖြင့် အချက်အလက်တွေကို သိမ်းထားလိုက်ပါမယ်။ နောက်ပိုင်းမှာ လူတစ်ယောက်ရဲ့ ခရီးသွားမှတ်တမ်းကို သိချင်ရင် နာမည်တို့ မှတ်ပုံတင်နံပါတ်တို့နဲ့ ပြန်ရှာကြည့်လို့ရပါမယ်။ မြို့အမည်နဲ့ ထုတ်ယူလိုက်ရင် အဲ့ဒီမြို့ကို ဝင်ထားတဲ့လူစာရင်း၊ ထွက်ထားတဲ့ လူစာရင်းကို ပြန်ရပါမယ်။ လက်တွေ့ အသုံးချဖို့ထက် နမူနာလုပ်ကြည့်ဖြစ်လို့ လုံးဝပြီးပြည့်စုံဖို့ထက် စမ်းသင့်တာ စုံအောင်စမ်းဖြစ်ဖို့ကို ဦးစားပေးသွားမှာပါ။

ပရောဂျက်တစ်ခု လုပ်တော့မယ်ဆိုတော့ အဲ့ဒီပရောဂျက်ရဲ့ အချက်အလက်တွေကို သိမ်းဆည်းနိုင်မယ့် Database နည်းပညာကို အရင်ဆုံး ကြည့်ကြပါမယ်။ MongoDB လို့ ခေါ်တဲ့ နည်းပညာကို အသုံးပြုမှာပါ။ MongoDB ဟာ NoSQL Database တစ်မျိုးဖြစ်ပါတယ်။ Document Database လို့ခေါ်ပါတယ်။ အကျဉ်းချုပ်အားဖြင့် ရိုးရိုး SQL Database တွေမှာ ကြိုတင်တည်ဆောက်ထားတဲ့ Table တွေထဲမှာ Data တွေကို သိမ်းကြပါတယ်။ MongoDB မှာတော့ Table နဲ့ Table Structure တွေ မရှိဘဲ Record တစ်ခုကို Document တစ်ခုပုံစံမျိုးနဲ့ သိမ်းသွားမှာပါ။ Table Structure မရှိဘဲ သိမ်းမှာဆိုပါတော့။ ဒါပေမယ့် အဲ့ဒီ

လို ကြိုတင်သတ်မှတ်ထားတဲ့ Structure မရှိပေမယ့်၊ သိမ်းထားတဲ့ Data တွေကို SQL Database တွေမှာလိုပဲ Create, Read, Update, Delete တွေ လုပ်လို့ရပါမယ်။ ပြန်ရှာလို့ရပါမယ်။ Sorting တွေ စီလို့ရပါမယ်။ Filter တွေ လုပ်လို့ရပါမယ်။ Index တွေလုပ်လို့ရပါမယ်။ သိပ်မရှင်းရင် ကိစ္စမရှိပါဘူး။ ခဏနေလက်တွေ့စမ်းကြည့်လိုက်တဲ့အခါ သဘောပေါက်သွားပါလိမ့်မယ်။

စိတ်ဝင်စားဖို့ကောင်းပေမယ့် NoSQL Database တွေရဲ့ သဘောသဘာဝပိုင်းကို ဒီနေရာမှာ ကြားဖြတ်ပြီး မပြောတော့ပါဘူး။ Rockstar Developer စာအုပ်မှာ အကျယ်တစ်ဝင့် ရေးသားထားပြီးဖြစ်ပါတယ်။ လေ့လာကြည့်ဖို့ တိုက်တွန်းပါတယ်။ NoSQL Database အမျိုးမျိုးရဲ့ သဘောသဘာဝတွေ၊ သူတို့ရဲ့ အားသာချက် အားနည်းချက်တွေ ကနေ စပြီးတော့ Replication နဲ့ Cluster တည်ဆောက်ပုံလို အဆင့်မြင့် Server Architecture ပိုင်းတွေထိ ဖော်ပြထားပါတယ်။ ဒီနေရာမှာတော့ လက်တွေ့အသုံးပြုပုံပိုင်းကို ရွေးထုတ်ဖော်ပြသွားမှာပါ။

Install MongoDB

MongoDB အတွက် Installer ကို mongodb.com Website မှာ Download လုပ်လို့ရပါတယ်။ သူမှာ Product အမျိုးမျိုးရှိလို့ **Community Server** ကို ရွေးချယ်ပြီး Download လုပ်ရမှာပါ။ ဒီစာရေးနေချိန်မှာ Community Server Download လုပ်ဖို့အတွက် သွားလို့ရတဲ့ တိုက်ရိုက်လင့်ကို ဖော်ပြပေးလိုက်ပါတယ်။

– https://www.mongodb.com/try/download/community

Ubuntu Linux လို OS မျိုးမှာတော့ ကိုယ့်ဘာသာ Download လုပ်စရာမလိုပါဘူး။ `apt` ကို သုံးပြီး အခုလို Install လုပ်လို့ ရနိုင်ပါတယ်။

```
sudo apt install mongodb
```

`apt` နဲ့ Install လုပ်ရင်ရမယ့် Version နဲ့ Website ကနေ တိုက်ရိုက် Download လုပ်ရင်ရမယ့် Version နံပါတ် အနည်းငယ်တော့ ကွာနိုင်ပါတယ်။ အသုံးပြုပုံ သိပ်မကွာလို့ ကိစ္စမရှိပါဘူး။ လက်ရှိဒီစာရေးနေချိန်မှာ ရောက်ရှိနေတဲ့ နောက်ဆုံး Version ကတော့ 4.4.1 ပါ။ Install လုပ်တဲ့အဆင့်တွေကိုတော့ တစ်ဆင့်ချင်း မပြောတော့ပါဘူး။ Installer မှာပေါ်တဲ့ ညွှန်ကြားချက်တွေအတိုင်းပဲ သွားလိုက်ပါ။

Install လုပ်ပြီးနောက် ရလာမယ့်ထဲမှာ အထူးသတိပြုရမယ့် အရာနှစ်ခုကတော့ **mongod** နဲ့ **mongo** တို့ပဲ ဖြစ်ပါတယ်။ mongod က MongoDB Server ဖြစ်ပြီးတော့ mongo က MongoDB Client ပါ (Mongo Shell လို့ ခေါ်ပါတယ်)။ ဒါကြောင့် MongoDB Server ကို Run ချင်ရင် mongod ကို Run ပေးရမှာဖြစ်ပြီး တော့၊ Run ထားတဲ့ Server ကို ချိတ်သုံးချင်တယ်ဆိုရင် mongo နဲ့ ချိတ်သုံးရမှာပါ။ Mongo Compass လို့ ခေါ်တဲ့ GUI ပရိုဂရမ်တစ်ခုလည်း Install လုပ်စဉ်မှာ တစ်ခါတည်း ပါလာနိုင်ပေမယ့်၊ အဲ့ဒါကို နောက်ပိုင်း MongoDB အကြောင်း သိသွားပြီဆိုရင် ကိုယ့်ဘာသာ စမ်းသုံးသွားလို့ ရနိုင်ပါလိမ့်မယ်။ အခုဒီစာအုပ်မှာ တော့ ပိုပြီးအခြေခံကျတဲ့ mongo Shell ကနေ အသုံးပြုပုံကို ဖော်ပြသွားမှာပါ။

Windows မှာ mongod နဲ့ mongo ရှိနေတဲ့ ဖိုဒါကို PATH Environment Variable ထဲမှာ ကိုယ့်ဘာသာ ထည့်ပေးဖို့ လိုကောင်းလိုနိုင်ပါတယ်။ PATH သတ်မှတ်ပုံကိုတော့ ဒီနေရာမှာ ထည့်မပြောတော့ပါဘူး။ သိ မယ်လို့ ယူဆပါတယ်။ မသိသေးရင်လည်း ကြားဖြတ်ပြီး Windows မှာ PATH Environment Variable ဘယ်လို သတ်မှတ်ရလဲဆိုတာကို အရင်လေ့လာကြည့်လိုက်ပါ။ လက်ရှိဒီစာကို ရေးသားနေချိန်မှာ Default Installation ဖိုဒါက C:\Program Files\MongoDB\Server\4.0\bin ဖြစ်ပါတယ်။ သူကို PATH ထဲမှာ ထည့်ထားပေးဖို့ လိုတာပါ။ ဒီတော့မှ Windows ရဲ့ ဘယ်နေရာကနေမဆို mongod နဲ့ mongo တို့ကို Run လို့ရမှာပါ။

Install လုပ်စဉ်ကရွေးခဲ့တဲ့ Option ပေါ်မူတည်ပြီး mongod Server က System Service အနေနဲ့ အလို အလျောက် Run ပြီး ဖြစ်နိုင်ပါတယ်။ ဒီလို Server အလိုအလျောက် Run နေပြီဆိုရင်တော့ mongo Command ကို Command Prompt မှာ Run ပြီးချိတ်ကြည့်လို့ရပါတယ်။ အကယ်၍ ချိတ်လို့မရဘူးဆိုရင် mongod Server Run မနေလို့ ဖြစ်နိုင်ပါတယ်။ Windows မှာအခုလို mongod Server ကို System Service အနေနဲ့ Install လုပ်ပေးလိုက်လို့ ရပါတယ်။

```
mongod --install
```

ပြီးတဲ့အခါ၊ စက်ကို Restart လုပ်ပေးလိုက်မယ်ဆိုရင် နောက်ပိုင်း စက်ပြန်တက်ချိန်မှာ mongod Server က System Service အနေနဲ့ အလိုအလျောက် Run နေမှာပါ။ အကယ်၍ Service အနေနဲ့ Install မလုပ် ချင်ဘူးဆိုရင်လည်း ဒီလို Manual Run ပေးလို့ ရပါတယ်။

```
mongod --dbpath=C:\mongodata
```

C:\mongodata ဆိုတဲ့ ဖိုဒါကိုတော့ အရင်ကြိုဆောက်ထားပေးဖို့ လိုပါတယ်။ ဒါမှမဟုတ် တစ်ခြားနေရာမှာ ဖိုဒါဆောက်ထားပြီး C:\mongodata အစား အစားထိုးပေးမယ် ဆိုရင်လည်း ရပါတယ်။ MongoDB က Data တွေကို အဲ့ဒီဖိုဒါထဲမှာ သွားသိမ်းမှာပါ။ Server Run သွားပြီဆိုရင် Run ထားတဲ့ Command Prompt ကို ဒီအတိုင်းဆက်ဖွင့်ထားပေးရမှာပါ၊ ပိတ်လိုက်လို့မရပါဘူး။ ပိတ်လိုက်ရင် Server လည်း လိုက်ပိတ်သွားမှာပါ။ ဒါကြောင့် ဒီလို Manual Run တာက သိပ်အဆင်မပြေပါဘူး။ တစ်ခါသုံးချင်တိုင်း တစ်ခါ Run နေရမှာမို့လို့ အထက်မှာပြောထားသလို System Service အနေနဲ့ Install လုပ်ထားလိုက်တာကတော့ ပိုကောင်းပါတယ်။

`mongod` Server ကို Run နေပြီဆိုရင်တော့ Command Prompt မှာပဲ `mongo` Command နဲ့ Server ကို အခုလို လှမ်းချိတ်လိုက်လို့ရပါပြီ။

```
mongo

MongoDB shell version v3.6.8
connecting to: mongodb://127.0.0.1:27017
…
MongoDB server version: 3.6.8
…
2020-09-16T10:34:24.094+0630 I CONTROL  [initandlisten]
>
```

ဒါဆိုရင် Mongo Shell ထဲကို ရောက်ရှိသွားပြီဖြစ်လို့ Server မှာရှိတဲ့ Database နဲ့ Data တွေကို စတင်စီမံနိုင်ပြီပဲ ဖြစ်ပါတယ်။

CRUD

ပရောဂျက်အတွက် နမူနာ Data တွေထည့်ရင်းနဲ့ MongoDB မှာ Create, Read, Update, Delete လုပ်ငန်းတွေ ဘယ်လိုလုပ်ရလဲဆိုတာ လေ့လာကြည့်ကြပါမယ်။ ပထမဦးဆုံးအနေနဲ့ `show dbs` Command ကို စမ်းကြည့်ပါ (`mongo` Shell မှာစမ်းရမှာပါ)။ လက်ရှိ ရှိနေတဲ့ Database စာရင်းကို တွေ့ရပါလိမ့်မယ်။

```
> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
```

`admin, config, local` ဆိုတာ နဂိုကတည်းက ရှိနေတဲ့ Database တွေပါ။ `use` Command နဲ့ ကိုယ်အသုံးပြုလိုတဲ့ Database ကို ရွေးလို့ရပါတယ်။ မရှိသေးတဲ့ Database ကို ရွေးလိုက်ရင်တော့ အလိုအလျောက် Database အသစ် ဆောက်ပေးသွားမှာပါ။

```
> use travel
switched to db travel

> show dbs
admin   0.000GB
config  0.000GB
local   0.000GB
```

`use travel` လို့ ပြောလိုက်တဲ့အတွက် `travel` အမည်နဲ့ Database ကို ရွေးရမှာပါ။ မရှိတဲ့အတွက် အသစ်ဆောက် ပေးသွားပါလိမ့်မယ်။ ဒါပေမယ့် `show dbs` ကို ချက်ခြင်းခေါ်ကြည့်ရင် Database စာရင်းထဲမှာ `travel` မပါသေးတာကို သတိပြုနိုင်ပါတယ်။ အခုတစ်ကယ် မဆောက်သေးဘဲ Data တွေ စသိမ်းပြီ ဆိုတော့မှ Database ကို တစ်ခါတည်းဆောက်ပြီး တစ်ခါတည်း သိမ်းသွားပေးမှာပါ။ `use` နဲ့ Database တစ်ခုကို ရွေးပြီး/ဆောက်ပြီးပြီဆိုရင်၊ `db` ကနေတစ်ဆင့် Data သိမ်းတာတွေ၊ ပြင်တာတွေ၊ ဖျက်တာတွေ လုပ်လို့ရပါပြီ။

ရိုးရိုး SQL Database တွေမှာ Data တွေကို Table နဲ့ သိမ်းပါတယ်။ MongoDB မှာ Data တွေကို Collection နဲ့သိမ်းပါတယ်။ ကွာသွားတာက Table မှာ Columns နဲ့ Structure ရှိပြီး Collection မှာတော့ ဘာ Structure မှ မရှိဘဲ Data တွေ သိမ်းလို့ ရပါတယ်။

Collection တစ်ခု တည်ဆောက်လိုရင် `db.createCollection("name")` နဲ့ ဆောက်လို့ရပါတယ်။ `name` နေရာမှာ မိမိနှစ်သက်ရာ Collection အမည်ကို သတ်မှတ်ပေးနိုင်ပါတယ်။ Collection တွေ ပြန်ဖျက်လိုရင်တော့ `db.dropCollection("name")` ကို သုံးပြီး ဖျက်နိုင်ပါတယ်။ ထုံးစံအတိုင်း `name` နေရာမှာ ဖျက်လိုတဲ့ Collection အမည်ကို ပေးရမှာပါ။ Collection စာရင်းကို ကြည့်ချင်ရင်တော့ `show collections` နဲ့ ကြည့်လို့ရပါတယ်။

Collection တည်ဆောက်နည်း နောက်တစ်နည်းကတော့ Data သာ ထည့်လိုက်ပါ။ သူဘာသာ လိုတဲ့ Database တွေ Collection တွေကို အလိုအလျောက် ဆောက်ပေးသွားပါတယ်။ အဲ့ဒီနည်းကို အခုသုံးပါမယ်။ ဒီလိုပါ –

```
> use travel
switched to db travel

> db.records.insert({ name: "Bobo", age: 23 })
WriteResult({ "nInserted" : 1 })
```

`use travel` နဲ့ `travel` Database ကို သုံးပါတယ်။ ပြီးတဲ့အခါ `db.records.insert()` နဲ့ Data ထည့်လိုက်တဲ့အတွက် `records` အမည်ရ Collection ထဲကို Data တစ်ကြောင်း သိမ်းသွားမှာပါ။ Document တစ်ခု သိမ်းသွားတယ်လို့ ပြောမှတိကျပါမယ်။ ဒါပေမယ့် အခေါ်အဝေါ်တွေ ရှုပ်ကုန်မှာစိုးလို့ ပို မြင်သာမယ်လို့ ထင်တဲ့ Data လို့ပဲ ဆက်ပြောသွားပါမယ်။ Data ကို ပေးတဲ့အခါ JSON Format နဲ့ပဲ ပေးရပါတယ်။ တစ်ကယ့် အခေါ်အဝေါ်အမှန်ကတော့ BSON ခေါ် Binary JSON ပါ။ ဒါပေမယ့် အသုံးပြုနည်းမှာ သိပ်မကွာလို့ စာဖတ်သူနဲ့ ပိုရင်းနှီးပြီး ဖြစ်နိုင်တဲ့ JSON လို့ပဲ ဆက်သုံးသွားပါမယ်။

မူလက `travel` Database တို့ `records` Collection တို့ မရှိလည်း ကိစ္စမရှိပါဘူး။ MongoDB က တစ်ခါတည်း အလိုအလျောက်ဆောက်ပြီး သိမ်းပေးသွားမှာပါ။ သိမ်းထားတဲ့ Data တွေကို ပြန်ကြည့်ချင်ရင် `db.collection.find()` ကို သုံးရပါတယ်။ `collection` နေရာမှာ Collection အမည်မှန်နဲ့ အစားထိုးပေးလိုက်ယုံပါပဲ။ ဒီလိုပါ –

```
> db.records.find()
{ "_id" : ObjectId("5f62433057f76967c5d976ee"), "name" : "Bobo", "age" : 23 }
```

မြင်တွေ့ရတဲ့အတိုင်း စောစောကပေးလိုက်တဲ့ Data ကိုပြန်ရပါတယ်။ လွယ်လွယ်လေးပါ။ `insert()` နဲ့ ထည့်ပြီး `find()` နဲ့ပြန်ထုတ်ကြည့်ရတာပါ။ ထူးခြားချက်အနေနဲ့ Auto ID တစ်ခုလည်း ထွက်သွားတယ် ဆိုတာတွေ့ရနိုင်ပါတယ်။ `_id` လို့ ရှေ့ကနေ Underscore လေးခံပြီး ပေးပါတယ်။ UUID ခေါ် Universally Unique ဖြစ်တဲ့ ID ကို MongoDB က ထုတ်ပေးသွားတာပါ။ အဲ့ဒီ ID ရဖို့အတွက် MongoDB ရဲ့ အရင် Version တွေက Timestamp, Machine ID, Process ID နဲ့ Auto Increment Counter ဆိုတဲ့ အချက် (၄) ချက်ကိုပေါင်းပြီး ဖန်တီးယူပါတယ်။ နောက်ပိုင်း Version တွေမှာတော့ Timestamp, Random Value နဲ့ Auto Increment Counter တို့ကို ပေါင်းပြီး ဖန်တီးယူပါတယ်။ ဘယ်လိုပဲဖြစ်ဖြစ် Unique ဖြစ်တဲ့ ID တစ်ခုကို ရလိုက်တာပါပဲ။

UUID Value ကို `ObjectId()` ဆိုတဲ့ Function တစ်ခုနဲ့ ထည့်ပေးတာကိုလည်း သတိပြုပါ။ အဲ့ဒီ `ObjectId()` ရဲ့ အကူအညီနဲ့ UUID ထဲကနေ Timestamp တွေဘာတွေ လိုအပ်တဲ့အခါ ပြန်ထုတ်ယူလို့ ရပါတယ်။ ဆိုလိုတာက တစ်ချက်ခုတ် နှစ်ချက်ပြတ်ပါ။ တစ်ခုတည်းနဲ့ Unique ဖြစ်တဲ့ ID လည်းရမယ်၊ Timestamp လည်းရမယ်ဆိုတဲ့ သဘောပါ။ UUID ထဲကနေ Timestamp ကို အခုလိုပြန်ထုတ်ယူလို့ ရပါတယ်။

```
> ObjectId("5f62481e8873b1d7bff76272").getTimestamp()
```

ထည့်ထားတဲ့ Data တွေကို ပြန်ဖျက်ချင်ရင် `remove()` ကိုသုံးနိုင်ပါတယ်။ ဘာကိုဖျက်မှာလဲဆိုတဲ့ Filter ကို JSON နဲ့ ပေးရပါတယ်။ ဒီလိုပါ –

```
> db.records.remove({ name: "Bobo" })
WriteResult({ "nRemoved" : 1 })
```

တစ်ကယ်တန်း တိတိကျကျဖျက်ချင်ရင် `_id` ကိုပေးပြီး ဖျက်သင့်ပါတယ်။ အခုတော့ နမူနာအနေနဲ့ `name: Bobo` ကို ဖျက်ပေးပါလို့ Filter အနေနဲ့ ပေးလိုက်တာပါ။

ဆက်လက်ပြီးတော့၊ စမ်းစရာ Data ရအောင် နမူနာ Data တစ်ချို့ ထည့်လိုက်ပါမယ်။ တစ်ကြောင်းချင်း ထည့်စရာမလိုပါဘူး။ JSON Array အနေနဲ့ စုစည်းပြီး ပေးလိုက်လို့ရပါတယ်။ ဒီလိုပါ –

```
> db.records.insert([
{name: "Bobo", nrc: "A0131", from: "Yangon", to: "Mandalay", with: "5B9876"},
{name: "Nini", nrc: "A1476", from: "Yangon", to: "Bago", with: "3G6457"},
{name: "Coco", nrc: "B0487", from: "Bago", to: "Yangon", with: "4L2233"},
{name: "Mimi", nrc: "C1987", from: "Yangon", to: "Mandalay", with: "9E4343"},
{name: "Nono", nrc: "B0098", from: "Bago", to: "Yangon", with: "4L2233"},
{name: "Momo", nrc: "C0453", from: "Yangon", to: "Bago", with: "3G6457"}
])

BulkWriteResult({
     "writeErrors" : [ ],
     "writeConcernErrors" : [ ],
     "nInserted" : 6,
     "nUpserted" : 0,
     "nMatched" : 0,
     "nModified" : 0,
     "nRemoved" : 0,
     "upserted" : [ ]
})
```

ထည့်ထားတဲ့ Data တွေကို `find()` နဲ့ ပြန်ထုတ်ကြည့်ရင် Data တွေများလာပြီမို့လို့ ကြည့်ရတာ မျက်စိရှုပ်စရာ ဖြစ်နေတာကို တွေ့ရပါလိမ့်မယ်။ အဲ့ဒါကိုကြည့်ရတာ အဆင်ပြေစေချင်ရင် အခုလို `pretty()` နဲ့ ထပ်တွဲပြီး ကြည့်လို့ရပါတယ်။

```
> db.records.find().pretty()
{
     "_id" : ObjectId("5f62481e8873b1d7bff76271"),
     "name" : "Bobo",
     "nrc" : "A0131",
     "from" : "Yangon",
     "to" : "Mandalay",
     "with" : "5B9876"
}
{
     "_id" : ObjectId("5f62481e8873b1d7bff76273"),
     "name" : "Coco",
     "nrc" : "B0487",
     "from" : "Bago",
     "to" : "Yangon",
     "with" : "4L2233"
}
…
```

ဆက်လက်ပြီးတော့ Sorting, Filter နဲ့ Paging တွေဆက်ကြည့်ကြပါမယ်။ Filter အတွက်ကတော့ စောစောက `remove()` လိုပါပဲ၊ `find()` ကို JSON နဲ့ Filter တန်ဖိုးတွေ ပေးလို့ရပါတယ်။ ဒီလိုပါ –

```
> db.records.find({ from: "Yangon" })
```

ဒါဆိုရင် `from: Yangon` တန်ဖိုးနဲ့ ကိုက်ညီတဲ့ Data တွေကိုပဲ ပြန်ရမှာပါ။ Filter ကို တစ်ခုထက်ပိုပေးလို့လည်း ရပါတယ်။ ဥပမာ –

```
> db.records.find({ from: "Yangon", to: "Bago" })
```

ဒါဆိုရင် `from: Yangon` နဲ့ `to: Bago` ဆိုတဲ့ တန်ဖိုးနှစ်ခုလုံးနဲ့ကိုက်တဲ့ Data တွေကိုပဲ ပြန်ပေးမှာပါ။ `AND` Filter လို့ ပြောလို့ရပါတယ်။ `OR` Filter ပုံစံလိုချင်ရင်တော့ ရေးရတာ နည်းနည်းရှုပ်သွားပါတယ်။ ဒီလိုပါ။

```
> db.records.find({
    $or: [
        { from:"Yangon" },
        { to: "Yangon" }
    ]
})
```

`$or` Operator အတွက် Filter လုပ်ချင်တဲ့တန်ဖိုးတွေ တန်းစီပေးလိုက်ရတာပါ။ Greater Than တို့ Less Than တို့လို Filter တွေလည်းရပါတယ်။ ဒီလိုပုံစံရေးရပါတယ်။

```
> db.products.find({ price: { $gt: 9 } })
```

`products` Collection မရှိလို့ လက်တွေ့ရေးစမ်းဖို့ မဟုတ်ပါဘူး။ နမူနာ ပြတာပါ။ `$gt` အစား `$lt`, `$gte`, `$lte`, `$not` စသဖြင့် သုံးလို့ရပါတယ်။ စဉ်းစားကြည့်ရင် API ဒီဇိုင်းအကြောင်း ပြောခဲ့တုံးက နဲ့ သဘောသဘာဝတွေ ခပ်ဆင်ဆင်ပဲဆိုတာကို တွေ့ရနိုင်တယ်။

Sorting စီချင်ရင်တော့ ဒီလိုစီရပါတယ်။

```
> db.records.find().sort({ name: 1 })
```

`name: 1` လို့ပြောလိုက်တာဟာ `name` နဲ့စီမယ်လို့ ပြောလိုက်တာပါ။ စမ်းကြည့်လို့ရပါတယ်။ Sorting ကို Field တစ်ခုထက်လည်း ပိုစီလို့ရပါတယ်။

```
> db.records.find().sort({ from: 1, name: 1 })
```

ဒါဆိုရင် `from` နဲ့အရင်စီပြီးနောက် `name` နဲ့ ထပ်စီပေးမှာပါ။ ပြောင်းပြန် Descending ပုံစံ စီချင်ရင် 1 အစား -1 လို့ ပေးလို့ရပါတယ်။

Paging လုပ်ဆောင်ချက်အတွက် `skip()` နဲ့ `limit()` ကို သုံးနိုင်ပါတယ်။

```
> db.records.find().limit(3)
```

ဒါဆိုရင် `limit(3)` လို့ပြောထားလို့ Data (၃) ကြောင်းပဲ ယူပေးမှာပါ။

```
> db.records.find().skip(1).limit(3)
```

ဒါဆိုရင်တော့ `skip(1)` လို့ပြောထားလို့ (၁) ကြောင်းကျော်လိုက်ပြီးမှ (၃) ကြောင်းယူပေးသွားမှာပါ။

ပြီးတော့ SQL Query တွေမှာ `SELECT name, nrc FROM records` ဆိုပြီး ကိုယ်လိုချင်တဲ့ Fields ကိုပဲ ရွေးယူလို့ ရသလိုပဲ MongoDB မှာလည်း ရပါတယ်။ `find()` ရဲ့ ဒုတိယ Parameter မှာ လိုချင်တဲ့ Fields စာရင်းပေးပြီးတော့ ဒီလိုယူရပါတယ်။

```
> db.records.find({}, {name: 1, nrc: 1})
```

ဒါဆိုရင် `name` နဲ့ `nrc` နှစ်ခုကိုပဲ ယူလိုက်တာပါ။ ကျန်တဲ့ `from` တွေ `to` တွေ `with` တွေ မပါတော့ပါဘူး။ `find()` အတွက် ပထမ Parameter က Filter မို့လို့ မပေးချင်တဲ့အတွက် အလွတ်ပဲ ပေးခဲ့တာကိုလည်း သတိပြုပါ။

ဆက်ပြီးတော့ Update ကိုကြည့်ပါမယ်။ HTTP အကြောင်းပြောတုံးက Update ပုံစံနှစ်မျိုး ရှိတာ ပြောခဲ့ပါတယ်။ `PUT` နဲ့ `PATCH` ပါ။ တစ်ခုလုံးအစားထိုးပြီး Update လုပ်တာနဲ့ တစ်စိတ်တစ်ပိုင်းရွေးပြီး Update လုပ်တာဆိုပြီး နှစ်မျိုးရှိတာပါ။ MongoDB မှာလည်း နှစ်မျိုးလုံး ရှိပါတယ်။ `save()` နဲ့ `update()` ပါ။ `save()` က `PUT` လိုမျိုး တစ်ခုလုံးအစားထိုးပြီး Update လုပ်မှာပါ။ `update()` ကတော့ `PATCH` လိုမျိုး တစ်စိတ်တစ်ပိုင်း ရွေးပြီး Update လုပ်ပေးမှာပါ။

`save()` ကအလုပ်နှစ်မျိုး လုပ်ပါတယ်။ ပေးလိုက်တဲ့ Data ထဲမှာ `_id` ပါရင် Update လုပ်ပေးပြီး၊ `_id` မပါရင် Create လုပ်ပေးပါတယ်။ စမ်းသပ်နိုင်ဖို့အတွက် နမူနာတစ်ခုလောက် အခုလို ထပ်ထည့်ပြီး စမ်းကြည့်နိုင်ပါတယ်။

```
> db.records.save({ name: "Test", age: 22 })
WriteResult({ "nInserted" : 1 })

> db.records.find({ name: "Test" })
{ "_id" : ObjectId("5f6250f28873b1d7bff76277"), "name" : "Test", "age" : 22 }
```

ပေးလိုက်တဲ့ Data မှာ `_id` မပါတဲ့အတွက် `save()` က အသစ်တစ်ခု ထပ်ထည့်ပေးသွားတာကို တွေ့ရမှာဖြစ်ပါတယ်။ `_id` ပေးပြီး နောက်တစ်ခါ `save()` လုပ်ကြည့်ပါမယ်။

```
> db.records.save({ "_id" : ObjectId("5f6250f28873b1d7bff76277"), "name" :
"Test2" })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })

> db.records.find({ name: "Test2" })
{ "_id" : ObjectId("5f6250f28873b1d7bff76277"), "name" : "Test2" }
```

ဒီတစ်ခါတော့ `save()` က အသစ်မထည့်တော့ဘဲ Update လုပ်ပေးသွားတယ်ဆိုတာကို တွေ့ရနိုင်ပါတယ်။ ဒါပေမယ့် မူလရှိနေတဲ့ `age` တန်ဖိုး ပျောက်သွားတာကို သတိပြုကြည့်ပါ။ `save()` က ပေးလိုက်တဲ့ Data နဲ့ နဂိုရှိတာကို အစားထိုးလိုက်တာမို့လို့ နဂိုရှိတာ ပျောက်သွားတာပါ။ အစမ်းထည့်ထားတဲ့ Data ကို အခုလို ပြန်ဖျက်ထားနိုင်ပါတယ်။

```
> db.records.remove({ name: "Test2" })
```

`update()` ကို ဆက်ပြီးတော့စမ်းကြည့်ကြပါမယ်။ `update()` က ရေးရတာ နည်းနည်းတော့ ရှုပ်ပါတယ်။ Parameter (၃) ခုပေးရပါတယ်။ Filter, Data နဲ့ Option ပါ။ ဒီလိုရေးရပါတယ်။

```
> db.records.update(
    { to: "Bago" },
    { $set: { to: "Bagan" } },
    { multi: true }
)
WriteResult({ "nMatched" : 3, "nUpserted" : 0, "nModified" : 3 })
```

နမူနာမှာ `to: Bago` က Filter ပါ။ Data ကိုတော့ `$set` Operator နဲ့ ပေးထားပါတယ်။ `to` ကို `Bagan` လို့ ပြင်ချင်တာပါ။ နောက်ဆုံးက `multi: true` Option ကတော့ Filter က တစ်ကြောင်းထက်ပိုတွေ့ရင်လည်း တွေ့သမျှအကုန်ပြင်မယ်လို့ ပြောလိုက်တာပါ။ ဒါကြောင့်ရှိသမျှ `to: Bago` တွေ အကုန် `to: Bagan` ဖြစ်သွားမှာပါ။ ရွေးပြင်သွားတာဖြစ်လို့ တစ်ခြားမဆိုင်တဲ့ Data တွေလည်း မပျောက်ပါဘူး။

Data တွေကိုသိမ်းတဲ့အခါ JSON Structure အတိုင်း သိမ်းရတာဖြစ်လို့ ဖွဲ့စည်းပုံက ဒီလိုအဆင့်ဆင့် ဖြစ်မယ်ဆိုရင်လည်း ဖြစ်လို့ရပါတယ်။

```
{
      name: "Bobo",
      nrc: "A0131",
      trip: {
            from: "Yangon",
            to: "Mandalay",
            vehicle: {
                  type: "car",
                  license: "5B9876"
            }
      }
}
```

ဒီလိုဖွဲ့စည်းပုံမှာ အဆင့်ဆင့်ပါလာပြီဆိုရင် Filter တွေလုပ်တဲ့အခါ ဘယ်လိုလုပ်ရလဲ ဆိုတာလေး ဖြည့်မှတ်ထားပေးပါ။

```
> db.records.find({ "trip.vehicle.type": "car" })
```

`trip` ထဲက `vehicle` ထဲက `type` တန်ဖိုးနဲ့ Filter လုပ်ချင်တာဖြစ်လို့ `trip.vehicle.type` ဆိုပြီးတော့ Dot ခံပြီး ရွေးပေးရတာပါ။

ဒီလောက်ဆိုရင် MongoDB မှာ Data တွေ Create, Read, Update, Delete လုပ်ငန်းတွေ စုံအောင်လုပ်တတ်သွားပါပြီ။ MongoDB က လက်တွေ့လေ့လာအသုံးပြုရတာ လွယ်ပါတယ်။ လူသန်းပေါင်းများစွာက အသုံးပြုပြီး အချက်အလက် သန်းပေါင်းများစွာကို သိမ်းဆည်းရတဲ့ ပရောဂျက်မျိုးတွေနဲ့ သင့်တော်အောင် ဖန်တီးထားတဲ့ နည်းပညာမို့လို့ တစ်ကယ်တမ်း ခက်မှာက အဲ့ဒီလောက် Data တွေသိမ်းနိုင်တဲ့ Server Architecture ကို တည်ဆောက်ပုံ တည်ဆောက်နည်းတွေက ခက်တာပါ။ လောလောဆယ် အဲ့ဒီလောက် အဆင့် မသွားသင့်သေးပါဘူး။ အခုလို အခြေခံအသုံးပြုပုံလောက် သိတယ်ဆိုရင်ပဲ စတင်အသုံးပြုလို့ ရနေပါပြီ။ စိတ်ဝင်စားလို့ အခုကတည်းက လေ့လာချင်တယ်ဆိုရင်လည်း Rockstar Developer စာအုပ်မှာ လေ့လာကြည့်နိုင်ပါတယ်။ PDF Ebook ကို အခမဲ့ ဒေါင်းလို့ရပါတယ်။

– https://eimaung.com/rockstar-developer/

အခန်း (၇) – NodeJS

နမူနာပရောဂျက်ကို ExpressJS အသုံးပြုပြီး တည်ဆောက်မှာပါ။ ဟိုအရင်ကတော့ JavaScript နည်းပညာတွေကို သူ့ထက်ပိုထင်ရှားတဲ့ အခြားအမည်တူနည်းပညာတွေ ရှိနေရင် ကွဲပြားစေဖို့အတွက် နောက်ကနေ NodeJS, ExpressJS စသဖြင့် JS လေးတွေထည့်ပြီး ခေါ်ကြလေ့ရှိပါတယ်။ အခုတော့ သူတို့က ပိုထင်ရှားသွားကြပြီ။ နောက်က JS မပါလည်း နာမည်ကြားယုံနဲ့ သိကုန်ကြပြီမို့လို့ မထည့်ကြတော့ပါဘူး။ လောလောဆယ် အစဉ်အလာမပျက် ထည်ပြီးသုံးနှုန်းရေးသားနေပေမယ့် နောက်ပိုင်းမှာ မထည့်တော့ပါဘူး။ Node, Express စသဖြင့် အမည်သက်သက်ပဲ ဆက်သုံးပါတော့မယ်။

API ပရောဂျက် တည်ဆောက်ဖို့အတွက် တစ်ကယ်သုံးချင်တာက Node ကို တိုက်ရိုက်သုံးချင်တာ မဟုတ်ပါဘူး။ Express Framework ကိုသုံးပြီး ဖန်တီးချင်တာပါ။ Express က Node ကိုအသုံးပြုပြီး အလုပ်လုပ်တဲ့ Framework မို့လို့သာ Node အကြောင်းကို ထည့်ပြောရတဲ့သဘောပါ။

What

JavaScript ဟာ မူလက Client-side Programming Language တစ်ခုသာ ဖြစ်ပါတယ်။ အရင်ကဆိုရင် JavaScript ကုဒ်တွေက Web Browser ထဲမှာပဲ အလုပ်လုပ်ပါတယ်။ တစ်ခြားနေရာမှာ အလုပ်မလုပ်ပါဘူး။ Web Browser တွေမှာ အစိတ်အပိုင်း (၃) ပိုင်း ပေါင်းစပ်ပါဝင်တယ်လို့ ဆိုနိုင်ပါတယ်။ Rendering Engine, JavaScript Engine နဲ့ UI တို့ဖြစ်ပါတယ်။

Rendering Engine ရဲ့ တာဝန်ကတော့ HTML, CSS တွေပေါ်မှာ အခြေခံပြီး သင့်တော်တဲ့ အသွင်အပြင် ဖော်ပြပေးဖို့ ဖြစ်ပါတယ်။ HTML Element တွေကို ဖော်ပြပေးနေတာ Rendering Engine ပါ။ CSS Style သတ်မှတ်ချက်တွေအတိုင်း ဖော်ပြပေးနေတာ Rendering Engine ပါ။ ထင်ရှားတဲ့ Rendering Engine

တွေ့ ရှိကြပါတယ်။ Apple Safari Browser က Webkit လို့ခေါ်တဲ့ Rendering Engine ကို သုံးပါတယ်။ Microsoft Internet Explorer ကတော့ Trident လို့ခေါ်တဲ့ Engine ကို သုံးပါတယ်။ MSHTML လို့လည်း ခေါ်ပါတယ်။ Mozilla Firefox က Gecko ကိုသုံးပြီး၊ Opera Browser ကတော့ Presto ကိုသုံးပါတယ်။

အဲ့ဒီထဲမှာ Webkit က အထင်ရှားဆုံးပါ။ ကြားဖူးကြပါလိမ့်မယ်။ သူ့အရင်က KHTML လို့ခေါ်တဲ့ Open Source Rendering Engine တစ်မျိုး ရှိပါတယ်။ Konqueror လို့ခေါ်တဲ့ လူသိနည်းတဲ့ Web Browser မှာ သုံးဖို့ထွင်ထားတာပါ။ အဲ့ဒီ KHTML ကိုယူပြီးတော့ Apple က Webkit ဆိုတဲ့အမည်နဲ့ Rendering Engine ကို ထပ်ဆင့်တီထွင်ထားတာပါ။ နောက်ကျတော့၊ Webkit ကို ယူသုံးပြီး Google က Chromium Browser Project ကို ထွင်လိုက်ပါတယ်။

Chromium နဲ့ Chrome ဆိုတာ နာမည်လည်းဆင်တယ်၊ နည်းပညာလည်းတူတယ်။ ဒါပေမယ့် နည်းနည်း ကွဲပါတယ်။ Chromium ကို Source Code လို့ မြင်နိုင်ပါတယ်။ သူ့ကို Compile လုပ်လိုက်တော့ Chromium Browser ဖြစ်လာပါတယ်။ အဲ့ဒီ Source Code ကိုပဲ Google က သူ့ Configuration နဲ့သူ Compile လုပ်ယူလိုက်တဲ့အခါ Google Chrome Browser ဖြစ်လာတာပါ။ Source Code ကတူတယ်။ Compile လုပ်တဲ့သူ မတူဘူးလို့ ပြောလို့ရပါတယ်။ ဒါကြောင့် Apple ရဲ့ Safari Browser, Chromium Browser နဲ့ Google Chrome တို့ဟာ သုံးထားတဲ့ Rendering Engine တူကြတယ်လို့ ပြောလို့ရပါတယ်။

နောက်တော့ Google က Webkit ကို Blink ဆိုတဲ့အမည်နဲ့ သီးခြားခွဲပြီး ထပ်ထွင်လိုက်ပါတယ်။ ဒါကြောင့် KHTML → Webkit → Blink စသဖြင့် အဆင့်ဆင့်ဖြစ်ပေါ်လာလို့ အမျိုးအစားဆင်တူတွေလို့ ဆိုနိုင်ပါ တယ်။ ကနေ့ခေတ်မှာ Webkit/Blink ဟာ အကျယ်ပြန့်ဆုံး သုံးနေကြတဲ့ Rendering Engine ပါ။ Apple Safara, Chromium Browser, Google Chrome, Android Browser, iOS Browser စသည်ဖြင့်၊ အားလုံး က Webkit/Blink ကို သုံးထားကြတာပါ။ ဒါတင်မကသေးပါဘူး၊ အခုဆိုရင် Opera Browser, Brave Browser နဲ့ Microsoft Edge Browser တို့ကလည်း Chromium ကိုသုံးထားကြတာပါ။

တစ်ကယ်တော့ လိုတိုရှင်းစာအုပ်တစ်အုပ်မှာ ဒီလိုအကြောင်းတွေကို ရှည်ရှည်ဝေးဝေး ပြောမနေချင်ပါဘူး။ လက်တွေ့လိုအပ်မှာကိုပဲ တန်းပြောလိုက်ချင်ပါတယ်။ ဒါပေမယ့် အခုလို နောက်ခံဖြစ်စဉ်လေးတွေ သိထား ရင် ပိုပြီးတော့များ စိတ်ဝင်စားစရာ ဖြစ်သွားမလားလို့ ထည့်ပြောပြနေတာပါ။

စောစောက Browser မှာ Rendering Engine, JavaScript Engine နဲ့ UI တို့ ပေါင်းစပ်ပါဝင်တယ်လို့ ပြော ခဲ့ပါတယ်။ UI ကတော့ Browser Menu တွေ၊ Toolbar တွေ၊ URL Bar, Bookmark, History, Download စတဲ့ User တွေ မြင်တွေ့ထိပြီး အသုံးပြုတဲ့ အရာတွေပါ။ JavaScript Engine ရဲ့တာဝန်ကတော့ ရှင်းပါ တယ်။ JavaScript Code တွေကို Run ပေးခြင်းပဲ ဖြစ်ပါတယ်။ Google က Chromium ကို ထွင်တဲ့အခါ Rendering Engine အနေနဲ့ Webkit ကို ယူသုံးခဲ့ပေမယ့် JavaScript Engine ကိုတော့ သူဘာသာ အသစ် ထွင်ပြီး ထည့်ခဲ့ပါတယ်။ အဲ့ဒီ Engine ကို V8 လို့ပါတယ်။ တစ်ခြား JavaScript Engine တွေနဲ့ယှဉ်ရင် သိသိသာသာ ပိုမြန်တဲ့ Engine တစ်ခုရယ်လို့ ထင်ရှားပါတယ်။

V8 JavaScript Engine ကို Chromium မှာ သုံးထားပေမယ့် သီးခြားပရောဂျက်တစ်ခုပါ။ Open Source နည်းပညာတစ်ခုပါပဲ။ အဲ့ဒီ V8 ကို သုံးပြီးတော့ Ryan Dahl လို့ခေါ်တဲ့ ဆော့ဖ်ဝဲအင်ဂျင်နီယာတစ်ဦးက Node ကို တီထွင်ခဲ့တာပါ။ အဲ့ဒီအချိန်ကစပြီးတော့ JavaScript ရဲ့အခန်းကဏ္ဍ အကြီးအကျယ်ပြောင်းလဲ သွားခဲ့ပါတော့တယ်။ အရင်က JavaScript ဆိုတာ Browser ထဲမှာပဲ Run တဲ့ နည်းပညာပါ။ အခုတော့ Browser ရဲ့ ပြင်ပမှာ Node ကိုအသုံးပြုပြီးတော့ JavaScript Code တွေကို Run လို့ ရသွားပါပြီ။ ဒီတော့ JavaScript ဟာလည်း Client-side Language ဆိုတဲ့အဆင့်ကနေကျော်ပြီး၊ ကြိုက်တဲ့နေရာမှာသုံးလို့ရတဲ့ Language တစ်ခုရယ်လို့ ဖြစ်သွားပါတော့တယ်။ ကွန်ပျူတာမှာ Node ရှိနေရင် JavaScript ကုဒ်တွေ Run လို့ရနေပြီလေ။ JavaScript နဲ့ Command Line ပရိုဂရမ်တွေ ရေးပြီး Node နဲ့ Run လို့ရမယ်။ Server-side Code တွေရေးပြီး Node နဲ့ Run လို့ရမယ်။ Desktop Solution တွေရေးပြီး Node နဲ့ Run မယ်။ စ သဖြင့် JavaScript က စွယ်စုံသုံး Language တစ်ခု ဖြစ်သွားပါတော့တယ်။

ဒါကြောင့် Node ဆိုတာဘာလဲလို့ မေးလာခဲ့ရင် မှတ်ထားပါ။ Node ဆိုတာ JavaScript Run-Time နည်း ပညာဖြစ်ပါတယ်။ Node ဆိုတာ JavaScript မဟုတ်ပါဘူး။ Node ဆိုတာ JavaScript ကုဒ်တွေကို Run ပေးနိုင်တဲ့ နည်းပညာပါ။ ပုံမှန် JavaScript ကုဒ်တွေအပြင် အသင့်သုံးလို့ရတဲ့ Standard Module တွေကို လည်း ရေးပြီး ဖြည့်တင်းပေးထားလို့ မူလ JavaScript ထက်တော့ သူကနည်းနည်းပိုပြီး စွမ်းပါတယ်။

Node ကို စထွင်ခဲ့တဲ့ Ryan Dahl က အခုတော့ Node ကို ဦးဆောင်နေသူ မဟုတ်တော့ပါဘူး။ လက်ရှိ Node ကို OpenJS Foundation လို့ခေါ်တဲ့ အဖွဲ့အစည်းက စီမံနေပါတယ်။ Ryan Dahl ကတော့ အလားတူ နောက်ထပ်နည်းပညာတစ်ခုကို မကြာခင်ကမှ ထပ်ထွင်ထားပါတယ်။ Deno လို့ခေါ်ပါတယ်။ ဒီ နည်းပညာ ဘယ်လောက်ထိ ဖြစ်မြောက်လာမလဲဆိုတာတော့ စောင့်ကြည့်ကြရဦးမှာပါ။

Why

Node ရဲ့ ထူးခြားချက်ကတော့ Non-blocking I/O လို့ခေါ်တဲ့ နည်းစနစ်ကို အသုံးပြုခြင်းပါ။ JavaScript ဟာ Interpreted Language တစ်ခုဖြစ်လို့ Language ချည်းသက်သက်အရဆိုရင် သိပ်မြန်လှတဲ့ Language တော့ မဟုတ်ပါဘူး။ တစ်ခြားသူ့ထက်ပိုမြန်တဲ့ Language တွေရှိပါတယ်။ ဒါပေမယ့် Node ရဲ့ Non-blocking I/O သဘောသဘာဝကြောင့် Node နဲ့ရေးထားတဲ့ ပရိုဂရမ်တွေဟာ သာမာန်ထက် ပိုမြန်လေ့ရှိကြပါတယ်။

ကွန်ပျူတာရဲ့ Processor ဟာ တစ်စက္ကန့်မှာ Instruction ပေါင်း သန်းနဲ့ချီပြီး အလုပ်လုပ်ပေးနိုင်ပါတယ်။ ဒါပေမယ့် Hard Disk ပေါ်က အချက်အလက်တွေ Read/Write လုပ်ယူတာလို အလုပ်မျိုးက ဒီလောက် မမြန်နိုင်ပါဘူး။ အင်တာနက် အဆက်အသွယ်ကိုသုံးပြီး အချက်အလက်တွေ ယူရမယ်ဆိုရင်လည်း၊ ဘယ်လောက်မြန်တဲ့ အင်တာနက်ကြီး ဖြစ်နေပါစေ၊ ဒီအသွားအပြန်က Processor ရဲ့ စွမ်းဆောင်ရည်ကို ဘယ်လိုမှ အမှီလိုက်နိုင်မှာ မဟုတ်ပါဘူး။ အတူတူပါပဲ၊ ကင်မရာ၊ Fingerprint Sensor စသဖြင့် Input / Output Device တွေဟာ၊ Processor ရဲ့ အလုပ်လုပ်နိုင်စွမ်းနဲ့ နှိုင်းယှဉ်ကြည့်ရင် တော်တော် နှေးကြပါတယ်။ ရိုးရိုးပရိုဂရမ်တွေမှာ Process က မြန်ချင်ပေမယ့် မြန်လို့မရဘဲ I/O ကို ပြန်စောင့်နေရတာတွေ ရှိကြပါတယ်။ Node ကတော့ Asynchronous ရေးဟန်နဲ့အတူ Process က I/O ကို စောင့်စရာအောင် တီထွင်ထားပါတယ်။ ဒီသဘောပေါ်လွင်စေဖို့ ဥပမာကုဒ်ရိုးရိုးလေးတစ်ခုလောက် ရေးပြပါမယ်။

```
const fs = require("fs");

console.log("Some processes...");

fs.readFile("data.txt", "utf-8", function(err, data) {
    console.log(data);
});

console.log("Some more processes...");
```

ဒီကုဒ်မှာ ရေးထားတဲ့ အစီအစဉ်အရ Some processes ဆိုတဲ့စာတစ်ကြောင်းကို အရင်ရိုက်ထုတ်မယ်။ ပြီးရင် `data.txt` ကို ဖတ်ပြီး အထဲက Content ကို ရိုက်ထုတ်မယ်။ ပြီးရင် Some more processes ဆိုတဲ့ စာတစ်ကြောင်းကို ဆက်ပြီးရိုက်ထုတ်မယ်။ ဒီလိုရေးထားတာပါ။ ဒါပေမယ့် Node ရဲ့ Non-blocking I/O သဘောသဘာဝက `data.txt` ကို ဖတ်လို့ပြီးအောင် မစောင့်ဘဲ လုပ်စရာရှိတာ ဆက်လုပ်သွားမှာမို့လို့

Run ကြည့်ရင် ရလဒ်က ဒီလိုရမှာပါ။

```
Some processes…
Some more processes…
>> data entries…
```

ဖိုင်ကိုဖတ်ယူတဲ့ `fs.readFile()` အတွက် Callback Function ကိုပေးပြီး ရေးထားတဲ့အတွက် ဖိုင်ကို ဖတ်ယူတဲ့ I/O ကို စောင့်စရာမလိုဘဲ ကျန်တဲ့ Process တွေက ဆက်အလုပ်လုပ်သွားတာပါ။ ဖိုင်ကိုဖတ်လို့ ပြီးတော့မှသာ ပေးလိုက်တဲ့ Callback Function က အလုပ်လုပ်ခြင်း ဖြစ်ပါတယ်။ Non-blocking I/O ဆိုတာ ဒါမျိုးကိုဆိုလိုတာပါ။ ဒါကြောင့် တစ်ချို့ နည်းပညာလုပ်ငန်းကြီးတွေကအစ စွမ်းဆောင်ရည်မြင့် Service တွေ ဖန်တီးဖို့အတွက် Node ကို ရွေးချယ် အသုံးပြုနေကြတာပါ။

Install Node

Node Installer ကို nodejs.org ကနေ Download ရယူနိုင်ပါတယ်။ သူကတော့ သိပ်ဆန်းဆန်းပြားပြား တွေမလိုဘဲ အလွယ်တစ်ကူ Install လုပ်ပြီး သုံးလို့ရလေ့ရှိပါတယ်။ အခုဒီစာကိုရေးနေချိန်မှာ နောက်ဆုံး ထွက်ရှိထားတဲ့ Version ကတော့ 14.11.0 ပါ။ Feature သစ်တွေစမ်းချင်ရင် နောက်ဆုံး Version ကို သုံးရပြီး၊ လက်တွေ့သုံးဖို့ ရည်ရွယ်ရင် LTS (Long-Term Support) Version ကို သုံးရပါတယ်။ အခုဒီစာအုပ်မှာ ဖော်ပြမယ့်ကုဒ်ကတော့ နမူနာသက်သက်မို့ ဘာကိုပဲရွေးရွေး ကိစ္စမရှိပါဘူး။

Install လုပ်လိုက်ရင် ပါဝင်လာမယ့် နည်းပညာက (၂) ခုပါ။ **node** နဲ့ **npm** ဖြစ်ပါတယ်။ JavaScript ကုဒ်တွေကို `node` နဲ့ Run ရမှာဖြစ်ပြီး `npm` ကတော့ Package Manager ပါ။ `npm` ကိုသုံးပြီး Express အပါအဝင် ကိုယ်သုံးချင်တဲ့ Package တွေကို ရယူအသုံးပြုနိုင်ပါတယ်။

Ubuntu Linux လို Operation System သုံးနေတဲ့ သူတွေကတော့ Website မှာသွားပြီး Download လုပ်နေစရာမလိုပါဘူး။ `apt` နဲ့ အခုလို Install လုပ်လို့ရနိုင်ပါတယ်။

```
sudo install nodejs npm
```

`npm` ကိုပါသီးခြားတွဲပြီး Install လုပ်ပေးရတယ်ဆိုတာကိုတော့ သတိပြုပါ။

Install လုပ်ပြီးပြီဆိုရင် Command Prompt ဖွင့်ပြီး အခုလိုစမ်းကြည့်နိုင်ပါတယ်။

```
node -v
v12.16.1

npm -v
6.14.8
```

`-v` Option က Version နံပါတ်ကို ထုတ်ကြည့်တာပါ။ ဒါကြောင့် အပေါ်ကနမူနာမှာလို ကိုယ် Install လုပ်ထားတဲ့ `node` Version နဲ့ `npm` Version ကို တွေ့မြင်ရပြီဆိုရင် အသုံးပြုဖို့ Ready ဖြစ်နေပါပြီ။

Node Modules

Node မှာ တစ်ခါတည်း အသင့်သုံးလို့ရတဲ့ Module တွေ ပါပါတယ်။ ဖိုင်တွေ၊ ဖိုဒါတွေ စီမံနိုင်တဲ့ File System Module, Web Server တွေ Service တွေ ဖန်တီးလို့ရတဲ့ HTTP Module, Network Socket ပရိုဂရမ်တွေ ဖန်တီးလို့ရတဲ့ Net Module စသဖြင့် ပါသလို Crypto, OS, Path, Zlib, Stream, URL, Timers စသဖြင့် အထွေထွေသုံး Module တွေ အများကြီး ပါပါတယ်။ အပေါ်မှာ Non-blocking I/O အကြောင်း ပြောတုံးက ပေးခဲ့တဲ့ ကုဒ်နမူနာက File System Module ကို သုံးထားတာပါ။ ဒီ Module တွေအကြောင်းကိုတော့ စုံအောင်ထည့်မပြောနိုင်ပါဘူး။ အသုံးဝင်တဲ့ Build-in Module တွေရှိတယ်၊ လိုအပ်ရင်သုံးလို့ရတယ် ဆိုတာကိုသာ မှတ်ထားပေးပါ။ JavaScript ရေးတတ်တယ်ဆိုရင် ဒီ Module တွေ ယူသုံးရတာက မခက်လှပါဘူး။ Node ရဲ့ Documentation မှာ လေ့လာပြီး သုံးသွားရင် အဆင်ပြေမှာပါ။

– https://nodejs.org/en/docs/

ကိုယ်တိုင် Module တွေ ရေးသားဖန်တီးပုံကိုတော့ ထည့်ပြောပါမယ်။ Module ရေးနည်းမှာ နှစ်မျိုးရှိပါတယ်။ CommonJS ခေါ် Node က မူလအစကတည်းက အသုံးပြုခဲ့တဲ့ ရေးနည်းနဲ့၊ နောက်ပိုင်း ES6 လို့ခေါ်တဲ့ JavaScript နည်းပညာသစ်နဲ့တူ ပါဝင်လာတဲ့ Module ရေးနည်းတို့ဖြစ်ပါတယ်။ ရေးနည်းနှစ်ခုက သဘောသဘာဝ ဆင်ပေမယ့် Syntax မတူလို့ ရောတတ်ပါတယ်။ ဒီအကြောင်းကို **React လို တို ရှင်း** စာအုပ်မှာလည်းဖော်ပြခဲ့ဖူးပါတယ်။ ES6 အကြောင်းအကျဉ်းချုပ်ကို ဒီလိပ်စာမှာလည်း ဖတ်လို့ရပါတယ်။

– https://gist.github.com/eimg/cf2ac1f2731f77cde166bb764eef7634

ES6 အကြောင်း မသိသေးရင် မဖြစ်မနေ ကြိုဖတ်ထားဖို့ လိုအပ်ပါတယ်။ နောက်ပိုင်းကုဒ်နမူနာတွေမှာ သိပြီးသားလို့ သဘောထားပြီး ဆက်ပြောသွားမှာမို့လို့ပါ။ ဥပမာအနေနဲ့ `Math` အမည်ရ Module တစ်ခုရေးမယ် ဆိုကြပါစို့။ အခုလို နှစ်မျိုး ရေးနိုင်ပါတယ်။

CommonJS Module

```
// math.js
function add(a, b) {
    return a + b;
}

module.exports = add;
```

ES6 Module

```
// math.js
function add(a, b) {
    return a + b;
}

export default add;
```

CommonJS က `module.exports` ကိုသုံးပြီး ES6 Module ကတော့ `export` ကို သုံးပါတယ်။ ဒီလို လုပ်ဆောင်ချက်တစ်ခုကို ပေးသုံးတဲ့နည်းကို Default Export လို့ခေါ်ပါတယ်။ ပြန်ယူသုံးတဲ့အခါ အခုလို ပြန်ယူသုံးနိုင်ပါတယ်။

CommonJS Module

```
const add = require("./math");

console.log( add(1, 2) );       // => 3
```

ES6 Module

```
import add from "./math";

console.log( add(1, 2) );       // => 3
```

CommonJS က `require()` ကိုသုံးပြီး ES6 Module က `import from` ကို သုံးပါတယ်။ Module

ဖိုင်ရဲ့ Path လမ်းကြောင်းကိုသုံးပြီး လိုချင်တဲ့ Module ကိုရယူလိုက်တာပါ။ အကယ်၍ လုပ်ဆောင်ချက်တွေ တစ်ခုထက်ပိုရှိလို့ ပေးချင်တယ်ဆိုရင် အခုလိုပေးလို့ရပါတယ်။

CommonJS Module

```
const PI = 3.14;

function add(a, b) {
    return a + b;
}

module.exports = { PI, add };
```

ES6 Module

```
const PI = 3.14;

function add(a, b) {
    return a + b;
}

export { PI, add };
```

ရေးစရာရှိတာ အရင်ရေးပြီး နောက်ဆုံးမှ ပေးချင်တဲ့ လုပ်ဆောင်ချက်တွေကို `module.exports` တို့ `export` တို့နဲ့ ပေးလိုက်တာပါ။ အဲ့ဒီလိုရေးမယ့်စား နောက်တစ်မျိုးရေးလို့လည်း ရပါသေးတယ်။

CommonJS Module

```
exports.PI = 3.14;

exports.add = function (a, b) {
    return a + b;
}
```

ES6 Module

```
export const PI = 3.14;

export function add(a, b) {
    return a + b;
}
```

စရေးကတည်းက ပေးချင်တဲ့ လုပ်ဆောင်ချက်တွေကို တစ်ခါတည်း Export လုပ်သွားလိုက်တဲ့ သဘောပါ။ ဒီလုပ်ဆောင်ချက်တွေကို လိုအပ်တဲ့အခါ အခုလို ပြန်ယူသုံးနိုင်ပါတယ်။

CommonJS Module

```
const math = require("./math");

console.log( math.PI );              // => 3.14
console.log( math.add(1, 2) );       // => 3
```

ES6 Module

```
import { PI, add } from "./math";

console.log( PI );                   // => 3.14
console.log( add(1, 2) );            // => 3
```

ရေးနည်းက တူသလိုလို၊ မတူသလိုလို မို့လို့ အရမ်းရောပါတယ်။ သတိထားရပါတယ်။

အခုဒီစာရေးနေချိန်မှာ Node ရဲ့ Default က CommonJS ရေးထုံးဖြစ်ပါတယ်။ ES6 ရေးထုံးနဲ့ ရေးချင်ရင် ဖိုင် Extension ကို `.mjs` လို့ ပေးရပါတယ်။ နှစ်မျိုးရောသုံးမယ်ဆိုရင် ရနိုင်တဲ့နည်းအချို့ရှိနေပေမယ့် အဆင်မပြေပါဘူး။ CommonJS သို့မဟုတ် ES6 Module နှစ်မျိုးထဲက တစ်မျိုးကို ပြတ်ပြတ်သားသား ရွေးချယ် အသုံးပြုရမှာပါ။ ဒါက လက်ရှိအခြေအနေပါ။ နောက်ပိုင်းမှာတော့ အပြောင်းအလဲတွေ ရှိကောင်း ရှိနိုင်ပါတယ်။

Module တွေက ပုံစံ (၃) မျိုးဖြစ်နိုင်တယ်လို့ ဆိုနိုင်ပါတယ်။ တစ်မျိုးက Node ရဲ့ Build-in Module တွေပါ။ နောက်တစ်မျိုးက အထက်ကနမူနာမှာ ပြထားသလို ကိုယ်ပိုင်ဖန်တီးထားတဲ့ Module တွေပါ။ နောက်တစ်မျိုးကတော့ `npm` ရဲ့ အကူအညီနဲ့ ရယူအသုံးပြုတဲ့ Module တွေပဲ ဖြစ်ပါတယ်။ Express အပါအဝင် JavaScript Package ပေါင်းမြောက်များစွာ ရှိနေပါတယ်။ များလွန်းလို့တောင် ခက်နေပါသေးတယ်။ ကိုယ့် ဘာသာ ရေးစရာမလိုသလောက်ပါပဲ။ လုပ်ဆောင်ချက်တစ်ခု လိုအပ်ရင် ရှာကြည့်လိုက်၊ ယူသုံးလို့ရတဲ့ Third-party Module ရှိဖို့ ကျိမ်းသေတယ်လို့ ပြောရမလောက်ပါပဲ။ အဲ့ဒီလို ယူသုံးလို့ရတဲ့ Module တွေကို `npm` ရဲ့ အကူအညီနဲ့ ယူရတာမို့လို့ `npm` အကြောင်းကို ဆက်ပြောပါမယ်။

NPM

npm နဲ့ လုပ်မယ့်အလုပ် (၃) ခုရှိတယ်လို့ မှတ်နိုင်ပါတယ်။ တစ်ခုက ပရောဂျက် တည်ဆောက်မှာပါ။ နောက်တစ်ခုကတော့ လိုအပ်တဲ့ Package တွေကို ရယူမှာဖြစ်ပြီး၊ နောက်ဆုံးတစ်ခုကတော့ Script တွေ Run မှာ ဖြစ်ပါတယ်။

npm ကို သုံးပြီး ပရောဂျက်တည်ဆောက်ဖို့အတွက် npm init ကို သုံးနိုင်ပါတယ်။ သူက ပရောဂျက် အမည်၊ Version နံပါတ်၊ ပရောဂျက် Description၊ လိုင်စင်၊ စတဲ့အချက်အလက်တွေကို လာမေးပါလိမ့်မယ်။ တစ်ခုချင်း ဖြေပေးသွားလိုက်ရင် နောက်ဆုံးမှာ ကိုယ်ပေးလိုက်တဲ့ အချက်အလက်တွေ ပါဝင်တဲ့ package.json ဆိုတဲ့ ဖိုင်တစ်ခုထွက်လာပါလိမ့်မယ်။ ဖိုဒါတစ်ခုမှာ package.json ဖိုင် ပါဝင်သွားတာနဲ့ NPM Project တစ်ခု ဖြစ်သွားပါပြီ။ တစ်ကယ်တော့ အမေးအဖြေတွေ တစ်ကြောင်းချင်း လုပ်ရတာ ကြာပါတယ်။ npm init -y ဆိုပြီး နောက်ဆုံးက -y Option လေးထည့်ပေးလိုက်ရင် ဘာမှ လာမမေးတော့ဘဲ package.json ကို Default Value တွေနဲ့တည်ဆောက်ပေးသွားမှာပါ။ နောက်မှ ပြင်စရာရှိရင် အဲ့ဒီ package.json ကိုဖွင့်ပြင်တာက ပိုအဆင်ပြေပါတယ်။

ဖိုဒါတစ်ခုကို နှစ်သက်ရာအမည်နဲ့ ကိုယ့်ဘာသာ ဆောက်လိုက်ပါ။ အဲ့ဒီဖိုဒါထဲမှာ အခုလို Run ပေးလိုက်ရင် ရပါပြီ။ပရောဂျက်ဖိုဒါ ဖြစ်သွားပါပြီ။

```
npm init -y

Wrote to /path/to/project/package.json:

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

ပရောဂျက်ဖိုဒါထဲမှာပဲ npm ကိုသုံးပြီးတော့ လိုချင်တဲ့ Third-party Package တွေကို ရယူလို့ရပါတယ်။

ဥပမာ ဒီလိုပါ -

```
npm install express
```

ဒါဆိုရင် `express` ဆိုတဲ့ Package ကို `npm` က Download လုပ်ယူပေးယုံသာမက၊ `express` အလုပ်လုပ်နိုင်ဖို့အတွက် ဆက်စပ်လိုအပ်နေတဲ့ Dependency Package တွေကိုပါ အလိုအလျောက် ရယူပေးသွားမှာပါ။ ရရှိလာတဲ့ Package တွေကို `node_modules` ဆိုတဲ့ ဖိုဒါတစ်ခုထဲမှာ အကုန်ထည့်ပေးသွားမှာဖြစ်ပါတယ်။ အသုံးလိုတဲ့အခါ အခုလို အလွယ်တစ်ကူ ချိတ်ဆက်အသုံးပြုလို့ရပါတယ်။

```
const express = require("express");
```

Package အမည်ဖြစ်တဲ့ `express` လို့ပဲပြောလိုက်ဖို့လိုပါတယ်။ Package ရဲ့တည်နေရာကို ပြောပြစရာမလိုပါဘူး။ `npm` က Package တွေ `node_modules` ဆိုတဲ့ ဖိုဒါထဲမှာ ရှိမှန်းသိထားပြီးသားပါ။ `npm install` အစား အတိုကောက် ဒီလိုရေးလို့လည်း ရပါတယ်။ `i` တစ်လုံးတည်းပေးလိုက်တာပါ။

```
npm i express
```

ဒါဆိုရင်လည်း `express` Package ကို ရယူလိုက်တာပါပဲ။ ဒီလိုရယူပြီးနောက် `package.json` ဖိုင်ကို ဖွင့်ကြည့်လိုက်ပါ။ အခုလိုဖြစ်နေနိုင်ပါတယ်။

```
{
  "name": "project",
  …
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  …
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

`dependencies` ဆိုတဲ့အပိုင်းတစ်ပိုင်း တိုးလာပြီး အထဲမှာ `express` ပါဝင်နေတာကို တွေ့ရပါလိမ့်မယ်။ အဲ့ဒါကတော့ လက်ရှိပရောဂျက်ဟာ `express` ရှိမှ အလုပ်လုပ်တယ်လို့ သူ့အလိုလို သတ်မှတ်လိုက်တာပါပဲ။ တစ်ခြား Package တွေထပ်ပြီးတော့ `install` လုပ်ရင်လည်း အဲ့ဒီလိုထပ်တိုးပြီးတော့ ပါဝင်သွားဦးမှာပါ။

တစ်ချို့ Package တွေက လိုတော့လိုအပ်တယ်၊ ဒါပေမယ့် အဲ့ဒါရှိမှ အလုပ်လုပ်တာ မဟုတ်ဘူး၊ ကုဒ်တွေရေးနေစဉ်မှာပဲ လိုအပ်တာဆိုတာမျိုး ရှိတတ်ပါတယ်။ ဥပမာ - `eslint` လို နည်းပညာမျိုးပါ။ သူက ရေးထားတဲ့ ကုဒ်ထဲမှာရှိနေတဲ့ အမှားတွေကို ရှာပေးနိုင်ပါတယ်။ ဒါကြောင့် သူရှိမှ အလုပ်လုပ်တာမျိုးတော့ မဟုတ်ပါဘူး။ ကုဒ်တွေရေးနေစဉ် Development Time မှာပဲ လိုအပ်တဲ့သဘောပါ။ ဒီလို Package မျိုးကို Development Dependency လို့ခေါ်ပါတယ်။ `install` လုပ်တဲ့အခါ `--save-dev` ဆိုတဲ့ Option နဲ့ တွဲပြီးတော့ `install` လုပ်သင့်ပါတယ်။ အတိုကောက် `-D` လို့ပြောရင်လည်း ရပါတယ်။ ဒီလိုပါ။

```
npm i -D eslint
```

ဒီလို `install` လုပ်ပြီးတဲ့အခါ `package.json` ကို ပြန်လေ့လာကြည့်လိုက်ပါ။ ဒီလိုပုံစံမျိုး ဖြစ်နိုင်ပါတယ်။

```
{
  "name": "project",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.1"
  },
  "devDependencies": {
    "eslint": "^7.9.0"
  }
}
```

`devDependencies` ဆိုတဲ့ အပိုင်းတစ်ခု ထပ်တိုးသွားတာပါ။ ဒါကြောင့် ဒီဖိုင်ကို ကြည့်လိုက်ယုံနဲ့ ဒီပရောဂျက်မှာ ဘာတွေလိုလဲဆိုတာကို သိရနိုင်ပါတယ်။ စောစောက `install` လုပ်လိုက်တဲ့ Package တွေကို `node_modules` ဖိုဒါထဲမှာ သိမ်းသွားတယ်လို့ ပြောခဲ့ပါတယ်။ ကိုယ့် ပရောဂျက်ကို သူများကို ပေးတဲ့အခါ အဲ့ဒီ `node_modules` ဖိုဒါ ထည့်ပေးစရာမလိုပါဘူး။ အလားတူပဲ သူများ Package တွေကို ယူတဲ့အခါမှာလည်း `node_modules` ဖိုဒါမပါလို့ ဘာမှမဖြစ်ပါဘူး။ `package.json` ပါဖို့ပဲလိုပါတယ်။ `npm` ကို လိုတာတွေအကုန် `install` လုပ်လိုက်ပါလို့ အခုလို ပြောလို့ရပါတယ်။

```
npm i
```

ဒါပါပဲ။ `install` အတွက် နောက်က Package အမည် မပေးတော့တာပါ။ ဒါဆိုရင် `npm` က `package.json` ကို ကြည့်ပြီးတော့ လိုတာတွေအကုန် `install` လုပ်ပေးသွားမှာ ဖြစ်ပါတယ်။

NPM ကို သုံးမယ့်ကိစ္စ (၃) ခုမှာ (၂) ခုတော့ ရသွားပါပြီ။ တစ်ခုက ပရောဂျက်ဆောက်တာပါ။ နောက်တစ်ခုက လိုတဲ့ Package တွေ ရယူတာပါ။ ကျန်နေတဲ့ နောက်ဆုံးတစ်ခုကတော့ Script တွေ Run တာဖြစ်ပါတယ်။ `package.json` ထဲမှာ `scripts` ဆိုတဲ့ အပိုင်းတစ်ပိုင်းပါပါတယ်။ အဲ့ဒီ `scripts` မှာအခုလို နမူနာ `script` လေးတစ်ခုလောက် ရေးပြီး ထည့်လိုက်ပါ။

```
{
  "name": "project",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",

  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "fun": "echo \"NPM scripts are fun\""
  },
  …
  "dependencies": {
    "express": "^4.17.1"
  },
  "devDependencies": {
    "eslint": "^7.9.0"
  }
}
```

`scripts` ထဲမှာ `test` က နဂိုကတည်းကပါတာပါ။ `fun` လို့ခေါ်တဲ့ Script တစ်ခု ထပ်ရေးပြီး ထည့်ပေးလိုက်တာပါ။ ဘာမှအထူးအဆန်းတော့ မဟုတ်ပါဘူး စာလေးတစ်ကြောင်း ရိုက်ထုတ်ပေးတဲ့ Script ပါ။ အဲ့ဒီ Script ကို အခုလို `npm` နဲ့ Run ခိုင်းလို့ရပါတယ်။

```
npm run fun

> app@1.0.0 fun /home/eimg/Desktop/app
> echo "NPM scripts are fun"

NPM scripts are fun
```

ဒီနည်းကိုသုံးပြီး ပရောဂျက်နဲ့ Source Code တွေကို စီမံတဲ့ Script တွေရေးထားပြီး လိုတဲ့အချိန် Run လို့ရခြင်း ဖြစ်ပါတယ်။ Development Process Management အတွက် အရေးပါတဲ့ လုပ်ဆောင်ချက်ပါ။ ကုဒ်တွေရေးပြီးတဲ့အခါ Jest Test Library သုံးပြီး Unit Test တွေ Run ပေးဖို့လိုနိုင်ပါတယ်။ ESLint လိုနည်းပညာမျိုးနဲ့ ကုဒ်အမှားတွေ ရှားခိုင်းလို့ ရနိုင်ပါတယ်။ Babel လိုနည်းပညာမျိုး နဲ့ Compile လုပ်ဖို့ လိုတာတွေ Compile လုပ်ဖို့လိုနိုင်ပါတယ်၊ ဥပမာ React ရဲ့ JSX နဲ့ ရေးထားတဲ့ကုဒ်ကို JavaScript ဖြစ်အောင် ပြောင်းတာမျိုးပါ။ Webpack လိုနည်းပညာမျိုးနဲ့ ပရောဂျက်ကို Development Server နဲ့ Run ဖို့လိုနိုင်ပါတယ်။ ဒါမျိုးတွေကို တစ်ခုချင်း Manual လုပ်မယ့်အစား Script တွေရေးထားလိုက်မယ်ဆိုရင် NPM Script နဲ့ လိုအပ်ချိန်မှာ တစ်ချက်တည်းနဲ့ Run လို့ရမှာဖြစ်ပါတယ်။ ဒီအကြောင်းအရာတွေကတော့ ကျွန်တော်တို့အဓိကထား လေ့လာချင်တဲ့အကြောင်းအရာနဲ့ Out of scope ဖြစ်နေလို့ ထည့်မပြောပါဘူး။ NPM နဲ့ Script တွေ Run လို့ရတယ်ဆိုတာကိုသာ မှတ်ထားလိုက်ပါ။

နောက်ထပ်တစ်ခု မှတ်သင့်ပါသေးတယ်။ `npm` နဲ့ အခုနမူနာ `install` လုပ်ပြတဲ့ Package တွေကို Local Package လို့ခေါ်ပါတယ်။ လက်ရှိပရောဂျက်နဲ့ပဲ သက်ဆိုင်ပြီး လက်ရှိပရောဂျက် ထဲကနေပဲ သုံးခွင့်ရှိတဲ့ Package တွေပါ။ လိုအပ်ရင် Global Package ခေါ် System Wide ကြိုက်တဲ့နေရာကနေ ချိတ်သုံးလို့ရအောင်လည်း install လုပ်လို့ရပါတယ်။ `--global` သို့မဟုတ် `-g` Option နဲ့တွဲပြီး `install` လုပ်ရပါတယ်။ ဒီလိုပါ –

```
npm i -g eslint
```

ဒီတစ်ခါတော့ `eslint` ကို Global Package အနေနဲ့ Install လုပ်လိုက်တာပါ (Ubuntu Linux လို့ စနစ်မျိုးမှာ ရှေ့က `sudo` ထည့်ပေးဖို့ လိုနိုင်ပါတယ်)။ ဒါကြောင့် ဒီ Package ကို System Wide ကြိုက်တဲ့နေရာကနေ ခေါ်သုံးလို့ရသွားပါပြီ။

`npx` ဆိုတာလည်း ရှိပါသေးတယ်။ တစ်ချို့ Install လုပ်ထားတဲ့ Package တွေက Module အနေနဲ့ ခေါ်သုံးဖို့ မဟုတ်ပါဘူး။ လိုအပ်တဲ့အခါ Run နိုင်ဖို့ Install လုပ်ထားတာပါ။ အခုနမူနာပေးနေတဲ့ `eslint` ဆိုရင်လည်း ဒီသဘောပါပဲ။ ဒီ Package ကိုခေါ်သုံးပြီး ကုဒ်တွေရေးဖို့ထက်စာရင် ဒီ Package ကို Run ပြီးတော့ ကုဒ်တွေကို စစ်ဖို့ဖြစ်ပါတယ်။ Global Package အနေနဲ့ Install လုပ်ထားရင် အခုလို ရိုးရိုး Command တစ်ခုကဲ့သို့ Run လို့ရပါတယ်။ ကြိုက်တဲ့နေရာကနေ Run လို့ရပါတယ်။

```
eslint --init
eslint math.js
```

`eslint --init` နဲ့ Configuration ဖိုင်တည်ဆောက်ပြီး နောက်တစ်ဆင့်မှာ `math.js` ထဲမှာရှိတဲ့အမှားတွေကို စစ်ခိုင်းလိုက်တာပါ။ အကယ်၍ Local Package အနေနဲ့ Install လုပ်ထားတာ ဆိုရင်တော့ `npx` ကိုသုံးပြီး အခုလို Run ပေးရပါတယ်။

```
npx eslint --init
npx eslint math.js
```

ဒီလောက်ဆိုရင်တော့ Node တို့ NPM တို့နဲ့ပက်သက်ပြီး သိသင့်တာလေးတွေ စုံသလောက်ရှိသွားပါပြီ။

အခုပြောခဲ့တဲ့ထဲမှာ ရေးမယ်လို့ ရည်ရွယ်ထားတဲ့ ပရောဂျက်အကြောင်းတော့ မပါသေးပါဘူး။ နောက်တစ်ခန်းကျတော့မှ Express ကိုလေ့လာရင်း ဆက်ရေးသွားကြမှာမို့လို့ပါ။

အခန်း (၈) – Express

Express ဟာ JavaScript Server-side Framework တွေထဲမှာ လူသုံးအများဆုံးဖြစ်ပြီးတော့ Service နဲ့ API တွေဖန်တီးဖို့အတွက် အဓိကအသုံးပြုကြပါတယ်။ ရိုးရှင်းပြီး အသုံးပြုရလည်း လွယ်သလို စွမ်းဆောင်ရည်မြင့်ပြီး မြန်တဲ့အတွက် လက်တွေ့အသုံးချ ပရောဂျက်ကြီးတွေကထိ အားကိုးအားထား ပြုကြရတဲ့ Framework ပါ။

ပြီးခဲ့တဲ့အခန်းမှာ NPM အကြောင်းပြောရင်း ပရောဂျက်တည်ဆောက်ပုံနဲ့ Express ကို ရယူပုံဖော်ပြခဲ့ပါတယ်။ ပြန်ကြည့်နေရတာမျိုးမဖြစ်အောင် နောက်တစ်ခေါက် ပြန်ပြောလိုက်ပါမယ်။ ပထမဦးဆုံး မိမိနှစ်သက်ရာအမည်နဲ့ ဖိုဒါတစ်ခုဆောက်လိုက်ပါ။ ပြီးတဲ့အခါ အဲ့ဒီဖိုဒါထဲမှာ အခုလို ပရောဂျက်တစ်ခု ဖန်တီးပြီး Express ကိုထည့်သွင်းပေးလိုက်ပါ။

```
npm init -y
npm i express
```

ဒါဆိုရင် Express ကို အသုံးပြုပြီး ကုဒ်တွေ စရေးလို့ရပါပြီ။ ရေးချင်တဲ့ ပရောဂျက်ကုဒ်တွေ မရေးခင် Express ရဲ့ သဘောသဘာဝတစ်ချို့ကို အရင်ပြောပြချင်ပါတယ်။ ပထမဆုံးအနေနဲ့ Express ကိုသုံးပြီး API URL သတ်မှတ်ပုံနဲ့ Server Run ပုံ Run နည်းကို အရင်ပြောပြပါမယ်။ ပရောဂျက်ဖိုဒါထဲမှာ `index.js` အမည်နဲ့ ဖိုင်တစ်ခုဆောက်ပြီး အခုလို ရေးသားပေးပါ။

```
const express = require("express");
const app = express();

app.get("/api/people", function(req, res) {
    const data = [
        { name: "Bobo", age: 22 },
        { name: "Nini", age: 23 },
    ];

    return res.status(200).json(data);
});

app.listen(8000, function() {
    console.log("Server running at port 8000...");
});
```

ဒီကုဒ်မှာ ပထမဆုံး `express` ကို Import လုပ်ယူပါတယ်။ ပြီးတဲ့အခါသူ့ကို Run ပြီးတော့ `app` Object တစ်ခုတည်ဆောက်ပါတယ်။ နမူနာမှာ `app` ရဲ့ Method (၂) ခုကို သုံးထားပါတယ်။ `get()` နဲ့ `listen()` တို့ဖြစ်ပါတယ်။

`get()` ဟာ Route Method တစ်ခုဖြစ်ပြီး URL လိပ်စာတွေ သတ်မှတ်ဖို့ သုံးပါတယ်။ လက်ခံလိုတဲ့ Request Method ပေါ်မူတည်ပြီး `get()`, `post()`, `put()`, `patch()`, `delete()` စသဖြင့် Route Method တွေ အစုံရှိပါတယ်။ ကိုယ်လိုအပ်တဲ့ Method ကို အသုံးပြုနိုင်ပါတယ်။

Route Method တွေဟာ URL နဲ့ Callback Function တို့ကို Parameter များအနေနဲ့ လက်ခံပါတယ်။ URL သတ်မှတ်ပုံကတော့ ဟိုး RESTful API မှာ ရှင်းပြခဲ့ပြီးဖြစ်တဲ့ ဖွဲ့စည်းပုံအတိုင်း သတ်မှတ်နိုင်ပါတယ်။ Callback Function ကတော့ သတ်မှတ်ထားတဲ့ URL ကိုအသုံးပြုပြီး Request ဝင်ရောက်လာတဲ့အခါ အလုပ်လုပ်မယ့် Function ဖြစ်ပါတယ်။ သူ့မှာ Request နဲ့ Response Object တွေကို လက်ခံအသုံးပြုလို့ ရပါတယ်။ နမူနာမှာ Request ကို `req` ဆိုတဲ့ Variable နဲ့လက်ခံပြီး Response ကို `res` ဆိုတဲ့ Variable နဲ့ လက်ခံယူထားတာကို တွေ့နိုင်ပါတယ်။ `req` မှာ Request နဲ့ပက်သက်တဲ့ Header တွေ Body တွေ အကုန်ရှိပါတယ်။ တစ်ခြားအသုံးဝင်နိုင်တဲ့ Cookie တို့ Host တို့ IP Address တို့လည်း ရှိပါတယ်။ အလားတူပဲ `res` ကို သုံးပြီးတော့ လိုအပ်တဲ့ Response Header, Status Code နဲ့ Body တွေ သတ်မှတ်လို့ရပါတယ်။

ပေးထားတဲ့နမူနာအရ `get()` Method ကိုသုံးပြီးရေးထားလို့ Request Method က `GET` ဖြစ်ရပါမယ်။ ပါ။ URL က `/api/people` အတိအကျဖြစ်ရပါမယ်။ Request Header နဲ့ Body ကတော့ ပေးပို့သူ ကြိုက်သလိုပို့လို့ ရပါတယ်။ သူပို့သမျှ `req` ထဲမှာ အကုန်ရှိနေမှာပါ။

နမူနာကုဒ်ရဲ့ အလုပ်လုပ်ပုံကတော့ ရိုးရိုးလေးပါ။ `data` လို့ခေါ်တဲ့ Sample JSON Array တစ်ခုရှိပါတယ်။ `res.status()` ကိုသုံးပြီး Status Code သတ်မှတ်ပါတယ်။ `json()` ကတော့ အလုပ်နှစ်ခု လုပ်ပေးပါ တယ်။ Response Header မှာ `Content-Typ: application/json` လို့သတ်မှတ်ပြီး ပေးလိုက် တဲ့ JSON `data` ကို Response Body အနေနဲ့ ပြန်ပို့ပေးမှာပါ။ တစ်ခြား `send()`, `sendFile()` စ သဖြင့် ပြန်ပို့ချင်တဲ့ Content အမျိုးအစားပေါ်မူတည်ပြီး ပို့လို့ရတဲ့ လုပ်ဆောင်ချက်တွေ ရှိပါသေးတယ်။ ဒါ ပေမယ့် ကျွန်တော်တို့ကတော့ `json()` ကိုပဲ သုံးဖြစ်မှာပါ။ `end()` ဆိုတဲ့ လုပ်ဆောင်ချက်တစ်ခုတော့ ရံဖန်ရံခါလိုနိုင်ပါတယ်။ ဥပမာ –

```
res.status(204).end()
```

`204` ရဲ့ သဘောကိုက No Content ဖြစ်လို့ Response Body မပို့သင့်ပါဘူး။ ဒါကြောင့် `end()` နဲ့ Response ကို အပြီးသတ်ပေးလိုက်တဲ့ သဘောပါ (ဒါမှမဟုတ် အဲ့ဒီလို နှစ်ခုတွဲ ရေးမနေတော့ဘဲ `sendStatus()` ကို သုံးလို့လည်း ရပါတယ်။ သူလည်း Status Code ချည်းပြန်ပို့တာပါပဲ။ ကိုယ့်ဘာသာ `end()` လုပ်ပေးစရာတော့ မလိုတော့ပါဘူး)။

Response Header တွေ သတ်မှတ်လိုရင် `res.set()` ကိုသုံးနိုင်ပါတယ်။ ဥပမာ –

```
res.set({
    "Location": "http://domain/api/people/3",
    "X-Rate-Limit-Limit": 60,
});
```

`res.append()` ကိုလည်းသုံးနိုင်ပါတယ်။ သူကတော့ Header တွေကို အခုလိုတစ်ခါတည်း အကုန်မ သတ်မှတ်ဘဲ တစ်ခုချင်း ထပ်တိုးချင်ရင် အသုံးဝင်ပါတယ်။

```
res.append("X-Rate-Limit-Remaining": 58);
```

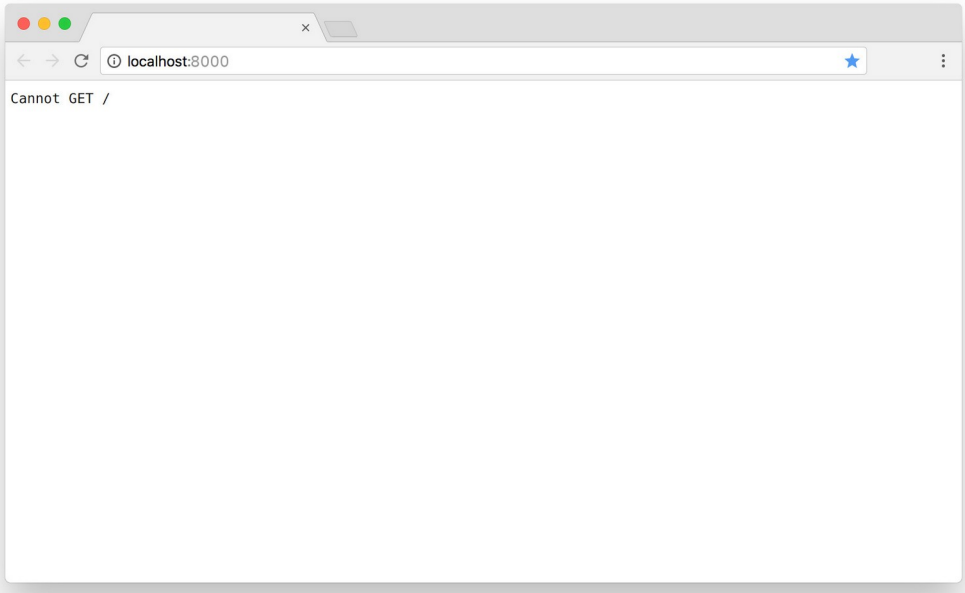
ဟိုအပေါ်မှာ ပေးထားတဲ့ကုဒ်နမူနာမှာ `res.status(200).json(data)` လို့ပြောတဲ့အတွက် Status Code `200`, `Content-type: application/json` နဲ့ `data` ကို Response Body အဖြစ် ပြန်ပို့ပေးသွားမှာပါ။ ပြီးတော့မှ `listen()` ကိုသုံးပြီး Server Run ခိုင်းထားပါတယ်။ Port နံပါတ်နဲ့ Callback Function ပေးရပါတယ်။ Port နံပါတ်ကို `8000` လို့ပေးထားပြီး Callback Function ကတော့ Server Run ပြီးရင် အလုပ်လုပ်ပေးသွားမှာ ဖြစ်ပါတယ်။

စမ်းကြည့်လို့ ရပါတယ်။ ရေးထားတဲ့ကုဒ်ကို အခုလို node နဲ့ Run ပေးရပါမယ်။

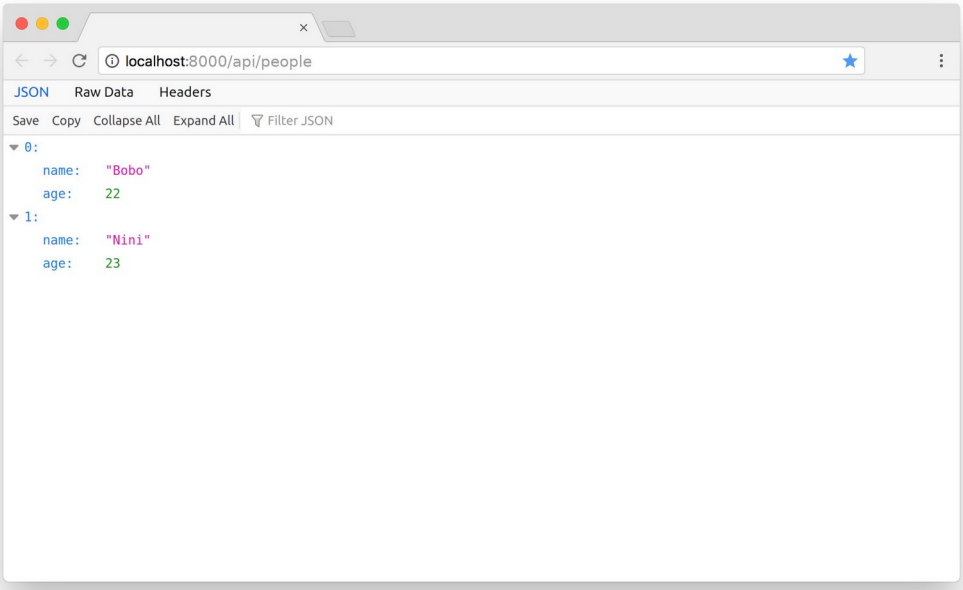
```
node index.js

Server running at port 8000…
```

Server Run နေပြီဖြစ်လို့ နှစ်သက်ရာ Web Browser တစ်ခုကိုဖွင့်ပြီး `localhost:8000` နဲ့ စမ်းကြည့်လို့ရပါတယ်။ အခုလိုတွေ့ရပါမယ်။



ဘာမှမပါတဲ့ Root URL အလွတ်ဖြစ်နေလို့ အခဲ့ဒီလိုတွေ့ရတာပါ။ ဒါကြောင့် `localhost:8000/api/people` ကို စမ်းကြည့်လိုက်ပါ။ အခုလိုတွေ့ရပါလိမ့်မယ်။



ဒါဟာ API တစ်ခုဖန်တီးလိုက်တာပါပဲ။ နောက်ထပ်သိသင့်တာလေး တစ်ချို့ထပ်မှတ်ပါ။ ပထမတစ်ခုကတော့ Dynamic URL သတ်မှတ်ပုံသတ်မှတ်နည်း ဖြစ်ပါတယ်။ စောစောကနမူနာမှာ URL က Static ပါ။ အသေသတ်မှတ်ထားလို့ သတ်မှတ်ထားတဲ့အတိုင်း အတိအကျအသုံးပြုမှ အလုပ်လုပ်ပါတယ်။ Dynamic URL တော့ URL ရဲ့ဖွဲ့စည်းပုံ သတ်မှတ်ထားတဲ့အတိုင်း မှန်ရမယ်၊ ဒါပေမယ့် တန်ဖိုးပြောင်းလို့ရတဲ့ URL အမျိုးအစားပါ။ ဒီလိုသတ်မှတ်ပေးနိုင်ပါတယ်။

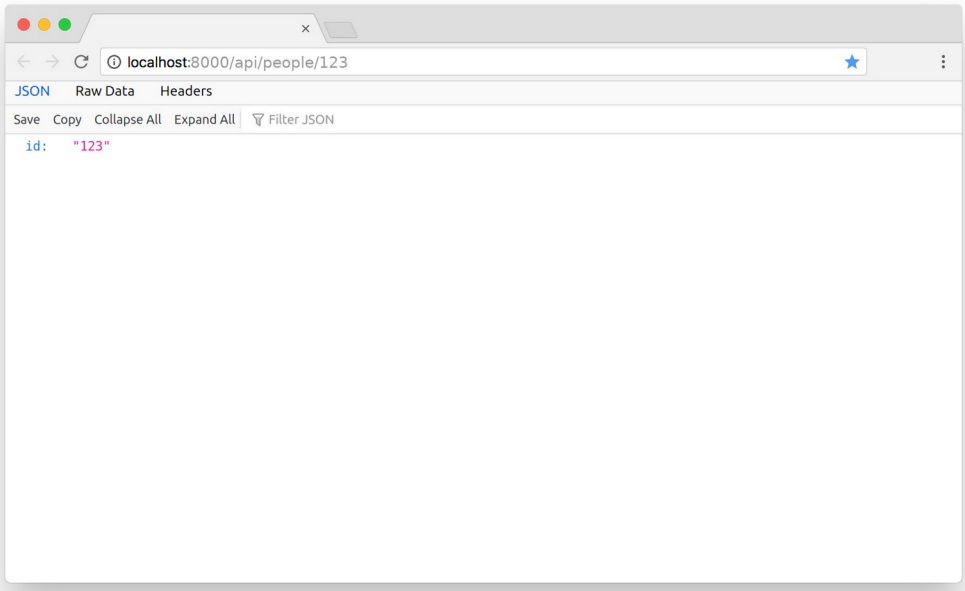
```
app.get("/api/people/:id", function(req, res) {
    const = req.params.id;

    return res.status(200).json({ id });
});
```

URL သတ်မှာ `:id` ဆိုတဲ့သတ်မှတ်ချက် ပါသွားတာကိုသတိပြုပါ။ `:id` နေရာမှာ နှစ်သက်ရာတန်ဖိုးကို ပေးပြီးတော့ သုံးနိုင်တဲ့ URL ဖြစ်ပါတယ်။ `:id` နေရာမှာ ပေးလာတဲ့ တန်ဖိုးကို `req.params.id` ဆိုပြီးတော့ ပြန်ယူလို့ရပါတယ်။ သတ်မှတ်တုံးက `:id` လို့သတ်မှတ်ခဲ့လို့ ပြန်ယူတဲ့အခါ `params.id` ဆိုပြီး ပြန်ယူရတာပါ။ `id` မှ မဟုတ်ပါဘူး၊ တစ်ခြားနှစ်သက်ရာအမည်ပေးပြီး လက်ခံနိုင်ပါတယ်။ ဒါပေမယ့်

REST သဘောသဘာဝအရ ကျွန်တော်တို့ကတော့ `:id` ကိုပဲ သုံးဖြစ်မှာပါ။

ဒီကုဒ်တွေ ရေးဖြည့်ပြီးရင် စမ်းကြည့်နိုင်ဖို့အတွက် API Server ကို Ctrl + C နဲ့ ရပ်လိုက်ပြီး နောက်တစ်ခါ `node index.js` နဲ့ ပြန် Run ပေးဖို့ လိုအပ်ပါတယ်။ အသစ်ထပ်ထည့်လိုက်တဲ့ကုဒ် အလုပ်လုပ်ဖို့ အတွက် အခုလို Server ကို ပြန်စပေးရတာပါ။ သတိပြုပါ။ အစပိုင်းမှာ မေ့တတ်ပါတယ်။ မေ့ပြီး ပြန်မစလို့ ရေးထားတဲ့ကုဒ် အလုပ်မလုပ်ဘူး ထင်တတ်ပါတယ်။ ဒီလို ကိုယ့်ဘာသာ ပြန်စပေးစရာ မလိုတဲ့နည်းတွေ ရှိပေမယ့် ထည့်မပြောတော့ပါဘူး။ လောလောဆယ် ကုဒ်တွေဖြည့်ရေးပြီးရင် Server ပြန်စပေးရတယ်လို့ သာမှတ်ထားပေးပါ။ ပြီးရင် စမ်းကြည့်လို့ရပါတယ်။ `localhost:8000/api/people/123` ဆိုရင် အခုလို ပြန်ရမှာပါ။



123 အစား တစ်ခြားတန်ဖိုးတွေ ကြိုက်သလိုပြောင်းပေးပြီး စမ်းကြည့်နိုင်ပါတယ်။

နောက်တစ်ခုထပ်ပြီး မှတ်သင့်တာကတော့ URL Route တွေကို စာမျက်နှာခွဲရေးထားလို့ ရနိုင်ခြင်း ဖြစ်ပါ တယ်။ ပရောဂျက်ကြီးလာရင် ကုဒ်တွေခွဲရေးတယ်ဆိုတာ မဖြစ်မနေ လိုအပ်မှာပါ။ Node Module ရေးထုံး အတိုင်းပဲ ခွဲရေးနိုင်ပါတယ်။ `routes.js` ဆိုတဲ့ဖိုင်ထဲမှာ အခုလိုရေးပြီး စမ်းကြည့်ပါ။

```
const express = require("express");
const router = express.Router();

router.get("/people", function(req, res) {
    const people = [
        { name: "Bobo", age: 22 },
        { name: "Nini", age: 23 },
    ];

    return res.status(200).json(people);
});

router.get("/people/:id", function(req, res) {
    const id = req.params.id;

    return res.status(200).json({ id });
});

module.exports = router;
```

ဒီတစ်ခါ app မပါတော့ပါဘူး။ အဲ့ဒီအစား router ကိုအသုံးပြုပြီး URL တွေ သတ်မှတ်ထားပါတယ်။ ပြီးတော့မှအဲ့ဒီ Router တစ်ခုလုံးကို Export လုပ်ပေးထားပါတယ်။ ဒါကြောင့် လိုအပ်တဲ့နေရာက ယူသုံးလို့ရပါပြီ။ index.js မှာ အခုလို ယူသုံးပြီး ရေးစမ်းကြည့်လို့ရပါတယ်။

```
const express = require("express");
const app = express();
const routes = require("./routes");

app.use("/api", routes);

app.listen(8000, function() {
    console.log("Server running at port 8000...");
});
```

စောစောကလို URL သတ်မှတ်ချက်တွေ မပါတော့ပါဘူး။ အဲ့ဒီအစား routes.js ကို Import လုပ်ပြီး use() ရဲ့အကူအညီနဲ့ /api တွေအားလုံး routes.js မှာ သတ်မှတ်ထားတဲ့ routes တွေကို သုံးရမယ်လို့ ပြောလိုက်တာပါ။ ရလဒ်က စောစောက နမူနာနဲ့ အတူတူပဲဖြစ်မှာပါ။ ကွာသွားတာက Route တွေကို ဖိုင်ခွဲပြီး ရေးလိုက်ခြင်းသာ ဖြစ်ပါတယ်။ ဆက်လက်ဖော်ပြမယ့် နမူနာမှာတော့ ဖိုင်တွေ ခွဲမနေတော့ပါဘူး။ တစ်မျက်နှာထဲမှာပဲ လိုတာအကုန် ရေးသွားမှာ ဖြစ်ပါတယ်။

Project

တစ်ကယ်တော့ သိသင့်တာ စုံအောင်ကြိုပြောထားပြီးမို့လို့ ပရောဂျက်လို့သာ နာမည်တပ်ထားတာ အများကြီးပြောစရာ မကျန်တော့ပါဘူး။ ရေးစရာရှိတဲ့ ကုဒ်နမူနာကို တန်းရေးပြလိုက်ယုံပဲ ကျန်ပါတော့တယ်။ MongoDB ကို Express ကနေ ချိတ်နိုင်ဖို့အတွက် `mongojs` လို့ခေါ်တဲ့ Package ကို ရယူအသုံးပြုပါမယ်။ MongoDB ရဲ့ Official Package က `mongo` ပါ။ ဒါပေမယ့် `mongojs` ကိုသုံးဖို့ ရွေးပါတယ်။ ဘာဖြစ်လို့လဲ ဆိုတော့ သူက Mongo Shell ထဲက Command တွေအတိုင်း သုံးလို့ရအောင် စီစဉ်ပေးထားတဲ့အတွက် ထပ်လေ့လာစရာ မလိုဘဲ သိပြီးသားအတိုင်း ဆက်သုံးနိုင်မှာမို့လို့ပါ။

နောက်ထပ် Package တစ်ခုလည်း လိုပါသေးတယ်။ `body-parser` လို့ခေါ်ပါတယ်။ ဟိုအရင်ကဆိုရင် Express မှာ လိုအပ်တာအကုန် ပါပါတယ်။ `body-parser` ဆိုတာ Express ရဲ့ အစိတ်အပိုင်းတစ်ခုပါ။ နောက်ပိုင်းတော့ Express က မလိုတာမသုံးဘဲ လိုတာပဲ ရွေးသုံးလို့ရအောင် Package တွေ ခွဲထုတ်ပြစ်ပါတယ်။ Cookie စီမံတာက Package တစ်ခု၊ File Upload စီမံတာက Package တစ်ခု၊ Request Body စီမံတာက Package တစ်ခု စသဖြင့်ပါ။ ကောင်းပါတယ်။ အခု ပရောဂျက်မှာဆိုရင် Cookie တို့ File Upload တို့ သုံးဖို့အစီအစဉ် မရှိပါဘူး။ မလိုဘဲ ပါနေရင်ရှုပ်ပါတယ်။ လိုတဲ့ Package ကိုသာ ရွေးပြီးတော့ ထည့်လိုက်ယုံပါပဲ။ Request Body ကို စီမံတယ်ဆိုတာ Request နဲ့အတူ ပါဝင်လာတဲ့ JSON String တွေ URL Encoded String တွေကို JSON Object ပြောင်းပေးတဲ့ အလုပ်ကို ဆိုလိုတာပါ။

အခုလို `install` လုပ်ပေးလိုက်ပါ။

```
npm i mongojs body-parser
```

ပြီးရင် `index.js` မှာ ရေးရမယ့်ကုဒ်ကိုတစ်ပိုင်းချင်း ပြသွားပါမယ်။ ပထမတစ်ပိုင်း စကြည့်ပါ။

```
const express = require("express");
const app = express();

const mongojs = require("mongojs");
const db = mongojs("travel", [ "records" ]);

const bodyParser= require("body-parser");

app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
```

ပထမဆုံး `express` ကို Import လုပ်ပါတယ်။ ပြီးတဲ့အခါ `mongojs` ကို Import လုပ်ပါတယ်။ `mongojs` ကို သုံးပြီး MongoDB Server ကိုချိတ်နိုင်ပါတယ်။ ပထမ Parameter က Database Name ဖြစ်ပြီး ဒုတိယ Parameter က Collection Array ကို ပေးရတာပါ။ Collection တစ်ခုပဲရှိလို့ Array Content လည်း နမူနာမှာ တစ်ခုပဲရှိတာပါ။ MongoDB Server Run ထားဖို့လိုမယ်ဆိုတာကို သတိပြုပါ။

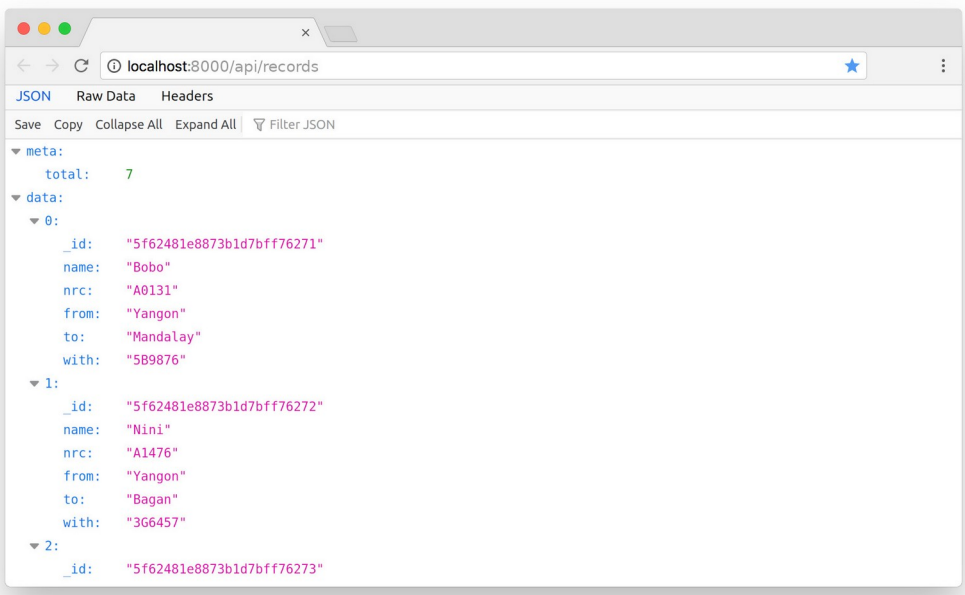
နောက်တစ်ဆင့်မှာ `body-parser` ကို Import လုပ်ပြီး `use()` နဲ့ သူ့ရဲ့ လုပ်ဆောင်ချက်နှစ်ခုကို ကြားဖြတ်ပြီး လုပ်ခိုင်းထားပါတယ်။ URL Encoded String တွေကို JSON အနေနဲ့ Parse လုပ်ဖို့ရယ်၊ JSON String တွေကို JSON အနေနဲ့ Parse လုပ်ဖို့ရယ် ပြောထားတာပါ။ `urlencoded()` အတွက်ပေးထားတဲ့ `extended: false` Option က မဖြစ်မနေ ပေးရမယ်လို့ သတ်မှတ်ထားလို့ ထည့်ပေးထားတာပါ။ URL Encode/Decode လုပ်ဖို့အတွက် Node မှာ Build-in လုပ်ဆောင်ချက် ပါပါတယ်။ အဲ့ဒါကို သုံးချင်ရင် `extended: false` လို့ ပေးရတာပါ။ Node ရဲ့ လုပ်ဆောင်ချက်ကို မသုံးဘဲ `body-parser` နဲ့အတူ ပါတဲ့ လုပ်ဆောင်ချက်ကို သုံးချင်ရင် `true` ပေးရမှာ ဖြစ်ပါတယ်။ ဒီလို ကြေညာသတ်မှတ်ပြီးနောက် Request Body မှာပါတဲ့ အချက်အလက်တွေကို JSON အနေနဲ့ `req.body` ကနေ အသင့်သုံးလို့ရသွားမှာ ဖြစ်ပါတယ်။

`use()` Function ကို Middleware တွေ ကြေညာဖို့သုံးပါတယ်။ Middleware ဆိုတာ လိုရင်းအနှစ်ချုပ်ကတော့ Request တစ်ခုဝင်လာတာနဲ့ ကြားထဲကဖမ်းပြီး အလုပ်လုပ်ပေးမယ့် လုပ်ဆောင်ချက်လေးတွေပါ။ `use()` ကိုသုံးပြီး သတ်မှတ်ထားတဲ့ လုပ်ဆောင်ချက်တိုင်းကို၊ Request ဝင်လာတိုင်း၊ ဝင်လာတဲ့ Request ပေါ်မှာ လုပ်ပေးလိုက်မှာ ဖြစ်ပါတယ်။

ဆက်လက်ပြီး Travel Records တွေအားလုံးကို ပြန်ပေးတဲ့ လုပ်ဆောင်ချက်ကိုရေးပါမယ်။ ဒီလိုပါ –

```
app.get("/api/records", function(req, res){
    db.records.find(function(err, data) {
        if(err) {
            return res.sendStatus(500);
        } else {
            return res.status(200).json({
                meta: { total: data.length },
                data
            });
        }
    });
});
```

Request Method GET ဖြစ်ရပါမယ်။ URL က `/api/records` ဖြစ်ပါတယ်။ ပထမဆုံး `find()` နဲ့ ရှိသမျှ `records` တွေအကုန်ထုတ်ယူပါတယ်။ အကြောင်းအမျိုးမျိုးကြောင့် Error ဖြစ်နေရင် `500` Internal Server Error ပြန်ပို့ပါတယ်။ Error မဖြစ်ရင်တော့ Data Envelope ထဲမှာ `data` နဲ့အတူ `meta.total` ပါ ထည့်ပေးလိုက်တာပါ။ ဒါကြောင့် အခုနေစမ်းကြည့်ရင် ရလဒ်က ဒီလိုဖြစ်မှာပါ။



ပြီးတဲ့အခါ တစ်လက်စတည်း Sorting တွေ၊ Paging တွေ၊ Filter တွေ အကုန်ပြည့်စုံအောင် ထည့်ပါမယ်။

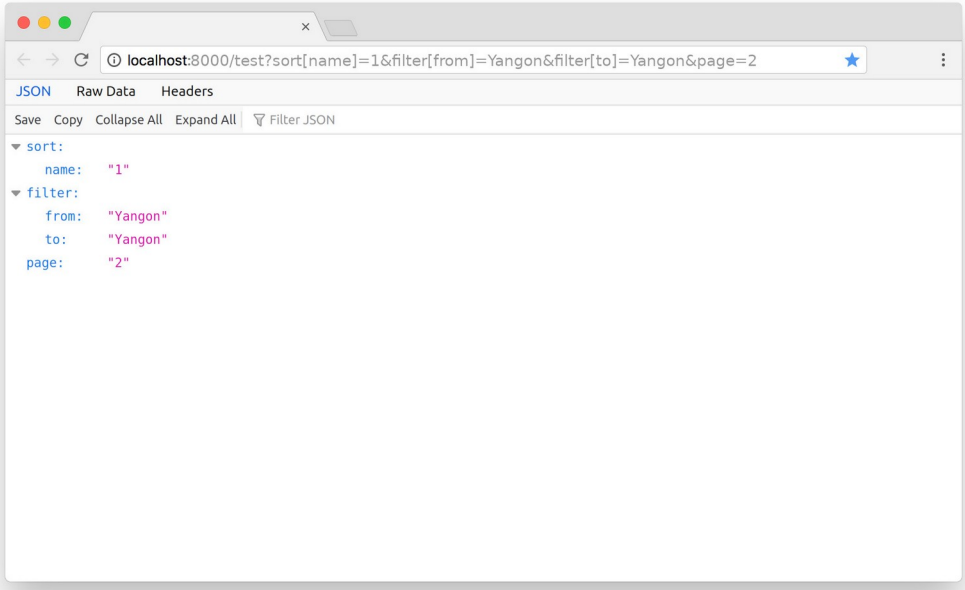
အခဲ့ဒါတွေမထည့်ခင် URL Query တွေကို Express က ဘယ်လိုလက်ခံ စီမံသလဲ စမ်းကြည့်လို့ရအောင် အခုလိုလေး အရင်ရေးကြည့်သင့်ပါတယ်။

```
app.get("/test", function(req, res) {
    return res.json(req.query);
});
```

ဒီကုဒ်က Request နဲ့အတူပါဝင်လာတဲ့ URL Query ကိုပဲ ပြန်ပေးထားတာပါ။ ဒါပေမယ့် JSON အနေနဲ့ ပြန်ပေးတာဖြစ်လို့ URL Query ကို JSON ပြောင်းလိုက်တဲ့အခါ ဘယ်လိုပုံစံရသလဲ လေ့လာကြည့်ဖို့ပါ။ ဒီ URL နဲ့ စမ်းကြည့်ပါ။

```
localhost:8000/test?sort[name]=1&filter[from]=Yangon&filter[to]=Yangon&page=2
```

ရလဒ်က အခုလိုဖြစ်မှာပါ။ သေချာလေး ဂရုပြုကြည့်ပေးပါ။



ကိုယ့်ဘက်က ဘာမှလုပ်ပေးစရာမလိုဘဲ URL Query က အသင့်သုံးလို့ရတဲ့ JSON Structure လေးနဲ့ ရနေတာကို တွေ့ရနိုင်ပါတယ်။ စနစ်ကျတဲ့ API URL ကို အသုံးပြုခြင်းရဲ့ အကျိုးပါ။ ဒီသဘောကို မြင်ပြီဆိုရင် စောစောက `/records` အတွက်ရေးထားတဲ့ ကုဒ်ကို အခုလိုပြင်ပေးလိုက်ပါ။

```
app.get("/api/records", function(req, res){

    const options = req.query;

    // validate options, send 400 on error

    const sort = options.sort || {};
    const filter = options.filter || {};
    const limit = 10;
    const page = parseInt(options.page) || 1;
    const skip = (page - 1) * limit;

    for(i in sort) {
        sort[i] = parseInt(sort[i]);
    }

    db.records.find(filter)
        .sort(sort)
        .skip(skip)
        .limit(limit, function(err, data) {
            if(err) {
                return res.sendStatus(500);
            } else {
                return res.status(200).json({
                    meta: { total: data.length },
                    data
                });
            }
        });
});
```

ဒါ Filter, Sorting, Paging လုပ်ဆောင်ချက်အားလုံး ပါဝင်သွားတာပါ။ Sort နဲ့ Filter အတွက် တန်ဖိုးတွေကို Client ပေးတဲ့အတိုင်း URL Query ကနေပဲ ယူထားတာပါ။ Paging အတွက် Client ကပေးတဲ့ Page နံပါတ်ကိုသုံးပြီး `skip` တန်ဖိုးကို ကိုယ့်ဘာသာ တွက်ယူပါတယ်။ Sorting အတွက် Client ကပေးတဲ့ `1`, `-1` Value တွေဟာ String အနေနဲ့လာမှာပါ။ MongoDB က Integer နဲ့မှအလုပ်လုပ်တာမို့လို့ `sort` Options တွေကိုတော့ Loop လုပ်ပြီး Integer ပြောင်းထားပါတယ်။ ကျန်တာကတော့ ရလာတဲ့ Options တွေကို `filter`, `sort`, `skip`, `limit` စသဖြင့် သူ့နေရာနဲ့သူ ထည့်ပေးလိုက်တာပါပဲ။ ဒီကုဒ်မျိုးက စာနဲ့ရှင်းတာထက်စာရင် ရေးထားတဲ့ ကုဒ်ကိုဖတ်ကြည့်တာ ပိုထိရောက်ပါတယ်။ ဖတ်ကြည့်လိုက်ပါ။ နားလည်ရလွယ်အောင်ရေးထားပါတယ်။

တစ်ကယ်လက်တွေ့ ပရောဂျက်မှာဆိုရင်တော့ Client ပေးတဲ့ Option တွေကို Validate လုပ်သင့်ပါသေးတယ်။ မပါသင့်တာတွေပါလာမှာ စိုးလို့ပါ။ သို့မဟုတ် ပေးပုံပေးနည်း မှားနေတာတွေ ဖြစ်မှာစိုးလို့ပါ။ ဒီမှာတော့ အခဲ့ဒီကိစ္စကို ထည့်မစစ်တော့ပါဘူး။ စမ်းကြည့်လို့ ရနေပါပြီ။ ဥပမာစမ်းချင်ရင် ဒီလိုစမ်းနိုင်ပါတယ်။

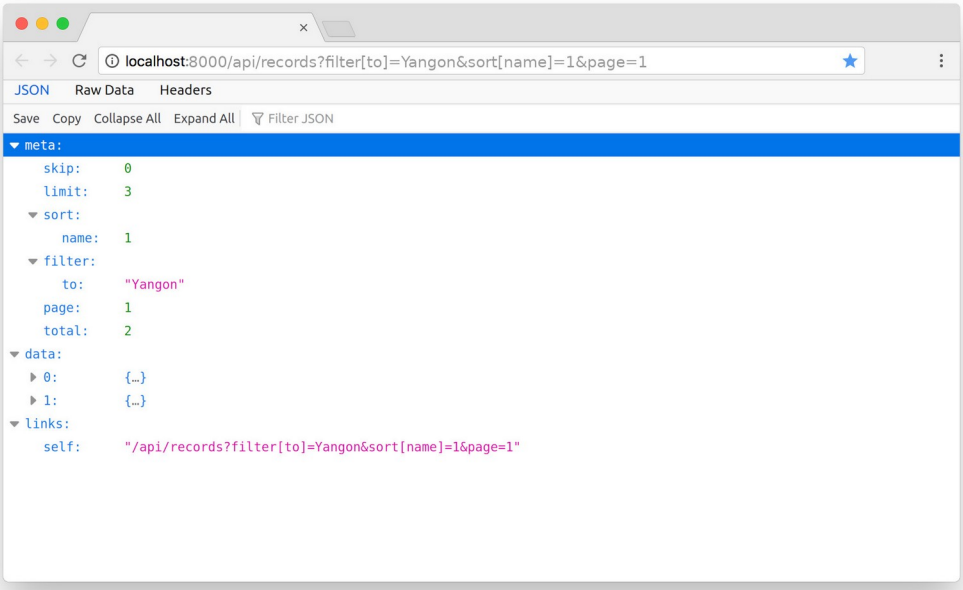
localhost:8000/api/records?filter[to]=Yangon&sort[name]=1&page=1

ဒါဆိုရင် `to: Yangon` တွေအားလုံးကို `name` နဲ့စီပြီး ပြန်ပေးမှာဖြစ်ပါတယ်။ `filter` မှာ `name, nrc, from, to, with` ကြိုက်တာနဲ့ ရွေးထုတ်လို့ရလို့ ကျွန်တော်တို့ပရောဂျက်မှာ ပါစေချင်တဲ့ နာမည်နဲ့ ပြန်ရှာထုတ်လို့ ရတာတွေ၊ မှတ်ပုံတင်နဲ့ ပြန်ရှာထုတ်လို့ ရတာတွေ၊ မြို့အလိုက် အကုန်ပြန်ထုတ်ယူလို့ ရတာတွေ တစ်ချက်တည်းနဲ့ အကုန်ပါဝင်သွားတာ ဖြစ်ပါတယ်။

Request ဘက်ကလာတဲ့ Query တွေကို လက်ခံအလုပ်လုပ်ပုံ ပြည့်စုံပြီဆိုပေမယ့် Response Data Envelope မှာ မပြည့်စုံသေးပါဘူး။ ဒါကြောင့် အခဲ့ဒီကုဒ်မှာပဲ `json()` Response ကို ဒီလိုလေး ထပ်ပြင်ပေးဖို့ လိုပါသေးတယ်။

```
res.status(200).json({
    meta: {
        skip,
        limit,
        sort,
        filter,
        page,
        total: data.length,
    },
    data,
    links: {
        self: req.originalUrl,
    }
});
```

ဒါဆိုရင် အသုံးဝင်တဲ့ `meta` Information တွေ ပါဝင်သွားမှာဖြစ်လို့ စမ်းကြည့်လိုက်ရင် ရလဒ်က ဒီလိုပုံစံရမှာဖြစ်ပါတယ်။



`links` အတွက်တော့ `self` တစ်ခုပဲ ထည့်ထားပါတယ်။ တစ်ကယ်တော့ Paging အတွက် `next`, `prev`, `first`, `last` စသဖြင့် တစ်ခြားလိုအပ်မယ့် Links တွေ တွက်ပြီးထည့်ပေးသင့်ပါသေးတယ်။ ရေးရမယ့်ကုဒ် များသွားမှာမို့လို့ မထည့်တော့ပါဘူး။ ဆိုလိုရင်းကို သဘောပေါက်မယ်လို့ယူဆပါတယ်။

ဆက်လက်ပြီးတော့ Record အသစ်တွေ ထပ်ထည့်လို့ရတဲ့ လုပ်ဆောင်ချက်ကို ဆက်သွားပါမယ်။ Record အသစ်ထည့်ဖို့အတွက် မထည့်ခင် Package လေးတစ်ခုအရင် Install လုပ်ကြပါဦးမယ်။ Request Body မှာပါလာတဲ့ အချက်အလက်တွေကို Validation စစ်ဖို့အတွက် `express-validator` ကို Install လုပ်မှာပါ။ ဒီလိုပါ –

```
npm i express-validator
```

ပြီးတဲ့အခါ အခုလို Import လုပ်ပြီး စသုံးလို့ရပါပြီ။

```
const {
    body,
    param,
    validationResult
} = require("express-validator");
```

`body`, `param` နဲ့ `validationResult` ဆိုတဲ့ (၃) ခု `express-validator` ကနေ Import လုပ်ယူထားတာပါ။ `body` ကိုသုံးပြီး Request Body တွေကို Validate စစ်ပါမယ်။ `param` ကိုသုံးပြီး Dynamic Route တန်ဖိုးတွေကို Validate စစ်ပါမယ်။ တစ်ခြားဟာတွေ ကျန်ပါသေးတယ်။ `query` တို့ `header` တို့ကိုလည်း စစ်ချင်ရင် စစ်လို့ရပါသေးတယ်။ စစ်ဆေးမှုရလဒ်ကို `validationResult` ကနေ ပြန်လည်ရယူရမှာပါ။ Record အသစ်ထည့်တဲ့ကုဒ်တွေရေးလို့ရပါပြီ။

```
app.post("/api/records", [

    body("name").not().isEmpty(),
    body("from").not().isEmpty(),
    body("to").not().isEmpty(),

], function(req, res) {

    const errors = validationResult(req);

    if (!errors.isEmpty()) {
        return res.status(400).json({ errors: errors.array() });
    }

    db.records.insert(req.body, function(err, data) {
        if(err) {
            return res.status(500);
        }

        const _id = data._id

        res.append("Location", "/api/records/" + _id);
        return res.status(201).json({ meta: { _id }, data });
    });
});
```

Method `POST` ကိုသုံးရမှာဖြစ်ပြီး URL ကတော့ `/api/records` ပါပဲ။ Callback Function မတိုင်ခင်ကြားထဲက ဒုတိယ Parameter အနေနဲ့ Validation စစ်ထားပါတယ်။ ဒီကုဒ်မှာတော့ "ဘာတွေပါရမယ်"

လို့ပဲ စစ်ထားတာပါ။ ကားနံပါတ်မို့လို့ စာလုံး (၆) လုံးပါရမယ်တို့၊ လူနာမည်မို့လို့ Special Character တွေ မပါရဘူးတို့၊ စသဖြင့် အသေးစိတ် စစ်မထားပါဘူး။ စစ်ချင်တယ်ဆိုရင် စစ်လို့ရတဲ့ Rule တွေအားလုံးကို ဒီမှာကြည့်လို့ ရပါတယ်။

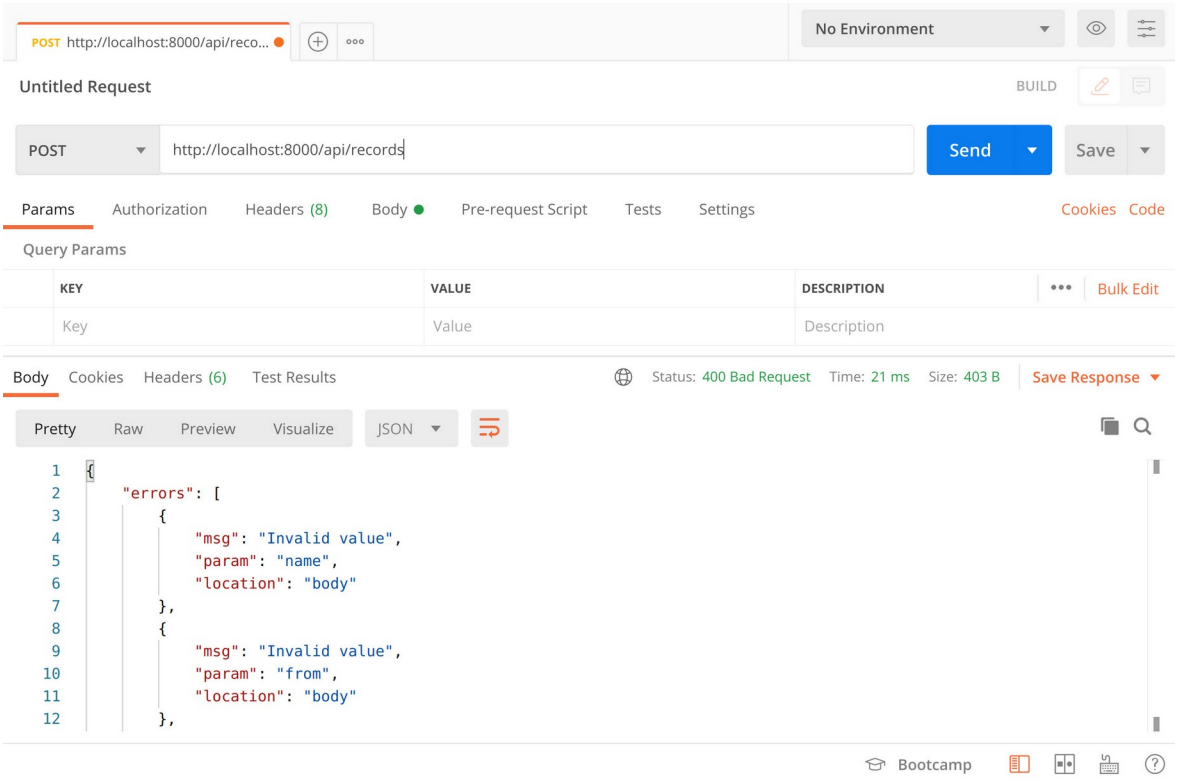
– https://github.com/validatorjs/validator.js#validators

ပြီးတဲ့အခါ Validation Result ကို စစ်ကြည့်ပြီး Error ရှိနေတယ်ဆိုရင် `400` ကို ပြန်ပေးထားပါတယ်။ Validation Error မရှိဘူးဆိုတော့မှ `insert()` နဲ့ထည့်လိုက်တာပါ။ `POST` နဲ့ အသစ်ထည့်တာဖြစ်လို့ အောင်မြင်တဲ့အခါ `201` ကို ပြန်ပေးပြီး `Location` Header ကိုပါ တွဲဖက် ထည့်သွင်းပေးထားပါတယ်။ Response Body ကိုတော့ ထုံးစံအတိုင်း Data Envelope နဲ့ တွဲပြီး ပြန်ပေးပါတယ်။

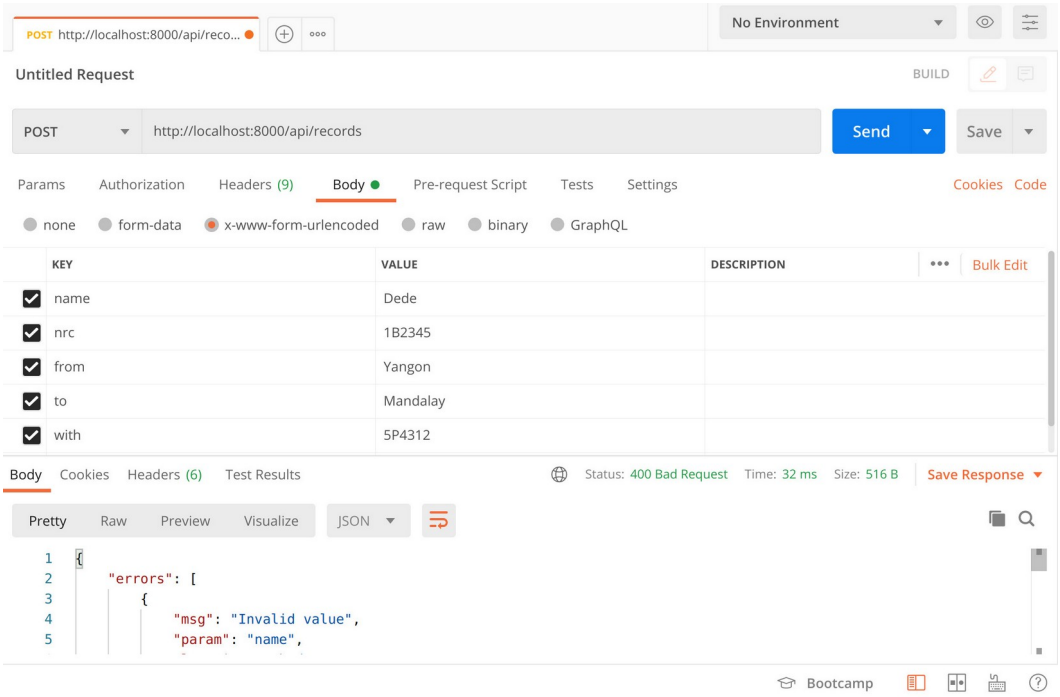
ဒါကို စမ်းကြည့်နိုင်ဖို့အတွက် API Testing Tool တစ်ခုခုတော့ လိုပါလိမ့်မယ်။ cURL, Insomnia, Postman စသဖြင့် Tool အမျိုးမျိုးရှိတဲ့ထဲက **Postman** လို့ခေါ်တဲ့ API Testing ပရိုဂရမ်တစ်ခုကို သုံးမှာပါ။ ဒီမှာ Download လုပ်ပြီး Install လုပ်လို့ရပါတယ်။

– https://www.postman.com/

Install လုပ်ပြီးတဲ့အခါ ဖွင့်လိုက်ပါ။ ပြီးရင်ဆက်လက်ဖော်ပြတဲ့ပုံမှာ ပြထားသလို Method နေရာမှာ POST ကိုရွေးပြီး API URL ကို အပြည့်အစုံ မှန်အောင်ရိုက်ထည့်လိုက်ပါ။ Send နှိပ်ကြည့်ရင် ကျွန်တော်တို့ရဲ့ API Server ကို Postman က Request ပေးပို့သွားမှာ ဖြစ်ပါတယ်။



နမူနာပုံမှာ Response အနေနဲ့ Validation Error Message တွေကို တွေ့မြင်ရခြင်းဖြစ်ပါတယ်။ Request Body အတွက် လိုအပ်မယ့်အချက်အလက်တွေ ထည့်ပြီးစမ်းနိုင်ဖို့ URL Bar အောက်က **Body** Tab ကိုနှိပ်ပြီး **x-www-form-urlencoded** ကိုထပ်ဆင့်ရွေးပါ။ နောက်တစ်မျက်နှာက နမူနာပုံကိုကြည့်ပါ။ ကျွန်တော်တို့ API က JSON ရော URL Encoded ကိုပါ လက်ခံအလုပ်လုပ်နိုင်ပါတယ်။ တစ်ကယ့် Request Data အမှန်ကတော့ JSON ဖြစ်သင့်ပါတယ်။ ဒါပေမယ့် အခုလောလောဆယ် စမ်းကြည့်ဖို့အတွက် JSON တွေကို Format မှန်အောင် ကိုယ့်ဘာသာ ရိုက်ထည့်နေရမှာစိုးလို့ ပိုပြီးထည့်ရလွယ်တဲ့ URL Encoded နဲ့ပဲ စမ်းကြည့်လိုက်ပါ။



အခုဆိုရင် အသစ်ထည့်တဲ့လုပ်ဆောင်ချက်လည်း ရသွားပါပြီ။ တစ်ကယ်ဆိုရင်တော့ ကုဒ်ဒီဇိုင်းကို ဒီထက်ပိုကောင်းအောင် ပြင်ရဦးမှာပါ။ `req.body` ကြီးကို Database ထဲ ပစ်ထည့်လိုက်တာ လက်တွေ့မကျပါဘူး။ မလိုလားအပ်တာတွေ ဝင်ကုန်ပါမယ်။ ဒါပေမယ့် အခုရှင်းပြချင်တာက ကုဒ်ဒီဇိုင်းပိုင်းမဟုတ်ဘဲ၊ API ရဲ့ သဘောသဘာဝကို ရှင်းပြချင်တာမို့လို့ ဖတ်ရ၊ နားလည်ရလွယ်အောင် နမူနာကုဒ်တွေကို အလွယ် ရေးပြထားတယ်ဆိုတာကို သတိပြုပေးပါ။

ဆက်လက်ပြီးတော့ ရှိပြီးသား အချက်အလက်တွေ ပြင်ဆင်ပေးနိုင်တဲ့လုပ်ဆောင်ချက်ကို ဆက်ရေးသွားပါမယ်။ `PUT` နဲ့ `PATH` နှစ်မျိုးရှိပြီး နှစ်မျိုးလုံးနဲ့ နမူနာရေးပြပါမယ်။

```
app.put("/api/records/:id", [

    param("id").isMongoId(),

], function(req, res) {
    const _id = req.params.id;

    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        return res.status(400).json({ errors: errors.array() });
    }

    db.records.count({
        _id: mongojs.ObjectId(_id)
    }, function(err, count) {
        if(count) {
            const record = {
                _id: mongojs.ObjectId(_id),
                ...req.body
            };

            db.records.save(record, function(err, data) {
                return res.status(200).json({
                    meta: { _id },
                    data
                });
            });
        } else{
            db.records.save(req.body, function(err, data) {
                return res.status(201).json({
                    meta: { _id: data._id },
                    data
                });
            });
        }
    });
});
```

Request Method `PUT` နဲ့လာတဲ့အခါ URL မှာ ID ထည့်ပေးရပါတယ်။ အဲ့ဒီ ID ဟာ MongoDB က အသုံးပြုတဲ့ ID ဟုတ်မဟုတ် `param()` နဲ့ Validation စစ်ထားပါတယ်။ Request Body ကိုတော့ စစ်မပြတော့ပါဘူး။ ကုဒ်တိုသွားအောင်လို့ပါ။ တစ်ကယ်တမ်း ပြည့်စုံချင်ရင်တော့ စစ်ရမှာပါ။ ရေးထားတဲ့ ကုဒ်အရ ပေးလာတဲ့ ID နဲ့ ရှာကြည့်ပြီး ရှိရင် Update လုပ်ပေးမှာဖြစ်ပါတယ်။ `PUT` Method ဖြစ်ပြီး `save()` Function ကိုသုံးတဲ့အတွက် အစားထိုးတဲ့နည်းနဲ့ Update လုပ်မှာပါ။ ရှာလို့ မတွေ့ရင်တော့ အသစ်တစ်ခုအနေနဲ့ ထည့်ပေးမှာဖြစ်ပါတယ်။ `_id` ကို `ObjectId()` ထဲမှာ ထည့်ပေးရတာကို သတိပြုပါ။

PATCH အတွက် အခုလိုထပ်ရေးပြီးစမ်းကြည့်နိုင်ပါတယ်။

```
app.patch("/api/records/:id", function(req, res) {
    const _id = req.params.id;

    db.records.count({
        _id: mongojs.ObjectId(_id)
    }, function(err, count) {
        if(count) {
            db.records.update(
                { _id: mongojs.ObjectId(_id) },
                { $set: req.body },
                { multi: false },
                function(err, data) {
                    db.records.find({
                        _id: mongojs.ObjectId(_id)
                    }, function(err, data) {
                        return res.status(200).json({
                            meta: { _id }, data
                        });
                    });
                }
            )
        } else {
            return res.sendStatus(404);
        }
    });
});
```

ဒီနမူနာမှာတော့ ID ကို MongoDB ID ဟုတ်မဟုတ် Validation စစ်တဲ့ကုဒ် ထပ်ထည့်မပေးတော့ပါဘူး။ ပြီးခဲ့တဲ့ နမူနာမှာ ထည့်ပြပြီးသား မို့လို့ပါ။ ပြည့်စုံချင်ရင် နောက်မှကိုယ့်ဘာသာ ထပ်ထည့်လိုက်ပါ။ ပြီးတော့ နည်းနည်းကွဲပြားသွားအောင် ID နဲ့ရှာမတွေ့ရင် အသစ်ထည့်မပေးတော့ပါဘူး။ `404` ကိုပဲပြန်ပေးလိုက်ပါတော့တယ်။ ရှာလို့ တွေ့ပြီဆိုတော့မှာ `update()` နဲ့ ပြင်ပေးလိုက်ပါတယ်။ PATCH ဖြစ်တဲ့အတွက် လိုတဲ့အပိုင်းကိုပဲရွေးပြီး Update လုပ်ပေးလိုက်တာပါ။

တစ်ခုပဲ ကျန်ပါတော့။ Delete ပါ။ အခုလိုရေးပြီး စမ်းကြည့်နိုင်ပါတယ်။

```
app.delete("/api/records/:id", function(req, res) {
    const _id = req.params.id;

    db.records.count({
        _id: mongojs.ObjectId(_id)
    }, function(err, count) {
        if(count) {
            db.records.remove({
                _id: mongojs.ObjectId(_id)
            }, function(err, data) {
                return res.sendStatus(204);
            });
        } else{
            return res.sendStatus(404);
        }
    });
});
```

ID နဲ့ ရှာလို့တွေ့ရင်ဖျက်ပြီး `204` ကို ပြန်ပေးပါတယ်။ မတွေ့ရင်တော့ `404` ကို ပြန်ပေးလိုက်တာပါ။ Response Body တော့ ပြန်မပေးတော့ပါဘူး။

အခုဆိုရင် Create, Read, Update, Delete လုပ်ဆောင်ချက်အပြည့်စုံပါဝင်တဲ့ API Service လေးတစ်ခု ရသွားပြီပဲ ဖြစ်ပါတယ်။ ရည်ရွယ်ချက်ကတော့ ခရီးသွားမှတ်တမ်းတွေကို ထည့်သွင်းသိမ်းဆည်းထားပြီး လိုအပ်တဲ့အခါ `name`, `nrc`, `from`, `to`, `with` စသဖြင့် ကိုယ်လိုတဲ့ အချက်အလက်နဲ့ ပြန် Filter လုပ်ပြီး စစ်ဆေးကြည့်နိုင်ဖို့ပဲ ဖြစ်ပါတယ်။

တစ်ကယ့်လက်တွေ့အသုံးချအဆင့် ရောက်ချင်ရင်တော့ နာမည်အတိအကျ သိစရာမလိုဘဲ Search လုပ်နိုင်တဲ့ လုပ်ဆောင်ချက်တွေ၊ `byTrain`, `byCar` စသဖြင့် Transportation အမျိုးမျိုးနဲ့ သိမ်းလို့ရအောင် လုပ်ပေးတာတွေ၊ သွားခဲ့တဲ့ ခရီးစဉ်တွေကို ချိတ်ဆက်ပြီး လမ်းကြောင်းပြတာတွေ၊ ခရီးစဉ်တစ်ခုနဲ့တစ်ခု အချိတ်အဆက် မမိဘဲ ကြားထဲမှာ ပျောက်နေရင် သတိပေးတာတွေ လုပ်လို့ရပါတယ်။ နမူနာကတော့ ဒီလောက်ဆိုရင် လုံလောက်ပြီမို့လို့ ဒါတွေထည့်မရေးတော့ပါဘူး။ ကိုယ်ဘာသာ Exercise လုပ်တဲ့သဘောနဲ့ စမ်းထည့်ချင်ရင် ထည့်လို့ရအောင် ပြောပြတဲ့သဘောပါ။ ရေးခဲ့တဲ့နမူနာကုဒ်တွေကို လိုအပ်ရင် ဒီမှာ Download လုပ်လို့ရပါတယ်။

- https://github.com/eimg/api-book

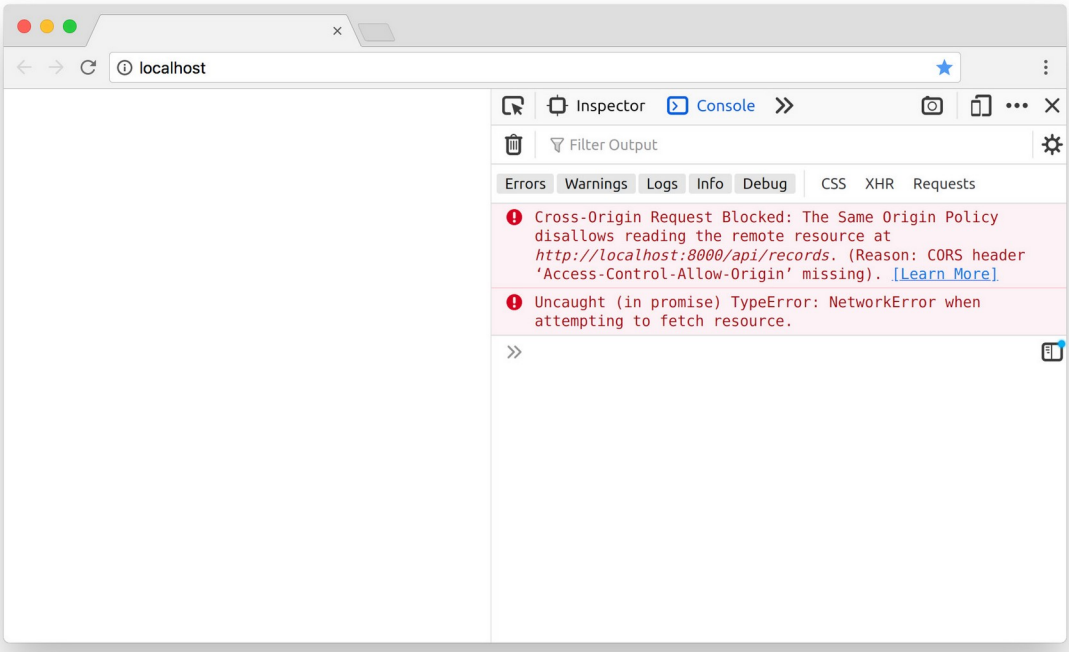
အခန်း (၉) – CORS

HTTP မှာ CORS လို့ခေါ်တဲ့ သဘောသဘာဝတစ်ခု ရှိပါသေးတယ်။ API လေ့လာသူများ မသိမဖြစ် သိထားဖို့ လိုအပ်ပါတယ်။ Cross-Origin Resource Sharing ဆိုတဲ့အဓိပ္ပါယ်ပါ။ Web Browser တွေမှာ အဓိက အားဖြင့် အသုံးပြုကြတဲ့ လုံခြုံရေးအစီအမံတစ်ခုပါ။

သူ့ရဲ့လိုရင်းသဘောကတော့ Origin (Host, Domain, Port) မတူရင် Request တွေ ပို့ခွင့်မပြုခြင်း ဖြစ်ပါတယ်။ Server က Origin မတူလည်းပဲ လက်ခံပါတယ်လို့ သီးသန့်ခွင့်ပြုထားမှသာ Request တွေကို ပေးပို့မှာ ဖြစ်ပါတယ်။ ဥပမာ – Client က `locahost` မှာအလုပ်လုပ်နေပြီး Server ကလည်း `localhost` မှာပဲအလုပ်လုပ်နေတာဆိုရင် Request တွေ ပေးပို့လို့ ရပါတယ်။ Origin တူလို့ပါ။ ဘာပြဿနာမှ မရှိပါဘူး။ Client က `localhost:3000` မှာ အလုပ်လုပ်နေပြီး Server က `localhost:8000` မှာအလုပ်လုပ်နေတာဆိုရင် CORS နဲ့ ညိသွားပါပြီ။ `localhost` ချင်းတူပေမယ့် Port မတူလို့ Origin မတူတော့ပါဘူး။ Request တွေပေးပို့တာကို Browser က ခွင့်ပြုမှာ မဟုတ်တော့ပါဘူး။ `localhost` နဲ့ ဥပမာပေးပေမယ့် လက်တွေ့မှာလည်း အတူတူပါပဲ။ `domain-a.com` ကနေ `domain-b.com` ကို Request တွေပေးပို့ဖို့ ကြိုးစားတဲ့အခါ Browser ကလက်ခံမှာမဟုတ်ပါဘူး။ စမ်းသပ်ချင်ရင် HTML Document တစ်ခုတည်ဆောက်ပြီး ဒီကုဒ်ကို ရေးစမ်းကြည့်ပါ။

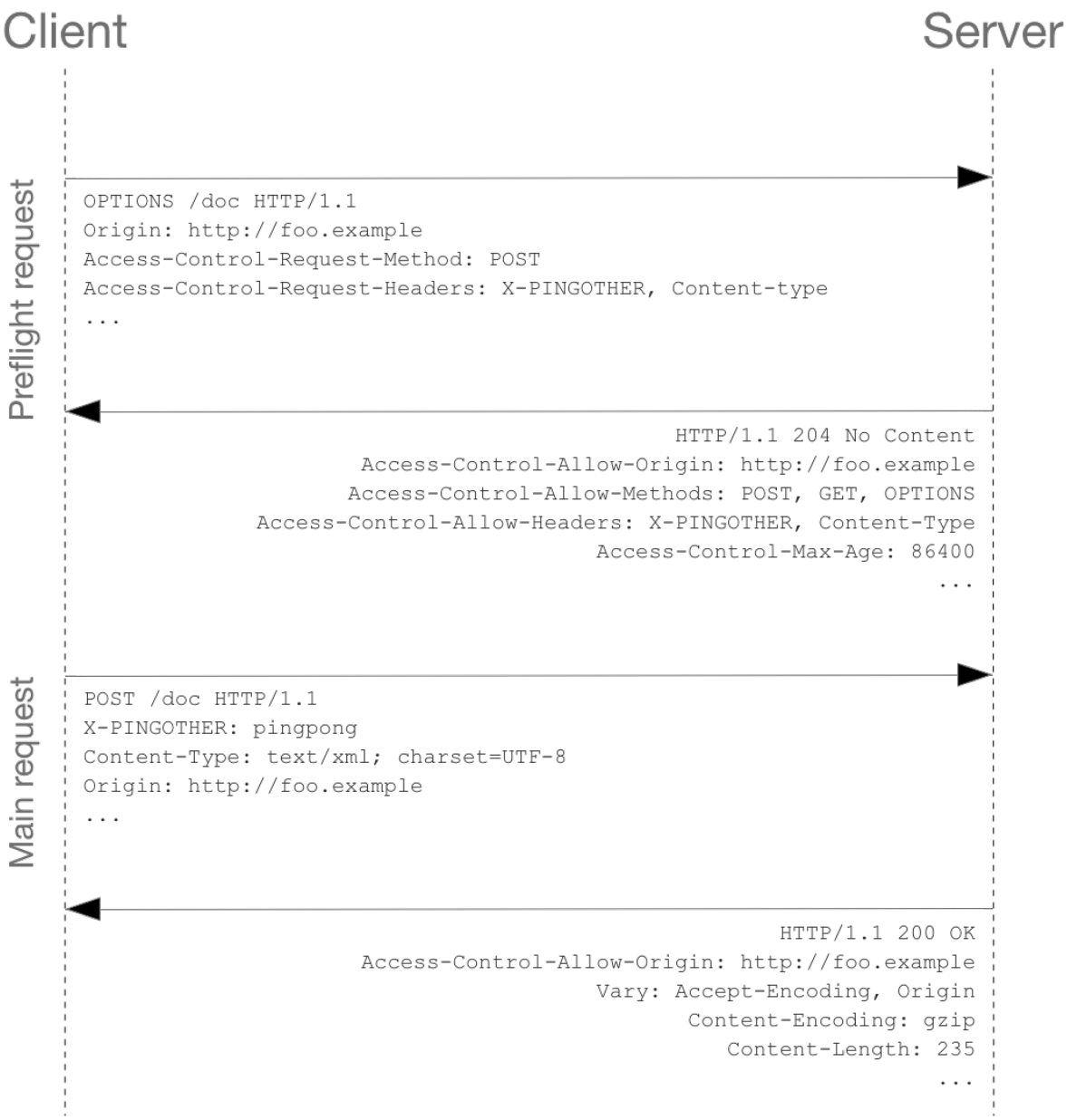
```
<script>
    fetch("http://localhost:8000/api/records")
        .then(function(res) {
            return res.json();
        })
        .then(function(json) {
            console.log(json);
        });
</script>
```

ကျွန်တော်တို့ရဲ့ API ကို JavaScript ရဲ့ `fetch()` Function သုံးပြီး Request ပေးပို့လိုက်တာပါ။ စမ်းကြည့်နိုင်ဖို့ API Server Run ထားပေးဖို့တော့ လိုပါတယ်။ ရေးထားတဲ့ ကုဒ်အရ Server က ပြန်ပေးတဲ့ Response ကို JSON ပြောင်းပြီး Console မှာ ရိုက်ထုတ်ခိုင်းလိုက်ပါတယ်။ ဒီကုဒ်ကို Browser မှာ စမ်းကြည့်ရင် အခုလို Error ကို ရရှိမှာဖြစ်ပါတယ်။



ဒါဟာ Browser က Origin မတူတဲ့အတွက် CORS နဲ့ ညီနေလို့ ပေးတဲ့ Error ပါ။

CORS နဲ့ ညီနေလို့ဆိုပြီး Request တွေ လုံးဝမပို့တာတော့ မဟုတ်ပါဘူး။ `OPTIONS` Method ကိုသုံးပြီး Origin မတူတာကို လက်ခံလိုခြင်း ရှိမရှိ Server ကို လှမ်းတော့ မေးပေးပါတယ်။ Browser က အလို အလျောက် မေးပေးတာပါ။ ကိုယ်ဘက်က အဲ့ဒီလိုမေးပေးဖို့ သတ်မှတ်ပေးစရာ မလိုပါဘူး။ CORS ရဲ့ အလုပ်လုပ်ပုံ အဆင့်ဆင့်ကို ဆက်လက်ဖော်ပြတဲ့ပုံမှာ လေ့လာကြည့်ပါ (ပုံ – MDN)။



ပထမဆုံး **Browser** က `OPTIONS` **Method** ကိုသုံးပြီး **Request** ပို့ပါတယ်။ မူလ `Origin` က `foo.example` ဖြစ်ပြီး ပေးပို့လိုတဲ့ **Method** က `POST` ဖြစ်ကြောင်း `Access-Control-Request-Method` **Header** နဲ့ပြောပြပါတယ်။ ပေးပို့လိုတဲ့ `Headers` စာရင်းကိုလည်း ထည့်ပြောပြပါတယ်။

Server က ခွင့်မပြုဘူးဆိုတော့မှ စောစောက Error ကို တွေ့ရတာပါ။ ခွင့်ပြုတယ်ဆိုရင် Server က `Access-Control-Allow-*` နဲ့ စတဲ့ Headers တွေကို ပြန်ပို့ပေးပါတယ်။ ခွင့်ပြုတဲ့ Origin တွေ၊ ခွင့်ပြုတဲ့ Methods တွေ၊ ခွင့်ပြုတဲ့ Headers တွေကို စာရင်းနဲ့ ပြန်ပို့တာပါ။ ဒီလို Server က ခွင့်ပြုတယ်ဆိုတော့မှ Browser က မူလပေးပို့လိုတဲ့ Request တွေကို ဆက်ပြီးပေးပို့ပေးသွားမှာပဲ ဖြစ်ပါတယ်။ ဒီသဘောသဘာဝကို Preflight Request လို့ ခေါ်ပါတယ်။ ဒါဟာ CORS ရဲ့ အလုပ်လုပ်ပုံ အနှစ်ချုပ်ပါပဲ။

ဒီတော့ ကျွန်တော်တို့ API ဘက်က CORS Request တွေကို ခွင့်ပြုမှာလား စဉ်းစားစရာရှိလာပါပြီ။ ခွင့်ပြုမယ်ဆိုရင်တော့ အခုလို ရေးသားသတ်မှတ်ပြီး ခွင့်ပြုပေးနိုင်ပါတယ်။

```
app.get("/api/records", function(req, res) {
    res.append("Access-Control-Allow-Origin", "*");
    res.append("Access-Control-Allow-Methods", "*");
    res.append("Access-Control-Allow-Headers", "*");
    …
});
```

Headers (၃) ခု ထည့်ပေးလိုက်တာပါ။ `Access-Control-Allow-Origin` နဲ့ ခွင့်ပြုလိုတဲ့ Host တွေကို သတ်မှတ်ပေးနိုင်ပါတယ်။ နမူနာမှာ * ကိုပေးထားလို့ Origin ဘယ်ကလာလာ အကုန် ခွင့်ပြုတယ်ဆိုတဲ့ အဓိပ္ပါယ်ရပါတယ်။ `http://a.com, http://b.com` စသဖြင့် ခွင့်ပြုလိုတဲ့ Host တွေတန်းစီပြီး ပေးထားလို့လည်းရပါတယ်။ `Access-Control-Allow-Methods` ကတော့ ခွင့်ပြုလိုတဲ့ Methods စာရင်းအတွက်ပါ။ အတူတူပါပဲ၊ တစ်ခုချင်းပေးချင်ရင် `GET, HEAD, POST` စသဖြင့် တန်းစီပြီးပေးထားလို့ရပါတယ်။ အကုန်ပေးချင်ရင်တော့ နမူနာမှာလို * ကိုပေးလိုက်ရင် ရပါတယ်။ ဒီလောက်ဆိုသဘောပေါက်မယ်ထင်ပါတယ်။ Headers လည်း ထိုနည်းလည်းကောင်း အတူတူပါပဲ။

ရေးထားတဲ့ကုဒ်အရ `/api/records` URL တစ်ခုတည်းအတွက်ပဲ CORS Headers တွေ သတ်မှတ်ထားတာပါ။ ဒါကြောင့် တစ်ခြား URL တွေအတွက် အလုပ်လုပ်မှာ မဟုတ်ပါဘူး။ အားလုံးအတွက် အလုပ်လုပ်စေချင်ရင် အခုလို ရေးလို့ရပါတယ်။

```
app.use(function(req, res, next) {
    res.append("Access-Control-Allow-Origin", "*");
    res.append("Access-Control-Allow-Methods", "*");
    res.append("Access-Control-Allow-Headers", "*");
    next();
});
```

use() ရဲ့အကူအညီနဲ့ Response အားလုံးအတွက် CORS Headers တွေသတ်မှတ်ပေးလိုက်တာပါ။ ဒါဟာ Middleware တစ်ခုဖြစ်လို့ next() ကိုသတိပြုပါ။ သူ့ရဲ့အဓိပ္ပါယ်က ဒီ Middleware ကိုအလုပ်လုပ်ပြီးရင် ရှေ့ဆက်ပြီး လုပ်စရာရှိတာ လုပ်သွားစေဖို့ဖြစ်ပါတယ်။ ဒီလိုကိုယ့်ဘာသာ မရေးချင်ဘူးဆိုရင်လည်း cors လို့ခေါ်တဲ့ Package တစ်ခုရှိပါတယ်။ install လုပ်ပြီး သုံးလို့ရပါတယ်။

```
npm i cors
```

ရေးပုံရေးနည်းကရှင်းပါတယ် ဒီလိုပါ။

```
const cors = require("cors");
app.use(cors());
```

ဒါပါပဲ။ ဒါဆိုရင် စောစောက ကျွန်တော်တို့ Manual ကိုယ့်ဘာသာ သတ်မှတ်ပေးလိုက်ရတဲ့ CORS Headers တွေ ပေးစရာမလိုတော့ပါဘူး။ အလိုအလျောက် ပါဝင်သွားမှာ ဖြစ်ပါတယ်။ Host တွေ Method တွေ အကုန်လက်ခံချင်တာမဟုတ်ဘူး၊ ရွေးပြီးလက်ခံချင်တယ်ဆိုရင်လည်း Options တွေ အခုလိုပေးလို့ရပါတယ်။

```
app.use(cors({
    origin: ["http://a.com", "http://b.com"],
    methods: ["GET", "POST"],
    allowHeaders: ["Authorization", "Content-Type"]
}));
```

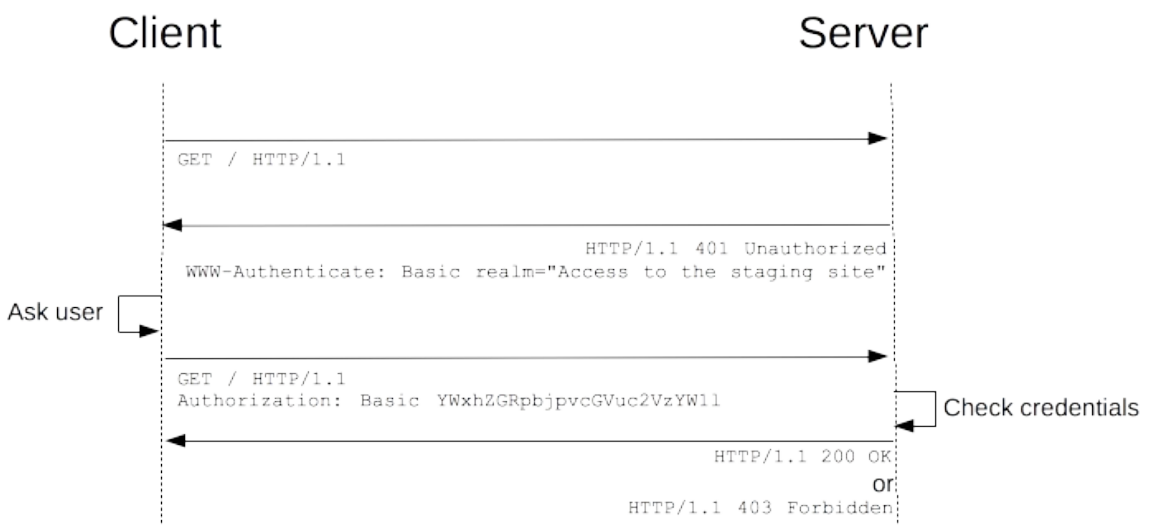
ဒါကြောင့် ရေးနည်းကမခက်ဘူးလို့ ဆိုနိုင်ပါတယ်။ နည်းပညာရဲ့ သဘောကို သိထားဖို့သာ လိုအပ်တာပါ။ ဒီလောက်ဆိုရင် CORS ရဲ့ သဘောသဘာဝ လိုရင်းအနှစ်ချုပ်ကို သဘောပေါက်မယ်လို့ ယူဆပါတယ်။

အခန်း (၁၀) – Auth

API မှာ ဝင်ခွင့်ပြုမပြုစီစစ်ခြင်း၊ လုပ်ခွင့်ပြုမပြုစီစစ်ခြင်း စတဲ့ Authentication/Authorization နဲ့ ပက်သက်ရင် အသုံးများတဲ့ နည်းလမ်းအနေနဲ့ ဒီလိုမျိုး (၅) မျိုး ရှိပါတယ်။

1. HTTP Basic
2. Session
3. Token
4. JWT
5. OAuth2

HTTP Basic Authentication ရဲ့အလုပ်လုပ်ပုံက ဒီလိုပါ (ပုံ – MDN)။



Client က Resource တစ်ခုကို Request လုပ်တဲ့အခါ Server က `401` Unauthorized ကို ပြန်ပေးပါတယ်။ တစ်လက်စတည်း `WWW-Authenticate` Header နဲ့အတူ အသုံးပြုရမယ့် Authentication Method ကို အကြောင်းပြန်ပါတယ်။ နမူနာပုံအရ `Basic` Authentication ကို အသုံးပြုရမယ်လို့ Server က ပြောနေတာ ဖြစ်တဲ့အတွက် Client က နောက်တစ်ကြိမ်မှာ `Authorization` Header နဲ့အတူ Username, Password ကို Base64 Encode နဲ့ Encode လုပ်ပြီး ပြန်ပို့ပာထားခြင်း ဖြစ်ပါတယ်။ ဒီနည်းကတော့ ရှင်းပါတယ်။ Client ကလိုအပ်တဲ့ Username, Password ကို Header မှာ ထည့်ပို့ခြင်းဖြစ်ပြီး Request ပြုလုပ်တဲ့ အကြိမ်တိုင်းမှာ ထည့်ပို့ပေးဖို့ လိုအပ်ပါတယ်။

Session Authentication ကတော့ API မှာ သုံးလေ့ သိပ်မရှိကြပါဘူး။ သုံးလို့မရတာ မဟုတ်ပါဘူး။ ရပါတယ်။ ဒါပေမယ့် REST ရဲ့ မူသဘောအရ Stateless ဖြစ်ရမယ်ဆိုတဲ့ သတ်မှတ်ချက် ရှိထားတဲ့အတွက် Session က ဒီမူနဲ့ မကိုက်လို့ပါ။ Session Authentication ရဲ့ အလုပ်လုပ်ပုံကတော့ ဒီလိုပါ။

1. ပထမတစ်ကြိမ် Client က Username, Password ကို Request နဲ့အတူ ပေးရပါမယ်။
2. Server က စစ်ပြီး မှန်တယ်ဆိုရင် User နဲ့ သက်ဆိုင်တဲ့ အချက်အလက်တွေကို Session ထဲမှာ သိမ်းလိုက်ပါတယ်။
3. Server က Session ID ကို Response နဲ့အတူ ပြန်ပို့ပေးပါတယ်။
4. Client က လက်ခံရရှိတဲ့ Session ID ကို Cookie ထဲမှာ သိမ်းပါတယ်။
5. နောက်ပိုင်းမှာ Username, Password ထပ်ပေးစရာမလိုတော့ပါဘူး။ Cookie ထဲမှာသိမ်းထားတဲ့ Session ID ကိုပဲ ပြန်ပို့ရတော့မှာပါ။ Session ID နဲ့စစ်ကြည့်လိုက်လို့ Session ထဲမှာ User ရဲ့ အချက်အလက်တွေ ရှိနေသ၍ Authenticate ဖြစ်တယ်လို့ လက်ခံပြီး Server က အလုပ်လုပ်ပေးသွားမှာ မို့လို့ပါ။

Token Authentication ကိုတော့ API မှာ ကျယ်ကျယ်ပြန့်ပြန့်သုံးကြပါတယ်။ Stateless ဖြစ်တဲ့အတွက် ကြောင့်ပါ။ Cookie တွေ Session တွေ မလိုအပ်ပါဘူး။ သူ့ရဲ့အလုပ်လုပ်ပုံကဒီလိုပါ။

- ပထမတစ်ကြိမ် Client က Username, Password ကို Request နဲ့အတူ ပေးရပါတယ်။
- Server က စစ်ပြီး မှန်တယ်ဆိုရင် Token တစ်ခု Generate လုပ်ပြီး Response ပြန်ပေးပါတယ်။
- Token ကို User Table ထဲမှာလည်း သိမ်းထားကောင်း ထားလိုက်နိုင်လိုက်ပါတယ်။
- နောက်ပိုင်းမှာ Client က Username, Password ပေးစရာမလိုတော့ပါဘူး၊ ရထားတဲ့ Token ကို ပဲပြန်ပေးရတော့မှာပါ။ Server က Token ကိုစစ်ကြည့်ပြီး မှန်ကန်တယ်ဆိုရင် Authenticate Request အဖြစ် လက်ခံအလုပ်လုပ်ပေးမှာပါ။

JWT ကလည်း Token Authentication တစ်မျိုးပါပဲ။ **JSON Web Token** ရဲ့ အတိုကောက် ဖြစ်ပါတယ်။ ရိုးရိုး Token Authentication မှာ Token က Random Hash Value တစ်ခုဖြစ်လေ့ရှိပါတယ်။ အဲ့ဒီ Token ထဲမှာအသုံးဝင်တဲ့ အချက်အလက် မပါပါဘူး။

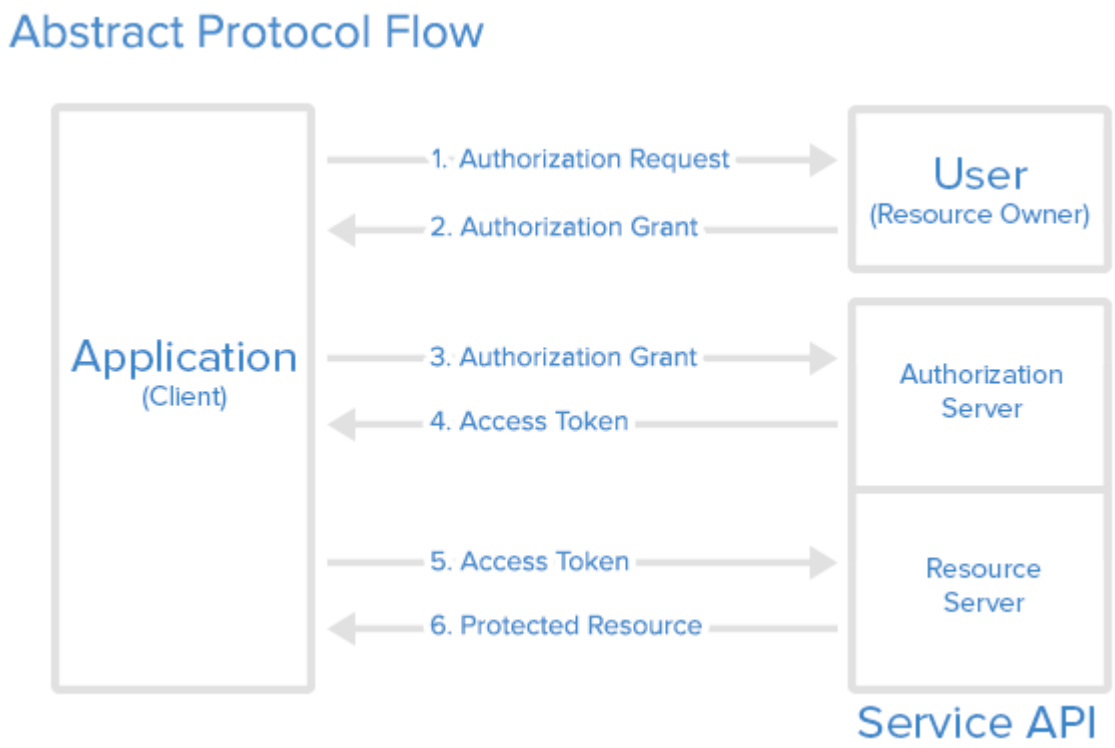
JWT ကတော့ အဲ့ဒီလို Random Token မဟုတ်တော့ပါဘူး။ User Information တွေကို Encrypt လုပ်ထားတဲ့ Token ဖြစ်သွားတာပါ။ ဒါကြောင့် User Information လိုချင်ရင် Token ကို Decrypt လုပ်ပြီး ပြန်ထုတ်ယူလို့ရပါတယ်။ Token ထဲမှာ အသုံးဝင်တဲ့ အချက်အလက်တွေ ပါသွားတဲ့ သဘောပါ။ ခဏနေတဲ့ အခါ JWT ကိုသုံးပြီး ကုဒ်နမူနာတွေ ရေးပြပါမယ်။

OAuth2 ကတော့ ရှုပ်ပါတယ်။ နည်းနည်းမဟုတ်ပါဘူး တော်တော်ရှုပ်တာပါ။ အတက်နိုင်ဆုံး ကြိုးစားပြီးတော့ ရှင်းအောင် ပြောကြည့်ပါမယ်။ OAuth မှာ အစိတ်အပိုင်း (၄) ပိုင်းပါတယ်လို့ မှတ်ပါ။ မျက်စိထဲ မြင်လွယ်အောင် Facebook နဲ့ ဥပမာပေးချင်ပါတယ်။ ဒီလိုပါ။

1. User (သင်)
2. Client Application (Facebook နဲ့ Login ဝင်ရတဲ့ App)
3. Resource Server (Facebook)
4. Authorization Server (Facebook Developer API)

ဒီသဘောနဲ့ အလုပ်လုပ်ပုံကို တွေ့ဖူးကြပါလိမ့်မယ်။ App တစ်ခုကို သုံးချင်လို့ဖွင့်လိုက်တယ်။ **Login with Facebook** ဆိုတဲ့လုပ်ဆောင်ချက်ပါတယ်။ ဒါကြောင့် အဲ့ဒီ App ကိုသုံးဖို့ Facebook နဲ့ Login ဝင်လို့ရမယ်။ နှိပ်လိုက်တယ်။ Dialog Box ပေါ်လာပြီး Facebook နဲ့ Login ဝင်တာကို Accept လုပ်မှာလား‌လို့ Facebook Developer API ကလာမေးတယ်။ Accept လုပ်ပေးလိုက်ရင် အဲ့ဒီ App ကို ကိုယ့် Facebook Account နဲ့ ဝင်သုံးလို့ ရသွားပါပြီ။

ဒီလိုပုံစံ အလုပ်လုပ်နိုင်စေဖို့အတွက် OAuth ကို အသုံးပြုရတာပါ။ သေချာစဉ်းစားကြည့်ပါ။ ပုံမှန်ဆိုရင် ကိုယ့် API က ပေးထားတဲ့ Auth နဲ့ ကိုယ့် API ကိုသုံးရတာပါ။ အခုက ကိုယ့် API က ပေးထားတဲ့ Auth နဲ့ တစ်ခြား App မှာ သွားပြီးသုံးလို့ ရနေတာပါ။ သူ့ရဲ့အလုပ်လုပ်ပုံကို အောက်က ပုံမှာလေ့လာကြည့်ပါ (ပုံ – Digital Ocean)။



1, 2, 3, 4 နံပါတ်စဉ်တပ်ပေးလို့ အစီအစဉ်အတိုင်း ကြည့်သွားလို့ ရပါတယ်။

1. ပထမဆုံးအနေနဲ့၊ Facebook နဲ့ Login ဝင်ဖို့ User က ခွင့်ပြုမပြုမေးရပါတယ်။
2. User က Allow လုပ်ပြီး ခွင့်ပြုလိုက်တဲ့အခါ Authorization Code ထွက်လာပါတယ်။
3. App က Authorization Code ကိုသုံးပြီး Facebook Developer API ကို User ရဲ့ အချက်အလက်တွေ Access လုပ်ခွင့် တောင်းပါတယ်။

    a) App ကို Developer API မှာအရင် Register လုပ်ထားဖို့လည်း လိုပါသေးတယ်။ ဒီတော့ မှ Client ID တွေဘာတွေထက်လာမှာပါ။
    b) Client ID တွေဘာတွေ သေသေချာချာ ပြည့်စုံမှန်ကန်အောင်ပါမှ User ရဲ့ အချက်အလက်ကို Third-party ဘယ် App က ယူသလဲဆိုတဲ့ မှတ်တမ်းကိုရမှာမို့လို့ပါ။

4. Authorization Code, Client ID နဲ့ အချက်အလက် ပြည့်စုံမှန်ကန်တယ်ဆိုရင် Developer API က Access Token ပြန်ထုတ်ပေးပါတယ်။
5. App က လိုချင်တဲ့ User ရဲ့အချက်အလက်ကို အခုဒီ Access Token ကိုသုံးပြီး ရယူလို့ရသွားပါပြီ။

ကိုယ့် Service က Facebook လို Resource Server ဖြစ်နိုင်သလို၊ Facebook Developer API လို Authorization Server လည်း ဖြစ်နိုင်ပါတယ်။ ဒါဟာ OAuth ရဲ့ အလုပ်လုပ်ပုံ အကျဉ်းချုပ်ပါပဲ။ အကျယ်ပြောမယ်ဆိုရင် သူ့ချည်းပဲ စာတစ်အုပ်စာ ရှိပါလိမ့်မယ်။ ဒါကြောင့် သဘောသဘာဝ ပိုင်းလောက်ပဲ မှတ်ထားပေးပါ။ OAuth အကြောင်း Digital Ocean မှာဖော်ပြထားတဲ့ ဆောင်းပါးတစ်ပုဒ်ကို ဖြည့်စွက်လေ့လာကြည့်ဖို့ အကြံပြုပါတယ်။

– https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2

JWT Authentication & Authorization

JWT ရဲ့ သဘောသဘာဝကို အပေါ်မှာ ပြောခဲ့ပါတယ်။ Token ကို အခြေခံတဲ့ Authentication ဖြစ်ပြီးတော့ Token ထဲမှာ အသုံးဝင်တဲ့ အချက်အလက်တွေ ပါဝင်မှာပါ။ ဒီအတွက် Token ဖန်တီးတာတွေ၊ မှန်မမှန် ပြန်စစ်တာတွေ၊ Token ကို Encrypt/Decrypt လုပ်တာတွေ၊ အကုန်လုပ်ပေးနိုင်တဲ့ Package တစ်ခုရှိပါတယ်။ `jsonwebtoken` လို့ခေါ်ပါတယ်။ စမ်းသပ်နိုင်ဖို့အတွက် ရေးသားလက်စ ပရောဂျက်ထဲမှာ အခုလို Install လုပ်လိုက်ပါ။

```
npm i jsonwebtoken
```

ပြီးတဲ့အခါ၊ ထုံးစံအတိုင်း Import လုပ်ပေးလိုက်ရင် စသုံးလို့ရပါပြီ။

```
const jwt = require("jsonwebtoken");
const secret = "horse battery staple";
```

နမူနာကုဒ်မှာတွေ့ရတဲ့ `secret` ဆိုတာကတော့ Token တွေကို Encrypt/Decrypt လုပ်ရာမှာ သုံးမယ့် Code ဖြစ်ပါတယ်။ Auth လုပ်ငန်းတွေ စမ်းသပ်ရေးသားနိုင်ဖို့အတွက် User Account တစ်ချို့ လိုပါမယ်။ Database တွေဘာတွေမသုံးတော့ပါဘူး။ ရိုးရိုး JSON Array တစ်ခုနဲ့ပဲစမ်းကြည့်ကြပါမယ်။ ဒီလိုပါ -

```
const users = [
    { username: "Alice", password: "password", role: "admin" },
    { username: "Bob", password: "password", role: "user" },
];
```

User Account နှစ်ခုရှိပါတယ်။ `role` မတူကြပါဘူး။ တစ်ဦးက `admin` ဖြစ်ပြီး နောက်တစ်ဦးကတော့ `user` ဖြစ်ပါတယ်။ ပြီးတဲ့အခါ `login` လုပ်ဆောင်ချက်တစ်ခုကို အခုလိုရေးကြပါမယ်။

```
app.post("/api/login", function(req, res) {

    const { username, password } = req.body;

    const user = users.find(function(u) {
        return u.username === username && u.password === password;
    });

    if(auth) {
        jwt.sign(user, secret, {
            expiresIn: "1h"
        }, function(err, token) {
            return res.status(200).json({ token });
        });
    } else {
        return res.sendStatus(401);
    }
});
```

Request Method `POST` ဖြစ်ရမှာဖြစ်ပြီး URL က `/api/login` ဖြစ်ပါတယ်။ Request နဲ့အတူ မှန်ကန်တဲ့ Username, Password ပါရမှာဖြစ်ပြီး၊ မှန်တယ်ဆိုရင် JWT Token တစ်ခုကို ပြန်ပေးမှာပါ။ မမှန်ရင် `401` ကို ပြန်ပို့မှာပါ။ `jwt.sign()` ကိုသုံးပြီး Token ဖန်တီးယူပါတယ်။ Parameter (၄) ခု ပေးထားပါတယ်။ User Data, Secret, Expire Time နဲ့ Callback Function တို့ဖြစ်ပါတယ်။ Request က ဒီလိုပုံစံ ဝင်လာတယ်လို့ သဘောထားပါ။

Request

```
POST /api/login
Content-type: appliction/json
{ username: "Bob", password: "password" }
```

ဒါဆိုရင် ပြန်ရမယ့် Token ရဲ့ ဖွဲ့စည်းပုံက ဒီလိုပုံစံ ဖြစ်နိုင်ပါတယ်။ Postman နဲ့ စမ်းကြည့်နိုင်ပါတယ်။

Token

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6IkJvYiIsInBhc3
N3b3JkIjoicGFzc3dvcmQiLCJyb2xlIjoidXNlciIsImlhdCI6MTYwMDc2MzI1NiwiZ
XhwIjoxNjAwNzY2ODU2fQ.-OBn8nIEmJqdNc9XfoUVVcZc7PEVUWHVQOP85YIlygo
```

Token တည်ဆောက်စဉ်မှာ Expire Time ပေးခဲ့တာကို သတိပြုပါ။ ဒီ Token ဟာ (၁) နာရီသက်တမ်းအတွင်းပဲ Valid ဖြစ်မှာပါ။ (၁) နာရီကျော်ရင် နောက်တစ်ခါ `login` လုပ်ပြီး Token ပြန်ထုတ်ရမှာဖြစ်ပါတယ်။ Token မှန်မမှန်စစ်တဲ့ Function တစ်ခုလောက် ဆက်ရေးကြပါမယ်။

```
function auth(req, res, next) {
    const authHeader = req.headers["authorization"];
    if(!authHeader) return res.sendStatus(401);

    const [ type, token ] = authHeader.split(" ");

    if(type !== "Bearer") return res.sendStatus(401);

    jwt.verify(token, secret, function(err, data) {
        if(err) return res.sendStatus(401);
        else next();
    });
}
```

`Authorization` Header ပါမပါ စစ်ပါတယ်။ မပါရင် 401 ပြန်ပို့ပါတယ်။ JWT ရဲ့ Standard အရ `Authorization` Header ရဲ့ ဖွဲ့စည်းပုံ ဒီလိုဖြစ်ရပါတယ်။

```
Authorization: Bearer [token]
```

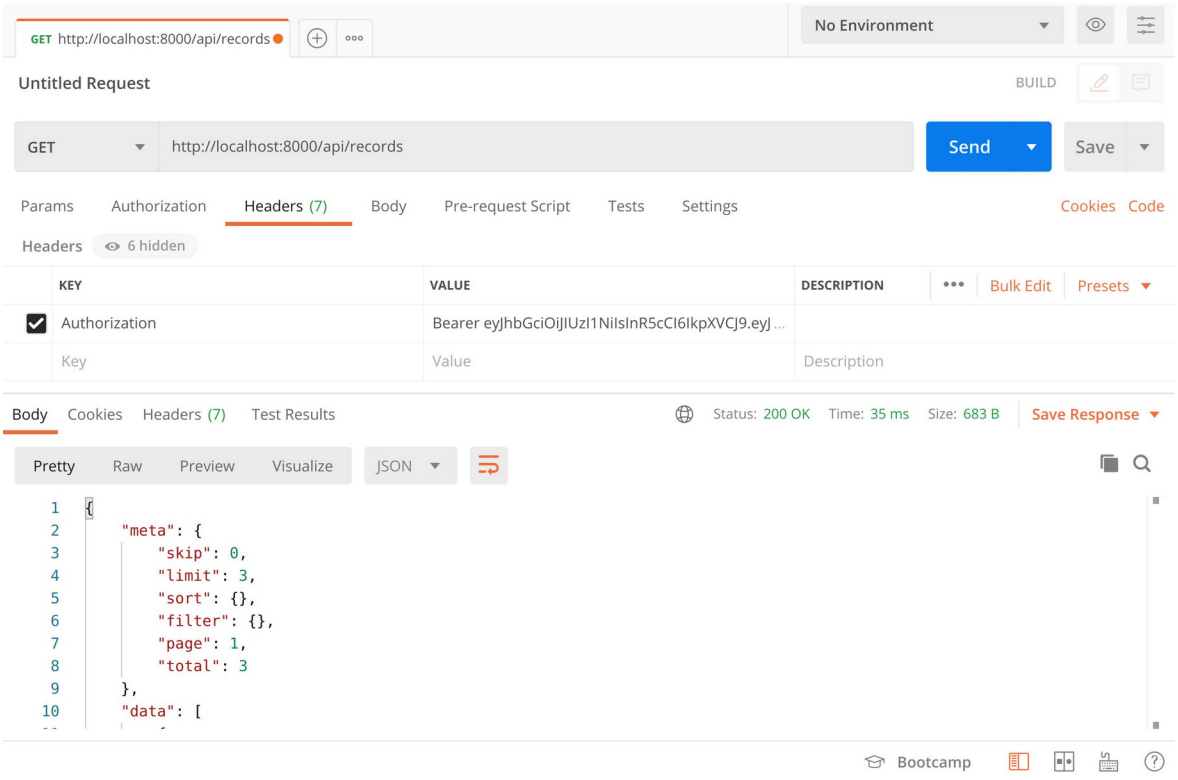
ဒါကြောင့် `Authorization` Header Value ကို Split လုပ်ပြီး နှစ်ပိုင်းခွဲလိုက်ပါတယ်။ ပထမတစ်ပိုင်းက Bearer ဖြစ်ပြီး နောက်တစ်ပိုင်းက Token ဖြစ်ရပါမယ်။ Token ကို `verify()` နဲ့ မှန်မမှန်စစ်ပါတယ်။ မှန်တယ်ဆိုတော့မှ `next()` နဲ့ ဆက်အလုပ်လုပ်ခွင့်ကို ပေးထားခြင်း ဖြစ်ပါတယ်။

ဒီ Function က Middleware Function တစ်ခုဖြစ်ပါတယ်။ ဒါပေမယ့် `app.use()` နဲ့ Route အားလုံးမှာ သုံးဖို့ သတ်မှတ်မထားပါဘူး။ သတ်မှတ်လို့မဖြစ်ပါဘူး။ `login` လိုလုပ်ဆောင်ချက်မျိုးကို Token ပါရမယ်လို့ သွားပြောလို့ မဖြစ်ပါဘူး။ Token မရှိလို့ဘဲ `login` နဲ့ Token ထုတ်နေတာပါ။ ဒါကြောင့် ကိုယ် သတ်မှတ်ချင်တဲ့ Route မှာပဲ အခုလို သတ်မှတ်ပေးလိုက်လို့ ရပါတယ်။

```
app.get("/api/records", auth, function(req, res){
    …
});
```

ဒီသတ်မှတ်ချက်အရ `/api/records` ကို Request ဝင်လာတဲ့အခါ စောစောကရေးပေးထားတဲ့ **auth** Middleware ကိုသုံးပြီး စစ်ပေးသွားမှာပါ။ Token မပါရင် 401 ကိုပြန်ပေးမှာဖြစ်ပြီး Token မမှန်ရင်လည်း 401 ကိုပဲ ပြန်ပေးသွားမှာပါ။ အလုပ်လုပ်ခွင့်ပေးမှာ မဟုတ်ပါဘူး။ Token မှန်မှသာ ဆက်အလုပ်လုပ်ခွင့်ပေးမှာပဲ ဖြစ်ပါတယ်။ ဒီနည်းနဲ့ ကိုယ့် API အတွက် Authentication လုပ်ဆောင်ချက် ထည့်သွင်းနိုင်ခြင်း ဖြစ်ပါတယ်။

စမ်းကြည့်နိုင်ဖို့အတွက် အရင်ဆုံး `login` လုပ်လိုက်ပါ။ ရလာတဲ့ Token ကိုသုံးပြီး ဆက်လက်ဖော်ပြထားတဲ့ပုံမှာ နမူနာပြထားသလို စမ်းကြည့်နိုင်ပါတယ်။



ဒါက Authentication ပိုင်းပါ။ Token ပါတယ်၊ မှန်တယ်ဆိုရင် ခွင့်ပြုလိုက်တာပါ။ `role` ပေါ်မူတည်ပြီး သက်ဆိုင်ရာအလုပ်ကို လုပ်ပိုင်ခွင့်ရှိမရှိ စစ်တဲ့တဲ့ Authorization ပိုင်းကိုလည်း ဆက်ကြည့်ပါဦးမယ်။ နောက်ထပ် Function တစ်ခုအခုလို ထပ်ရေးပေးရမှာပါ။

```
function onlyAdmin(req, res, next) {

    const [ type, token ] = req.headers["authorization"].split(" ");

    jwt.verify(token, secret, function(err, user) {
        if(user.role === "admin") next();
        else return res.sendStatus(403);
    });
}
```

Token ကို Decrypt လုပ်လိုက်ပြီး အထဲက `role` တန်ဖိုးကိုပဲ စစ်လိုက်တာပါ။ `role` က `admin` ဖြစ်မှပဲ ဆက်လုပ်ခွင့်ပေးပြီး `admin` မဟုတ်ရင် `403` ကို ပြန်ပေးထားပါတယ်။ ဒီ Middleware ကို Admin ဖြစ်မှ လုပ်ခွင့်ပြုချင်တဲ့ Route တွေမှာ အခုလို ထည့်ပေးနိုင်ပါတယ်။

```
app.delete("/api/records/:id", auth, onlyAdmin, function(req, res) {
    …
});
```

နမူနာအရ `DELETE` လုပ်ဆောင်ချက်အတွက် `auth` ရော `onlyAdmin` ကိုပါ Middleware တွေအဖြစ် သတ်မှတ်ပေးလိုက်တာပါ။ ဒါကြောင့် Auth ဖြစ်ယုံနဲ့တောင် ဒီအလုပ်ကို လုပ်လို့မရတော့ပါဘူး။ `role` က `admin` ဖြစ်မှပဲ လုပ်ခွင့်ရှိတော့မှာဖြစ်ပါတယ်။

ဒီနည်းနဲ့ JWT ကို သုံးပြီး API အတွက် Authentication တွေ Authorization တွေ လုပ်လို့ရနိုင်ပါတယ်။ ဒီကုဒ်ဟာ အခြေခံသဘောသဘာဝကို ပေါ်လွင်စေဖို့ ဦးစားပေး ဖော်ပြတဲ့ကုဒ်ဖြစ်ပါတယ်။ တစ်ကယ့်လက်တွေ့မှာ -

- User Account တွေကို Database ထဲမှာထားပြီး Register တွေဘာတွေလုပ်လို့ရဖို့လိုပါမယ်။
- Password တွေကို ဒီအတိုင်းမသိမ်းဘဲ Hash လုပ်ပြီး သိမ်းဖို့လိုပါမယ်။
- User နဲ့ Role တွေကို စီမံနိုင်တဲ့ လုပ်ငန်းတွေ ထည့်ရေးပေးရပါမယ်။
- Secret ကို ကုဒ်ထဲမှာ အသေမရေးဘဲ `.env` လိုဖိုင်မျိုးနဲ့ ခွဲထားပြီးခေါ်သုံးပေးဖို့ လိုပါမယ်။

မပြောဖြစ်လိုက်တာမျိုး ဖြစ်မှာစိုးလို့သာ ထည့်ပြောတာပါ။ ဒီနေရာမှာတော့ အဲ့ဒီထိပြီးပြည့်စုံအောင် ဖော်ပြနိုင်ခြင်း မရှိပါဘူး။ ဒီစာအုပ်မှာ ဖော်ပြခဲ့တဲ့ အခြေခံသဘောသဘာဝတွေကို ကောင်းကောင်နားလည်တယ်ဆိုရင် ဒါတွေကို ကိုယ်တိုင်ဆက်လက် လေ့လာပြီးလုပ်လို့ရသွားမှာပါ။

API Authentication နဲ့ပက်သက်ရင် PassportJS လို Framework မျိုးတွေလည်းရှိပါသေးတယ်။ လူကြိုက်များပြီး လက်တွေ့ပရောဂျက်တွေမှာ တွင်တွင်ကျယ်ကျယ် အသုံးပြုကြပါတယ်။ ဒီလိုနည်းပညာမျိုးကိုလည်း ဆက်လက်ပြီး ဖြည့်စွက်လေ့လာထားကြဖို့ တိုက်တွန်းပါတယ်။

- http://www.passportjs.org/

အခုဆိုရင် ဒီစာအုပ်မှာဖော်ပြချင်တဲ့ အကြောင်းအရာတွေ ပြည့်စုံသွားပါပြီ။ နှစ်ပိုင်းရှိတယ်လို့ ဆိုနိုင်ပါတယ်။ ပထမတစ်ပိုင်းက API ဒီဇိုင်းနဲ့ ပက်သက်ပြီး သိသင့်တဲ့အကြောင်းအရာတွေပါ။ Request, Response တွေရဲ့သဘောသဘာဝတွေ၊ RESTful URL နဲ့ Response Structure အကြောင်းတွေ လေ့လာခဲ့ကြပါတယ်။ နောက်တစ်ပိုင်းကတော့ Mongo, Node, Express ကိုအသုံးပြုပြီး လက်တွေ့နမူနာလေးတစ်ခု ရေးသားခဲ့ကြရာမှာ CORS တို့ Auth တို့လို အကြောင်းတွေကိုပါ ဖြည့်စွက်လေ့လာခဲ့ကြခြင်း ဖြစ်ပါတယ်။

နမူနာအနေနဲ့ ရေးခဲ့တဲ့ကုဒ်တွေကို ဒီမှာ Download လုပ်လို့ရပါတယ်။

– https://github.com/eimg/api-book

နောက်တစ်ခန်းမှာ ဆက်လက် လေ့လာသင့်တာလေးတွေကိုလည်း စုစည်းပြီးတော့ ပေးထားပါသေးတယ်။ ဒီစာအုပ်မှာ ဖော်ပြထားတာတွေကို အရင်နားလည်အောင် ကြိုးစားပါ။ နားလည်ပြီ၊ ရပြီဆိုရင် ဝိုပြည့်စုံသွားအောင် ဆက်လက်လေ့လာနိုင်ဖို့ပဲ ဖြစ်ပါတယ်။

အခန်း (၁၁) – What's Next

Best practices for a pragmatic RESTful API

– https://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api

Specification for building APIs in JSON

– https://jsonapi.org/

Stack Overflow Best practices for REST API design

– https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/

Microsoft REST API Guideline

– https://github.com/microsoft/api-guidelines/blob/vNext/Guidelines.md

Google JSON Style Guide

– https://google.github.io/styleguide/jsoncstyleguide.xml

REST in Practice (Book)

– http://restinpractice.com/

နိဒါန်း

ဒီစာအုပ်ဟာ မြန်မာနိုင်ငံမှာ COVID-19 ကပ်ရောဂါ Second Wave ဖြစ်ပွားနေချိန်၊ အိမ်တွင်းအောင်းပြီး ရေးဖြစ်ခဲ့တဲ့ စာအုပ်ပါ။ ပြီးခဲ့တဲ့လအချို့က First Wave ဖြစ်ပွားနေချိန်အတွင်းကလည်း ဒီလိုမျိုး လိုတိုရှင်း စာအုပ် (၂) အုပ်ရေးဖြစ်ခဲ့ပါသေးတယ်။ **React – လို – တို – ရှင်း** နဲ့ **Laravel – လို – တို – ရှင်း** လို့ခေါ်ပါတယ်။ ဒီ **API – လို – တို – ရှင်း** စာအုပ်ကို အဲ့ဒီစာအုပ်တွေနဲ့ တွဲပြီးဖတ်ရှုသင့်ပါတယ်။ React က Front-End ပိုင်းဖြစ်ပြီး API က Back-end ပိုင်းဖြစ်လို့ နှစ်မျိုးပေါင်းလိုက်ရင် ပြည့်စုံသွားတဲ့သဘောပါ။

Laravel – လို – တို – ရှင်း စာအုပ်မှာလည်း API အကြောင်းကို အခြေခံသဘော ထည့်ရေးပေးခဲ့ပါတယ်။ ဒါကြောင့် ဒီစာအုပ်မှာ ဖော်ပြထားတာတွေနဲ့ ပေါင်းစပ်လိုက်မယ်ဆိုရင်၊ Laravel နဲ့လည်း စနစ်ကျတဲ့ API တွေ ဖန်တီးနိုင်သွားမှာပါ။ ဒီစာအုပ်မှာ Express ကို အသုံးပြုပြီး ဖော်ပြခဲ့ပေမယ့် API တွေရဲ့ သဘော သဘာဝတွေကတော့ အတူတူပဲဖြစ်ပါတယ်။

ဒီစာအုပ်ကို မရေးခင်က စိတ်ဝင်စားတဲ့သူ နည်းမယ်လို့ ယူဆခဲ့ပေမယ့်၊ Pre-Order ဖွင့်လိုက်ချိန်မှာ နာရီပိုင်းအတွင်း သတ်မှတ်ပမာဏပြည့်သွားတဲ့အထိ ဝိုင်းဝန်းအားပေးခဲ့ကြလို့ အားလုံးကိုကျေးဇူးတင်ပါတယ်။ ဒီလိုအားပေးပံ့ပိုးမှုတွေကြောင့် ရေးရခက်တဲ့ ဒီစာအုပ်ကို ရေးရတာ ပိုပြီးတော့ အားရှိသွားတာပါ။ ဒီစာအုပ်ကနေ စာဖတ်သူများတွေအတွက် မှတ်သားဖွယ်ရာ အသိပညာဗဟုသုတတွေ နဲ့ အသုံးဝင်တဲ့ အတတ်ပညာတွေကို ရရှိလိုက်မယ်လို့ မျှော်လင့်မိပါတယ်။ စာဖတ်သူများအားလုံး ကပ်ဘေးတွေကို ကျော်လွှားနိုင်ပြီး ကိုယ်စိတ်နှစ်ဖြာ ကျန်းမာချမ်းသာ ကြပါစေလို့ ဆုတောင်းလိုက်ပါတယ်။

အိမောင် (Fairway)

၂၀၂၀ ပြည့်နှစ်၊ စက်တင်ဘာ (၂၄) ရက်နေ့တွင် ရေးသားပြီးစီးသည်။