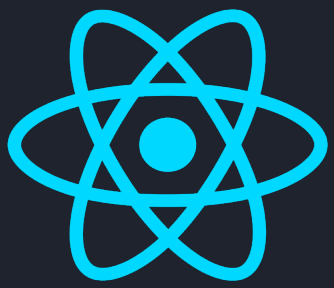


React

လို - တို - ရှင်း

အိမောင်

Fairway

© Copyright 2020, **Ei Maung**

Fairway Technology. All right reserved.

မာတိကာ

မိတ်ဆက်

React ဟာ ကနေ့အချိန်မှာ အရမ်းဟော့နေတဲ့ Front-end နည်းပညာတစ်ခုပါ။ တစ်ကယ်တော့ Front-end နည်းပညာ အနေနဲ့တင် မကပါဘူး၊ Hybrid Mobile App နည်းပညာ အနေနဲ့ရော၊ Cross-platform Software Development နည်းပညာအနေနဲ့ပါ လူကြိုက်များနေပါတယ်။ Front-end တို့ Hybrid တို့ Cross-platform တို့လို့ စကားလုံးတွေကြောင့် ခေါင်းစားသွားရင် သိပ်စိတ်မပူပါနဲ့။ React ဟာ သူသဘာဝ အရကိုက နည်းပညာပိုင်း အဆင့်မြင့် ရှုပ်ထွေးပြီး လေ့လာရ ခက်ခဲပါတယ်။ ဒီစာအုပ်မှာ အဲ့ဒီလို လေ့လာရ ခက်ခဲတဲ့ နည်းပညာကို အတတ်နိုင်ဆုံး လွယ်သွားအောင်၊ ရှင်းသွားအောင် လိုရင်းတိုရှင်း ဆက်လက် ဖော်ပြသွားမှာပါ။

ကနေ့အချိန်အနေအထားအရ လေ့လာသူတွေဟာ စာတွေအများကြီးကို မဖတ်ချင်ကြတော့ပါဘူး။ လိုရင်းပဲ သိချင်ကြတဲ့ခေတ် ဖြစ်နေပါပြီ။ အင်တာနက်ကြောင့်ပါ။ လိုရင်းကိုသိရရင် ကျန်တာက ချက်ခြင်း သိဖို့မလိုဘူးလေ။ လိုလာတော့မှ ကြည့်လိုက်လို့ ရနေတာကိုး။

ဒါကြောင့် အပိုတွေကို လျှော့ပြီး လိုရင်းကိုပဲ ဆက်လိုက်ကြရအောင်။

အခန်း (၁) – ES6

ခြောက်ကြိမ်မြောက် Version ဖြစ်တဲ့အတွက် ES6 လို့ အတိုကောက်ပြီး၊ (၂၀၁၅) ခုနှစ်မှာ ဖန်တီးခဲ့တဲ့ အတွက် ECMAScript2015 လို့လည်းခေါ်တဲ့ နည်းပညာဟာ လက်ရှိ JavaScript (ES5) မှာ အသုံးဝင်တဲ့ ဖြည့်စွက် လုပ်ဆောင်ချက်ပေါင်းများစွာနဲ့ အဆင့်မြှင့်တင်ထားတဲ့ နည်းပညာ ဖြစ်ပါတယ်။ ES6 မှာ ပါဝင်လာတဲ့ လုပ်ဆောင်ချက်တွေထဲက React ကို လေ့လာတဲ့အခါမှာ မဖြစ်မနေ သိထားသင့်တဲ့ လုပ်ဆောင်ချက်တွေကို ရွေးထုတ်ဖော်ပြပါမယ်။ အားလုံးကိုသေချာမှတ်ထားပေးပါ။ ရှေ့လျှောက် ဒီရေး နည်းတွေကို ဆက်တိုက် အသုံးပြုသွားတော့မှာ မို့လို့ပါ။

Block-scope Variables

JavaScript မှာ ပုံမှန်အားဖြင့် **var** Keyword ကို သုံးပြီး Variable တွေ ကြေညာနိုင်ပါတယ်။ အဲ့ဒီလို ကြေညာထားတဲ့ Variable တွေဟာ Function Scope သဘောသဘာဝ ရှိကြပါတယ်။ ဥပမာ –

```
function app() {
  if(true) {
     var i = 10;
  }

  return i;
}

app();      // => 10
```

နမူနာ Function မှာ Variable `i` ကို `if` Statement ထဲမှာ ကြေညာထားပေမယ့် အပြင်ကနေလည်း အသုံးပြုလို့ ရနေတာကို တွေ့ရမှာပါ။ ES6 မှာပါလာတဲ့ **`let`** Keyword ကို အသုံးပြု၍ Variable တွေ ကြေညာမယ်ဆိုရင်တော့ Block Scope Variable တွေကို ရရှိမှာ ဖြစ်ပါတယ်။ ဥပမာ –

```
function app() {
  if(true) {
     let i = 10;
  }

  return i;
}

app();      // => ReferenceError: i is not defined
```

ဒီနမူနာမှာ အလားတူပဲ Variable `i` ကို `if` Statement ထဲမှာပဲ ကြေညာထားပါတယ်။ ဒါပေမယ့် ပြင်ပ ကနေ ရယူအသုံးလို့ မရတော့တာကို တွေ့ရမှာ ဖြစ်ပါတယ်။ ဒါဟာ မဆိုင်တဲ့ Variable တွေကို မှားယွင်း အသုံးပြုမိခြင်းကနေ ကာကွယ်ပေးလို့ ပိုကောင်းတဲ့ စနစ်ဖြစ်ပါတယ်။ ဒါကြောင့် အခုနောက်ပိုင်း **`var`** ကို လုံးဝ မသုံးသင့်တော့ဘူးလို့ ပြောကြတာမျိုးတွေ ရှိလာပါပြီ။

နောက်ထပ် ES6 မှာပါဝင်တဲ့ Block Scope Variable တစ်မျိုးကတော့ Constant ဖြစ်ပါတယ်။ **`const`** Keyword ကို အသုံးပြု ကြေညာရပါတယ်။ Constant ဖြစ်လို့ တန်ဖိုးကို တစ်ကြိမ်သာ သတ်မှတ်နိုင်မှာဖြစ် ပြီး ပြင်ဆင်ခွင့်ကိုပေးမှာ မဟုတ်ပါဘူး။ ဥပမာ –

```
const num = 1;
num = 2;    //=> TypeError: invalid assignment to const `num`
```

ဒါဟာလည်း အရေးပါတဲ့ ဖြည့်စွက်ချက် ဖြစ်ပါတယ်။ မပြောင်းသင့်တဲ့ တန်ဖိုးတွေကို မတော်တဆ ပြောင်းလဲမိခြင်းကနေ ကာကွယ်ပေးပါတယ်။ ထူးခြားချက်တစ်ခုကိုတော့ သတိပြုပါ။ ဒီလိုပါ –

```
const user = { name: 'Bob' };
user.name = 'Tom';
```

ဒီရေးနည်းက TypeError မဖြစ်ဘဲ အလုပ်လုပ်သွားမှာ ဖြစ်ပါတယ်။ Constant ဟာ Object တစ်ခုဖြစ်မယ် ဆိုရင် Object Property တွေ ပြင်တာကိုတော့ လက်ခံတယ်ဆိုတဲ့ သဘောပါ။

Map, Filter, Reduce

နောက်တစ်ဆင့်မှာ Arrow Function အကြောင်း ပြောချင်ပါတယ်။ နမူနာတွဲပြချင်လို့ `map()`, `filter()` နဲ့ `reduce()` ဆိုတဲ့ အရမ်းအသုံးဝင်တဲ့ Array Function တွေအကြောင်း အရင်ပြောပါ မယ်။ `map()` Function ဟာ ပေးလိုက်တဲ့ Array ပေါ်မှာ အခြေခံပြီး အခြား Array တစ်ခုကို ပြန်ပေးတဲ့ Function ဖြစ်ပါတယ်။ ဥပမာ –

```
let nums = [ 1, 2, 3, 4 ];
let result = nums.map(function(n) {
  return n + 1;
});

result;     // => [ 2, 3, 4, 5 ]
```

နမူနာအရ `nums` Array ကို အခြေခံပြီး `result` Array ကို ပြန်ရတာ ဖြစ်ပါတယ်။ Array Item တစ်ခု ချင်းစီ ပေါ်မှာ ပေးလိုက်တဲ့ Callback Function အလုပ်လုပ်သွားလို့ မူလတန်ဖိုးကို 1 ပေါင်းပြီး ပြန်ရတာ ကို တွေ့ရမှာ ဖြစ်ပါတယ်။ ဒီနေရာမှာ ထူးခြားချက်အနေနဲ့ သတိပြုရမှာက `nums` Array ရဲ့ မူလတန်ဖိုး တွေကို လုံးဝ မပြောင်းဘဲ Array အသစ်တစ်ခု ထုတ်ပေးတာဖြစ်လို့ ပိုစနစ်ကျတဲ့ ရေးနည်းလို့ ဆိုနိုင်ပါ တယ်။ ဒီရေးနည်းဟာ Functional Programming ကနေ လာပါတယ်။ ဒီစာအုပ်မှာ Functional Programming အကြောင်းတော့ ထည့်သွင်းဖော်ပြနိုင်မှာ မဟုတ်ပါဘူး။

အကျဉ်းချုပ်အနေနဲ့ နောက်ပိုင်းမှာ Array တွေ စီမံဖို့အတွက် `for` တို့ `for-in` တို့လို Loop တွေကို မသုံး တော့ဘဲ `map()` ကို အသုံးပြုသင့်တယ် လို့မှတ်သားနိုင်ပါတယ်။ ပိုစနစ်ကျသလို ရေးထုံးအရလည်း ပို ကျစ်လစ် သွားမှာပါ။

`filter()` Function ကလည်း အလားတူပဲ Array တစ်ခုပေါ်မှာ အခြေခံပြီး နောက် Array တစ်ခုကို ပြန်ပေးပါတယ်။ ဒါပေမယ့် `map()` လို Item အားလုံးကို ပြန်ပေးတာ မဟုတ်ဘဲ Condition ကိုက်ညီတဲ့ Item တွေကိုပဲ ရွေးပြီး ပြန်ပေးတာ ဖြစ်ပါတယ်။ ဥပမာ –

```
let nums = [ 1, 2, 3, 4 ];
let result = nums.filter(function(n) {
  return n % 2;
});

result      // => [ 1, 3 ]
```

နမူနာအရ `nums` Array ကို အခြေခံပြီး `result` Array ရပါတယ်။ ဒါပေမယ့် Item တစ်ခုချင်းစီပေါ်မှာ အလုပ်လုပ်ဖို့ ပေးလိုက်တဲ့ Callback Function မှာ Item Value `n` ကို 2 ရဲ့ စားကြွင်း ရှိမှ ပြန်ပေးဖို့ စစ်ထားလို့ စားကြွင်းရှိတဲ့ `[ 1, 3 ]` ကိုပဲ ပြန်ရတာ ဖြစ်ပါတယ်။ ဒီထက်ပို စိတ်ဝင်စားဖို့ကောင်းတဲ့ ဥပမာကို ပေးရရင် ဒီလိုပါ။

```
let users = [
  { name: 'Bob', gender: 'male' },
  { name: 'Alice', gender: 'femail' },
  { name: 'Tom', gender: 'male' }
];

let result = users.filter(function(user) {
  return user.gender == 'male';
});

result;

// [
//     { name: 'Bob', gender: 'male' },
//     { name: 'Tome', gender: 'male' }
// ]
```

နမူနာအရ ပေးထားတဲ့ `users` Object Array ထဲက `gender` Property တန်ဖိုး `male` ဖြစ်နေသူတွေကိုသာ ပြန်ပေး `result` Array ကို ရရှိခြင်းပဲ ဖြစ်ပါတယ်။

`reduce()` Function ကတော့ ပေးလိုက်တဲ့ Array ပေါ်မှာ အခြေပြုပြီး တန်ဖိုးတစ်ခုကို ပြန်ပေးပါတယ်။ ဥပမာ –

```
let nums = [ 1, 2, 3, 4 ];
let result = nums.reduce(function(a, n) {
  return a + n;
});

result;     // => 10
```

`reduce()` ရဲ့ Callback ကတော့ Argument နှစ်ခုလက်ခံပါတယ်။ အရင်တန်ဖိုးနဲ့ လက်ရှိတန်ဖိုးပါ။ နမူနာအရ အရင်တန်ဖိုးကို လက်ရှိတန်ဖိုးနဲ့ ပေါင်းပေါင်းပေးသွားလို့ နောက်ဆုံးမှာ အားလုံးပေါင်းခြင်း တန်ဖိုးကို ရရှိမှာ ဖြစ်ပါတယ်။

ဒီ Function သုံးခုဟာ အလွန်အသုံးဝင်ပြီး ရှေ့လျှောက် ဆက်တိုက် အသုံးပြုသွားတော့မှာပါ။ ဒါကြောင့် ကောင်းကောင်းကြေညက် နားလည်အောင်၊ လိုအပ်ရင် နောက်တစ်ခေါက်လောက် ပြန်ကြည့်ထားပေးပါ။ ကြည့်ယုံနဲ့လည်း မကျေနပ်ပါနဲ့၊ ချရေးပြီးတော့လည်း စမ်းကြည့်ပါ။ Web Browser Console ထဲမှာတင် ရေးစမ်းလို့ ရနိုင်ပါတယ်။

Arrow Function

JavaScript မှာ Function ကြေညာနည်း (၂) နည်းရှိတယ်လို့ ပြောနိုင်ပါတယ်။ ပထမနည်းက ဒီလိုပါ –

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

နောက်တစ်နည်းက ဒီလိုပါ –

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

ဒီဒုတိယနည်းက၊ တစ်ကယ်တော့ Variable တစ်ခုရဲ့ တန်ဖိုးနေရာမှာ Anonymous Function တစ်ခုကို ထည့်ပေးလိုက်တာပါ။ JavaScript မှာ Function Name မပါတဲ့ Nameless Function ခေါ် Anonymous

Function တွေကို ကျယ်ကျယ်ပြန့်ပြန့် သုံးကြပါတယ်။ ES6 မှာတော့ Arrow Function လို့ခေါ်တဲ့ ရေးထုံးသစ် ထပ်တိုး ပါဝင်လာပါတယ်။ ဒီလိုပါ –

```
let add = (a, b) => {
  return a + b;
}
```

အလွယ်မှတ်ချင်ရင် Parameter List ရှေ့က `function` Keyword ကို ဖယ်ထုတ်လိုက်ပြီး Parameter List နောက်မှာ => သင်္ကေတလေးကို ထည့်ပေးလိုက်တာပါ။ ဒီ Arrow Function ရေးထုံးမှာ ပါဝင်တဲ့ Statement က တစ်ကြောင်းတည်းဆိုရင် တွန့်ကွင်းတွေနဲ့ `return` Keyword ကို မထည့်ဘဲ ရေးလို့ရပါတယ်။ ဒီလိုပါ –

```
let add = (a, b) => a + b;
```

ပြီးတော့ Parameter တစ်ခုတည်းဆိုရင် ဝိုက်ကွင်း အဖွင့်ပိတ် မထည့်ဘဲလည်း ရေးလို့ရပါတယ်။ ဒီလိုပါ –

```
let twice = n => n * 2;
```

ဒါဆိုရင် `n` ကို လက်ခံပြီး 2 နဲ့မြှောက် ပြန်ပေးတဲ့ Function တစ်ခုကို ရသွားတာပါ။ အစပိုင်း အမြင်စိမ်းပြီး မျက်စိလည်ချင်စရာ ကောင်းပေမယ့်၊ သဘောပေါက်သွားရင် တော်တော်ရေးလို့ကောင်းပြီး အသုံးဝင်တဲ့ ရေးထုံးဖြစ်တယ်ဆိုတာကို တွေ့ရမှာပါ။ အထက်မှာ လေ့လာခဲ့တဲ့ Array Function တွေရဲ့ တွဲသုံးပြပါမယ်။

```
// Regular Function
let nums = [ 1, 2, 3, 4];
let result = nums.map(function(n) {
  return n + 1;
});

// Arrow Function
let nums = [ 1, 2, 3, 4 ];
let result = nums.map(n => n + 1);
```

ရေးထုံးကျစ်လစ်မှုပိုင်းမှာ အများကြီး ကွာသွားတာကို တွေ့ရမှာပါ။ ဒီထက်ပိုပြီး တိုချုပ်ရင်လည်း ဒီလို တိုက်ရိုက်ရေးလိုက်လို့ ရနိုင်ပါတယ်။

```
[ 1, 2, 3, 4 ].map(n => n + 1);     // => [ 2, 3, 4, 5 ]
```

အလားတူပါပဲ `filter()` တို့ `reduce()` တို့မှာလည်း ဒီတိုင်းရေးလို့ ရပါတယ်။

```
[ 1, 2, 3, 4 ].filter(n => n % 2);      // => [ 1, 3 ]
[ 1, 2, 3, 4 ].reduce((a, n) => a + n);// => 10
```

`reduce()` မှာ Parameter နှစ်ခုမို့ ဝိုက်ကွင်းနဲ့ ထည့်ရေးခဲ့ရတာလေး ပြန်သတိပေးချင်ပါတယ်။ ဒီ Arrow Function ရေးထုံးဟာလည်း ရှေ့လျှောက်ဆက်တိုက် အသုံးပြုသွားမယ့် ရေးထုံးဖြစ်ပါတယ်။ ဒါကြောင့် စမ်းရေးကြည့်ပြီး ကောင်းကောင်းနားလည်အောင် သေချာပြန်ကြည့်ထားစေချင်ပါတယ်။

Default Parameter Value and Rest Parameter

Default Parameter Value လုပ်ဆောင်ချက်ကတော့ PHP တို့ Python တို့မှာ အရင်ကတည်းက ပါဝင်တဲ့ လုပ်ဆောင်ချက်ပါ။ JavaScript အတွက်တော့ ES6 ကျတော့မှပဲ ပါလာပါတယ်။ အသုံးဝင်တဲ့ လုပ်ဆောင်ချက်ပါပဲ။ ဥပမာ –

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

add(1);     // => NaN
```

နမူနာမှာ `add()` Function ကိုခေါ်တဲ့အခါ Argument နှစ်ခုပေးရမယ့်အစား တစ်ခုပဲ ပေးလိုက်တဲ့အခါ ဒုတိယ Argument က `undefined` ဖြစ်သွားလို့ အဆင်မပြေတဲ့သဘောကို တွေ့ရနိုင်ပါတယ်။ Function ကြေညာစဉ်မှာ Parameter တန်ဖိုး ထည့်သတ်မှတ်ခဲ့မယ်ဆိုရင် အခုလို ဖြစ်သွားပါမယ်။

```
function add(a, b = 0) {
  return a + b;
}

add(1);     // => 1
```

b Parameter အတွက် ခေါ်ယူစဉ်မှာ Argument မပေးခဲ့ပေမယ့် 0 ကို Default Value အနေနဲ့ အသုံးပြုသွားလို့ အဆင်ပြေသွားတဲ့ သဘောပါ။ Argument တွေ Parameter တွေကြောင့်လည်း မျက်စိလည် မသွားပါနဲ့ဦး။ ဒီတန်ဖိုးတွေနဲ့ ပက်သက်ပြီး Function ကြေညာစဉ်မှာ Parameter လို့ခေါ်ပြီး Function ခေါ်ယူတဲ့အခါ Argument လို့ ခေါ်ကြတဲ့အတွက် နှစ်မျိုးဖြစ်နေတာပါ။ တစ်ကယ်တော့ အတူတူပါပဲ။

Rest Parameter ဆိုတာကတော့ Function ကြေညာစဉ်မှာ Parameter အရေအတွက်ကို အသေ မသတ်မှတ်တော့ပဲ ပေးချင်သလောက်ပေး အကုန်လက်ခံလိုတဲ့အခါ အသုံးဝင်ပါတယ်။ ဥပမာ –

```
function add(a, b, ...c) {
  return c;
}

add( 1, 2, 3, 4, 5 )          // => [3, 4, 5]
```

နမူနာမှာ Argument အနေနဲ့ ပေးလိုက်တဲ့ တန်ဖိုးတွေထဲက 1 က a အတွက်ဖြစ်သွားပြီး၊ 2 က b အတွက်ဖြစ်သွားပါတယ်။ ကျန်တဲ့ 3, 4, 5 ကတော့ Array အနေနဲ့ c အတွက် ဖြစ်သွားတဲ့ သဘောကို တွေ့ရမှာ ဖြစ်ပါတယ်။ ထုံးစံအတိုင်း လက်တွေ့ချရေးကြည့်ပြီး သေချာနားလည်အောင် လုပ်ထားဖို့ လိုပါမယ်။

Spread Operator

Array, String စသည့် Loop ပါတ်လို့ ရနိုင်တဲ့ Iteratable Value တွေကို ခွဲဖြန့် ပေးနိုင်တဲ့ လုပ်ဆောင်ချက်ကို Spread လုပ်တယ်လို့ ခေါ်ပါတယ်။ ဥပမာ –

```
let alphas = [ 'a', 'b', 'c' ];
let nums = [1, 2, 3];
[ alphas, nums ];    // => [ ['a', 'b', 'c'], [1, 2, 3] ]
[ ...alphas, ...nums ]; // => ['a', 'b', 'c', 1, 2, 3]
```

ပထမနမူနာမှာ `alphas` Array နဲ့ `nums` Array ကို နောက် Array တစ်ခုထဲ ဒီတိုင်းထည့်လိုက်တဲ့အခါ ရလဒ် Array က နှစ်ထပ်ဖြစ်သွားတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။ နောက်နမူနာမှာတော့ အစက်ကလေးသုံးစက် (...) ရှေ့ကပါသွားလို့ Array Content ကို ဖြန့်ချပေးသွားလို့ အတွဲလိုက် တစ်ဆက်တည်း ဖြစ်သွားတာကို တွေ့ရမှာပါ။ Array Spread ဆိုတာ ဒါပါပဲ။ ရှေ့ဆက် အသုံးပြုသွားမယ့် လုပ်ဆောင်ချက် ဖြစ်ပါတယ်။

Destructuring

Array တွေ Object တွေကို ဖြည်ချလို့ရတဲ့ လုပ်ဆောင်ချက်ပါ။ ဥပမာ – ရိုးရိုးရေးမယ်ဆိုရင် ဒီလို ရေးရနိုင်ပါတယ်။

```
let fruits = [ 'Apple', 'Orange' ];
let apple = fruits[0];
let orange = fruits[1];

apple;      // => Apple
```

ဒါကိုပဲ Destructure လုပ်ပြီးရေးမယ်ဆိုရင် အခုလိုရေးလိုရပါတယ် –

```
let fruits = [ 'Apple', 'Orange' ];
let [ apple, orange ] = fruits;

apple;      // => Apple
```

Object တွေမှာလည်း အတူတူပါပဲ။ ရိုးရိုးရေးမယ်ဆိုရင် ဒီလိုရေးရနိုင်ပါတယ် –

```
let user = { name: 'Alice', age: 22 };
let name = user.name;
let age = user.age;
```

အဲ့ဒါကို Destructure လုပ်ပြီးရေးမယ်ဆိုရင်တော့ အခုလို ရေးနိုင်ပါတယ်။

```
let user = { name: 'Alice', age: 22 };
let { name, age } = user;
```

ဒီနည်းနဲ့ ကျစ်လစ်တဲ့ရေးဟန်ကို ရရှိနိုင်ပါတယ်။ ပိုစိတ်ဝင်စားဖို့ကောင်းအောင် Function နဲ့ နမူနာ တွဲသုံးပြပါဦးမယ်။ ရိုးရိုးဆိုရင် ဒီလိုပါ။

```
function show(user) {
  return user.name + ' is ' + user.age + ' years old.';
}

let user = { name: 'Alice', age: 22 };

show(user);    // => Alice is 22 years old.
```

Destructure ကိုအသုံးပြုပြီး ရေးမယ်ဆို ဒီလိုရေးနိုင်ပါတယ်။

```
function show({ name, age }) {
  return name + ' is ' + age + ' years old.';
}
```

Function Parameter သတ်မှတ်ကတည်းက Destructure လုပ်ပြီး သတ်မှတ်လိုက်တာ ဖြစ်ပါတယ်။ တူညီတဲ့ ရလဒ်ကိုပဲ ရမှာဖြစ်ပြီး ဒီရေးဟန်ကိုလည်း ဆက်လက်အသုံးပြုသွားမှာ ဖြစ်ပါတယ်။

String Interpolation

ပြီးခဲ့တဲ့ နမူနာမှာပဲ String တွေကို Format လုပ်ရတာ အဆင်မပြေတာကို တွေ့ခဲ့ရပါပြီ။ + Operator တွေ သုံးပြီး Variable တွေနဲ့ String တွေကို တွဲရဆက်ရတာ ရေးရအလုပ်ရှုပ်သလို ဖတ်ရလည်း ခက်ပါတယ်။ ES6 String Interpolation ကိုအသုံးပြုပြီး ဒီပြဿနာကို ဖြေရှင်းနိုင်ပါတယ်။

```
function show({ name, age }) {
  return `${name} is ${age} years old.`;
}
```

ဒီရေးနည်းမှာတော့ Plus + တွေနဲ့ တွဲဆက်နေစရာမလိုတော့ဘဲ String အတွင်းမှာ တန်ဖိုးတွေကို တစ်ခါတည်း ထည့်ပေးလိုက်လို့ ရသွားပါပြီ။ တန်ဖိုးတွေကိုတော့ ဒေါ်လာသင်္ကေတနောက်မှာ တွန့်ကွင်း အဖွင့်အပိတ်နဲ့ ထည့်ပေးရပါတယ်။ ဒါဟာလည်း တော်တော်အသုံးဝင်တဲ့ လုပ်ဆောင်ချက်လေး တစ်ခုပါပဲ။ String ကို Single Quote / Double Quote များနှင့် မရေးသားပဲ Backtick အဖွင့်အပိတ်ဖြင့် ရေးသားတာကိုတော့ သတိပြုပါ။

Property Shorthand & Trailing Comma

မူလ JavaScript မှာ ဒီလိုပြဿနာမျိုးကို မကြာခဏ တွေ့ရပါတယ်။

```
let name = 'Bob';
let age = 22;
let user = {
    name: name,
    age: age,
    say: function() {
        return 'Hello'
    }
}
```

Property Name နဲ့ Value အဖြစ်အသုံးပြုတဲ့ Variable အမည်တူနေတာပါ။ ဒီလိုအခြေအနေမျိုးမှာ ES6 Property Shorthand အကူအညီနဲ့ အခုလိုရေးလို့ရပါတယ်။

```
let name = 'Bob';
let age = 22;
let user = {
    name,
    age,
    say() {
        return 'Hello'
    }
};
```

Method `say()` အတွက်လည်း function Keyword မပါဝင်တော့တာကို သတိပြုပါ။ ရိုးရိုးလေးနဲ့ အသုံးဝင်တဲ့ ရေးဟန်ဖြစ်ပါတယ်။ Trailing Comma ဆိုတာကတော့ ရိုးရိုး JavaScript မှာ ဒီကုဒ်ဟာ

Error တက်ပါတယ်။

```
let users = [
  { name: 'Alice', age: 22 },
  { name: 'Bob', age: 23 },
  { name: 'Mary', age: 22 },
];
```

နောက်ဆုံးက Comma လေးတစ်ခု ပိုသွားလို့ပါ။ ES6 မှာတော့ ဒါကိုလက်ခံပါတယ်။ ပိုသွားလည်းပဲ Error မဖြစ်တော့ဘဲ လက်ခံအလုပ်လုပ်ပေးသွားမှာ ဖြစ်ပါတယ်။

Class

JavaScript ဟာ မူလဒီဇိုင်းအရ Classical OOP မဟုတ်ပါဘူး။ Prototype OOP လို့ခေါ်ပါတယ်။ Object တွေ တည်ဆောက်ဖို့ Class တွေကိုမသုံးဘဲ Object Constructor နဲ့ JSON တို့ကို အသုံးပြုပါတယ်။ ဒါပေမယ့် ES6 မှာတော့ Class ရေးထုံးပါဝင်လာပါပြီ။ ဒီလိုပါ –

```
class Animal {
  constructor(legs, wings) {
     this.legs = legs;
     this.wings = wings;
  }

  say() {
     return 'Hello, World';
  }
}
```

Constructor တစ်ခုနဲ့ Method တစ်ခုတို့ ပါဝင်တဲ့ Class ဖြစ်ပါတယ်။ အခြားသော Classical OOP ရေထုံးတွေနဲ့ သိပ်မကွာပါဘူး။ ထူးခြားချက်ကို ရွေးထုတ်ပြောရရင် Public, Private စသဖြင့် Access Control Modifier တွေ မရှိခြင်းနဲ့ Method တွေကြေညာတဲ့အခါမှာ function Keyword ထည့်စရာ မလိုခြင်းတို့ကို သတိပြုသင့်ပါတယ်။ Static Method တွေတော့ ရေးလို့ရပါတယ်။ ဒီလိုပါ –

```
class Math {
  static add(a, b) {
     return a + b;
  }
}

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

ES6 Class ထွက်ပေါ်ခါစမှာ Property တွေကို တိုက်ရိုက် သတ်မှတ်အသုံးပြုလို့မရဘဲ Constructor အတွင်းမှာပဲ `this` ကနေတစ်ဆင့် ကြေညာသတ်မှတ်ရပါတယ်။ နောက်ပိုင်းမှာတော့ Class Field လို့ခေါ်တဲ့ လုပ်ဆောင်ချက် ပါဝင်လာလို့ Property တွေကို တိုက်ရိုက် ကြေညာသတ်မှတ်လို့ ရလာသလို Method တွေကိုလည်း Class Field အနေနဲ့ပဲ ရေးလို့ရလာပါတယ်။ ဥပမာ –

```
class Dog {
  name = 'Bobby';

  walk() {
     return `${this.name} is walking`;
  }

  run = () => {
     return `${this.name} is running`;
  }
}

let bobby = new Dog();
bobby.name;    // => Bobby
bobby.run();       // => Bobby is running
```

နမူနာမှာ `name` Property ကို Class Field အနေနဲ့ တိုက်ရိုက် ကြေညာထားပြီး၊ `run` ဟာလည်း Class Field တစ်ခုဖြစ်ပါတယ်။ ဒါပေမယ့် သူ့အတွက်ပေးထားတာ တန်ဖိုးတစ်ခုမဟုတ်ဘဲ Arrow Function တစ်ခုဖြစ်လို့ Method အနေနဲ့ အသုံးပြုရတာကို တွေ့နိုင်ပါတယ်။ ဒီရေးနည်းဟာ ရှေ့ဆက် အသုံးပြုသွားမယ့် ရေးနည်းပဲ ဖြစ်ပါတယ်။

Module

မူလ JavaScript မှာ Module လုပ်ဆောင်ချက် မရှိပါဘူး။ ဒါကြောင့် သီးခြားဖိုင်တွေမှာ ခွဲရေးထားတဲ့ JavaScript Code တွေကို တွဲဖက်အသုံးပြုလိုရင် HTML အတွင်းမှာ `<script src="">` Element ကို သုံးကြရပါတယ်။ ES6 မှာတော့ Module လုပ်ဆောင်ချက် ပါဝင်လာပါတယ်။

```
// math.js
export const PI = 3.1416

export function area(r) {
    return PI * r * r;
}

// app.js
import { PI, area } from './math'

area(8)    // => 201.0624
```

နမူနာမှာ `math.js` က သူရဲ့ PI တန်ဖိုးနဲ့ `area()` Function လုပ်ဆောင်ချက်တွေကို `export` လုပ်ပေးထားပြီး `app.js` က ဒီတန်ဖိုးနဲ့ လုပ်ဆောင်ချက်တွေကို `import` လုပ်ယူ အသုံးပြုထားတာကို တွေ့ရနိုင်ပါတယ်။ Import လုပ်ယူတဲ့အခါ လိုအပ်ရင် Module File Path ကို အပြည့်အစုံပေးရပေမယ့်၊ နောက်ဆုံးက `.js` Extension က ချန်ထားလို့ရပါတယ်။

ဒီကုဒ်ကို လက်တွေ့ စမ်းသပ်ဖို့တော့ နည်းနည်း ခက်ပါတယ်။ Browser တွေက Support လုပ်ပေမယ့် တစ်ခြားကုဒ်တွေကို Browser Console ထဲမှာ ကောက်ရေးကြည့်လိုက်လို့ မရပါဘူး။ အနည်းဆုံး `<script type="module">` Element ကို အသုံးပြုဖို့ လိုပါတယ်။ ပြီးတော့ ဖိုင်ကနေ တိုက်ရိုက်စမ်းလို့ မရပါဘူး၊ Web server လေးတစ်ခုလောက် ရှိမှ စမ်းလို့ရမှာပါ။ ဒါကြောင့် ရေးထုံးကိုပဲ မှတ်ထားပါ၊ React ကုဒ်တွေ ရေးတဲ့အခါမှပဲ တစ်ခါတည်း ထည့်စမ်းကြတာပေါ့။ Export နဲ့ Import မှာ ရေထုံးမူကွဲတွေ အများကြီးရှိပေမယ့် နောက်ထပ် Default Export လောက်ကို လေ့လာထားလိုက်ရင် ရပါပြီ။ ဒီလိုပါ –

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

(သို့မဟုတ်)

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

export default add;
```

Default Export ပေးထားတဲ့ တန်ဖိုးနဲ့ လုပ်ဆောင်ချက်တွေကို Import လုပ်ယူပုံက ဒီလိုပါ။

```
// app.js
import add from './math';
```

(သို့မဟုတ်)

```
// app.js
import sum from './math'
```

နမူနာမှာ Import လုပ်ယူတဲ့အခါ အမည်ကို `sum` လို့ ပေးထားပေမယ့် Default Export ပေးထားတဲ့ `add` လုပ်ဆောင်ချက်ကိုပဲ `sum` အမည်နဲ့ ရရှိမှာဖြစ်ပါတယ်။ ဒီရေးနည်းကို ဆက်လက်အသုံးပြုသွားမှာ ဖြစ်ပါတယ်။

အခန်း (၂) – React Basic

React ကိုအသုံးပြုပြီး စမ်းသပ်အဆင့်ကနေ လက်တွေ့သုံး ပရောဂျက်တွေအထိ ရေးသားဖို့အတွက် လိုအပ်မယ့် နည်းပညာအားလုံးက `create-react-app` လို့ခေါ်တဲ့ Package တစ်ခုမှာ အားလုံး စုစည်း ပါဝင်ပါတယ်။ ဒီ Package ကို မသုံးဘဲ ရိုးရိုး JavaScript ကုဒ်ဖိုင်အနေနဲ့ ထည့်သုံးလည်း ရနိုင်ပါ တယ်။ ဒီစာအုပ်မှာ `create-react-app` ကို အသုံးပြုပြီးတော့ပဲ ဖော်ပြသွားမှာပါ။

`create-react-app` ဟာ NPM Package တစ်ခုဖြစ်ပြီး စမ်းသပ်ရေးသားလိုတဲ့ ကွန်ပျူတာမှာ NodeJS ရှိထားဖို့ လိုအပ်မှာဖြစ်ပါတယ်။ NodeJS နဲ့ NPM အကြောင်းကို ဒီစာအုပ်မှာ အကျယ်မချဲ့ပါဘူး။ လိုသလောက်ပဲ ပြောသွားမှာပါ။ ပထမဦးဆုံးအနေနဲ့ ကိုယ့်စက်ထဲမှာ NodeJS ကို Install လုပ်ဖို့လိုပါ တယ်။ ပုံမှန်အားဖြင့် NodeJS ကို Install လုပ်လိုက်ရင် NPM က တစ်ခါတည်း ပါဝင်သွားမှာပါ။ ဒီစာရေး နေချိန်မှာ နောက်ဆုံး NodeJS Version က 13.12.0 ဖြစ်ပြီး Recommended Version က 12.16.1 ဖြစ်ပါ တယ်။ nodejs.org မှာ Download ရယူနိုင်ပါတယ်။ Download ယူနည်းတွေ Install လုပ်နည်းတွေ ထည့် မပြောပါဘူး။ သက်ဆိုင်ရာ Website နဲ့ အင်တာနက် လမ်းညွှန်တွေကနေတစ်ဆင့် ကိုယ်တိုင် Install လုပ် ယူရမှာဖြစ်ပါတယ်။ NodeJS နဲ့ NPM ကို Install လုပ်ပြီးနောက် Command Prompt (သို့မဟုတ်) Terminal ကိုဖွင့်ပြီး အခုလို စစ်ကြည့်နိုင်ပါတယ်။

```
>> node -v
v12.16.1

>> npm -v
6.14.4
```

ဒီလိုရလဒ်မျိုးကို မရသေးဘူးဆိုရင် Install လုပ်ထားတာ မပြည့်စုံသေးလို့ပါ။

မှတ်ချက် ။ ။ အနက်ရောက် Box နဲ့ ဖော်ပြထားတာတွေ အားလုံးက Command Prompt (သို့မဟုတ်) Terminal မှာ ရိုက်ရမယ့် Command တွေဖြစ်ပါတယ်။ ဒီမျှားလေးနှစ်ခု >> က ထည့်ရိုက်စရာ မလိုပါဘူး။ ရိုက်ရမယ့် Command ကို မျှားထိုး ပြထားတာပါ။ Command Prompt နဲ့ Terminal ဆိုပြီး နှစ်မျိုး ပြော ပြောနေရတာ အလုပ်ရှုပ်ပါတယ်။ Windows မှာ Command Prompt လို့ခေါ်ပြီး ကျန် OS တွေမှာ Terminal လို့ ခေါ်ကြလေ့ရှိလို့ပါ။ ရှေ့လျှောက် Terminal လို့ပဲ ဆက်ပြောသွားပါမယ်။

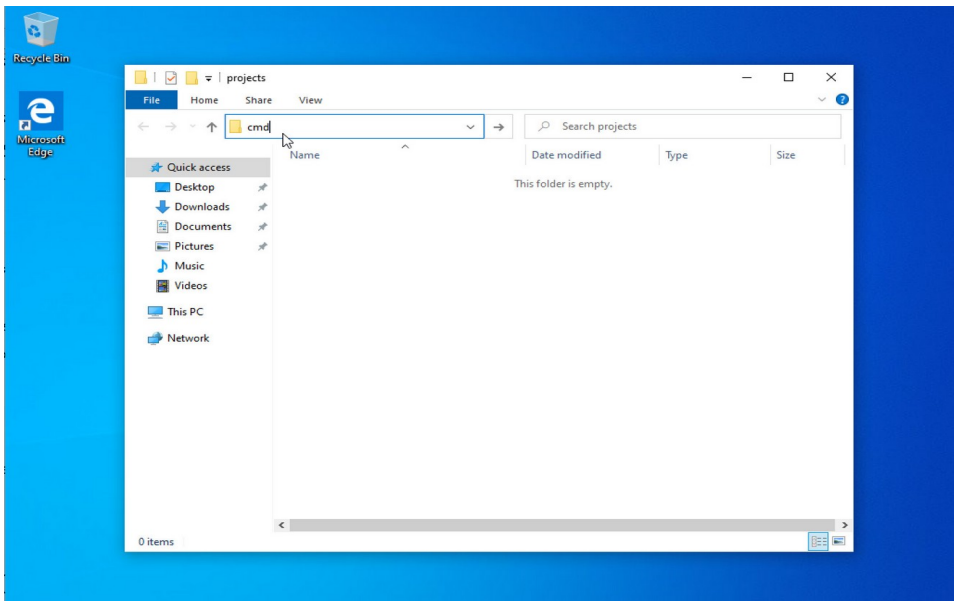
ဒီအဆင့်ထိ ရပြီဆိုရင်တော့ Step-by-Step လိုက်လုပ်လို့ရပါပြီ။ ဒီအခန်းက အရေးအကြီးဆုံးပါပဲ။ ဖတ်ယုံပဲ မဖတ်ပါနဲ့၊ တစ်ခါတည်း လက်တွေ့လိုက်လုပ်ကြည့်ပါ။ ဖတ်ယုံပဲဖတ်လို့ နားလည်တယ်ထား၊ နောက်နေ့ တန်းမေ့သွားမှာပါ။ တစ်ခါတည်း လိုက်လုပ်မှ ပိုပြီးနားလည်သလို မှတ်လည်းမှတ်မိမှာ ဖြစ်ပါတယ်။

Step 1 – Install create-react-app

ပထမအဆင့်အနေနဲ့ မိမိနှစ်သက်ရာ အမည်နဲ့ ဖိုဒါတစ်ခုဆောက်ပါ။ အဲ့ဒီဖိုဒါထဲမှာ Terminal ကိုဖွင့်ပါ။

Tip – Windows Explorer ရဲ့ Address Bar ထဲမှာ `cmd` လို့ ရိုက်၊ Enter နှိပ်ပြီး လက်ရှိ ရောက်နေတဲ့ ဖိုဒါ ထဲမှာ Terminal ကို တန်းဖွင့်လို့ရပါတယ်။

ပြီးရင်



`create-react-app` ကို အခုလို Install လုပ်ပါ။

```
>> npm install create-react-app
```

NPM ကို အသုံးပြုပြီး လက်ရှိဖိုဒါထဲမှာ `create-react-app` ကို Install လုပ်လိုက်တာပါ။ `install` အစား အတိုကောက် `i` လို့ ပြောရင်လည်း ရပါတယ်။ ဖွင့်ကြည့်ရင် `node_modules` ဆိုတဲ့ ဖိုဒါတစ်ခုနဲ့ `package-lock.json` ဆိုတဲ့ဖိုင်တစ်ခု ဝင်သွားတာကို တွေ့ရမှာပါ။ `node_modules` ဖိုဒါထဲမှာ Install လုပ်လိုက်တဲ့ Package တွေနဲ့ ဆက်စပ် Package တွေကို သိမ်းသွားမှာဖြစ်ပြီး `package-lock.json` မှာတော့ အဲ့ဒီ Package တွေရဲ့ Version နဲ့ ဆက်စပ်မှု အချက်အလက်တွေကို သိမ်းထားမှာ ဖြစ်ပါတယ်။

Step 2 – Create a React Project

နောက်တစ်ဆင့်အနေနဲ့ React ပရောဂျက်တစ်ခုကို အခုလို တည်ဆောက်ရပါမယ်။

```
>> npx create-react-app first
```

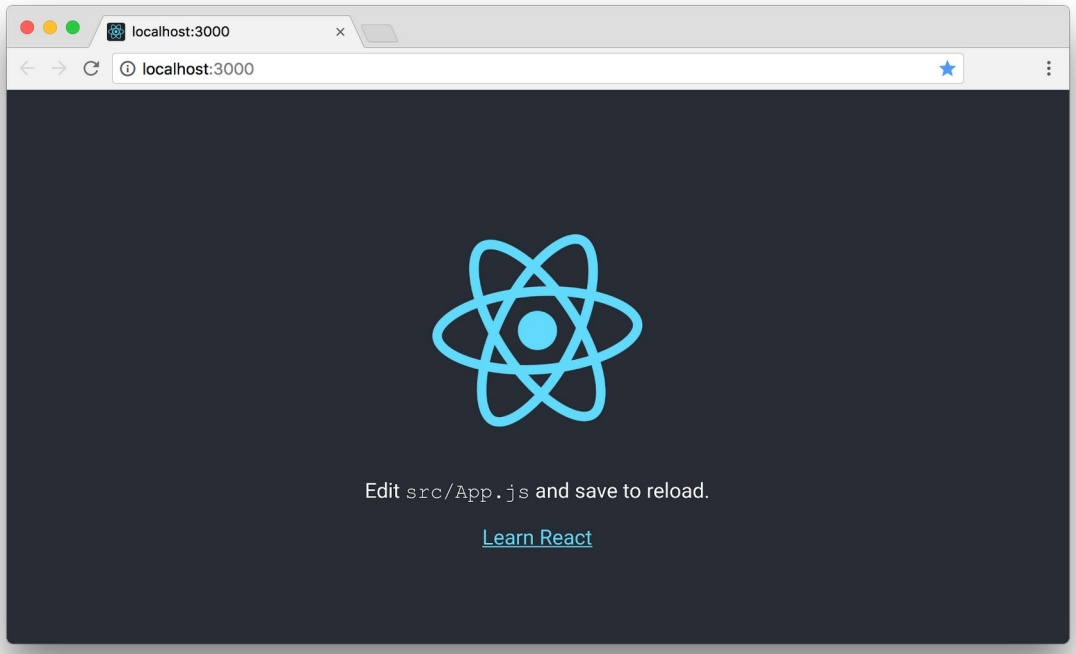
`npx` ကိုအသုံးပြုပြီး Install လုပ်ထားတဲ့ Package တွေကို Run လို့ ရပါတယ်။ ဒီလို ပရောဂျက်ဖိုဒါထဲမှာ Install လုပ်ထားတဲ့ Package တွေကို Local Package လို့ခေါ်ပြီး Global Package ဆိုတာလည်း ရှိပါသေးတယ်။ ထည့်မကြည့်ပါနဲ့ဦး၊ သိချင်ရင် နောက်မှလေ့လာပါ။ အခု Local Package နဲ့ပဲ ရှေ့ဆက်သွားပါမယ်။ ပေးထားတဲ့ Command အရ `create-react-app` ကို သုံးပြီး `first` အမည်နဲ့ React ပရောဂျက်တစ်ခု တည်ဆောက်လိုက်တာပါ။ တည်ဆောက်လိုက်တဲ့ React ပရောဂျက်ထဲကို Terminal မှာ သွားလိုက်ပါ။ အခုလို သွားလို့ရပါတယ်။

```
>> cd first
```

ပြီးရင် ပရောဂျက်ကို အခုလို Run လို့ရပါတယ်။

```
>> npm start
```

ဒီလို Run ပေးလိုက်တယ်ဆိုရင် Web Browser အလိုအလျောက် ပွင့်လာပြီး အခုလိုရလဒ်ကို ရရှိမှာ ဖြစ်ပါတယ်။



ဒါဟာ တည်ဆောက်လိုက်တဲ့ React ပရောဂျက် Sample ရလဒ် ဖြစ်ပါတယ်။ ဒီလိုရလဒ် ပေါ်တယ်ဆိုရင် React ပရောဂျက်တစ်ခု တည်ဆောက်ခြင်း ပြီးသွားပြီ ဖြစ်ပါတယ်။ မပေါ်ရင်တော့ Step 1 ကနေစပြီး သေချာဖတ်ပြီး နောက်တစ်ခေါက် ပြန်စမ်းကြည့်ပါ။

Step 3 – First React Component

ပရောဂျက်ဖိုဒါထဲက ဖိုင်တွေကို တစ်ချက်လေ့လာကြည့်ပါ။ လောလောဆယ်မှာ တစ်ခြားဖိုင်တွေကို ထိစရာ မလိုသေးပါဘူး။ `src` ဖိုဒါထဲက `App.js` ထဲမှာ ကျွန်တော်တို့ရဲ့ အဓိကကုဒ်တွေကို ရေးသားသွားမှာ ဖြစ်ပါတယ်။ ဒါကြောင့် `App.js` ထဲက Sample ကုဒ်တွေကို အကုန်ဖျက်ပြစ်လိုက်ပြီး ဒီကုဒ်ကို ကူးရေးပေးပါ။

```
import React from 'react';
```

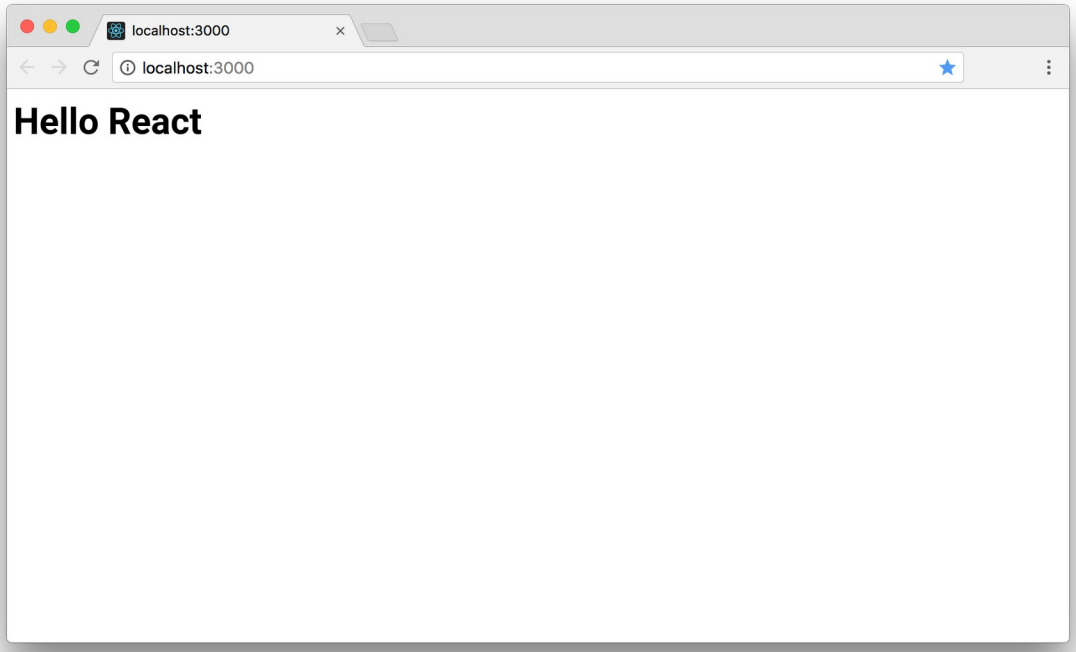
```
class App extends React.Component {
    render() {
        return <h1>Hello React</h1>;
    }
}

export default App;
```

ဒါဟာ အခြေခံအကျဆုံး React Component တစ်ခုဖြစ်ပါတယ်။ ဒီအဆင့်မှာ React Component တစ်ခုရဖို့ အချက် (၃) ချက်လိုတယ် လို့ မှတ်ထားပေးပါ။

၁။ `React.Component` ကို Extend လုပ်ထားတဲ့ Class ဖြစ်ရမယ်
၂။ `render()` Method ပါရမယ်။
၃။ Element တစ်ခုကို Return ပြန်ပေးရမယ်။

ဒီ (၃) ချက် ပြည့်စုံရင် React Component တစ်ခုရပါပြီ။ လွယ်ပါတယ်။ တစ်ခြားနည်းတွေလည်း ရှိသေးပေမယ့် တစ်ဆင့်ချင်းပဲ သွားပါမယ်။ ဒီအဆင့်မှာ React Component တစ်ခုရဖို့ ဒီ (၃) ချက်လိုတယ်လို့ သာ မှတ်ထားပေးပါ။

နမူနာအရ `App` Class ဟာ `React.Component` ကို Extend လုပ်ထားပြီး `render()` Method က `<h1>` Element တစ်ခုကို Return ပြန်ပေးထားပါတယ်။ တည်ဆောက်ရရှိလာတဲ့ `App` Component ကို နောက်ဆုံးမှာ Export လုပ်ပေးထားတာကို သတိပြုပါ။ အခုနေရလဒ်ကို ကြည့်ရင် အခုလိုတွေ့ရမှာပါ။



`create-react-app` က ပရောဂျက်အတွက် Hot Reloading လို့ခေါ်တဲ့ စနစ်ကို တစ်ခါတည်း ထည့်ထားပေးလို့ ရေးထားတဲ့ကုဒ်မှာ တစ်ခုခုပြင်လိုက်တဲ့အခါ နောက်တစ်ခါ ဖွင့်စရာမလိုဘဲ၊ Refresh တွေဘာတွေ လုပ်စရာ မလိုဘဲ ရလဒ်ကို တန်းမြင်ရတာကိုလည်း သတိပြုကြည့်ပါ။

Step 4 – JSX

ရေးထားတဲ့နမူနာမှာ ထူးခြားချက်တစ်ခု ကျန်ပါသေးတယ်။ `<h1>` ဆိုတဲ့ HTML Element ကို JavaScript ထဲမှာ တိုက်ရိုက်ထည့်ရေးထားပါတယ်။ ရိုးရိုး JavaScript အရဆိုရင် ဒါဟာ Syntax မှားနေပါတယ်။ တစ်ကယ်ဆို ဒီလိုဖြစ်သင့်ပါတယ်။

```
render() {
    return "<h1>Hello React</h1>";
}
```

ဒါမှ မှန်ကန်တဲ့ JavaScript ရေးထုံးဖြစ်မှာပါ။ React ပေါ်ခါစကဆိုရင် တူညီတဲ့ရလဒ်ရဖို့ အခုလိုရေးရပါတယ်။

```
React.createElement('<h1>', null, 'Hello React');
```

ဒီနည်းနဲ့ `<h1>` ကိုအသုံးပြုထားပြီး `Hello React` ဆိုတဲ့ Content ပါဝင်တဲ့ Component ကို တည်ဆောက်ယူရတာပါ။ ဒါပေမယ့် ရေးရတာ အဆင်မပြေပါဘူး။ ဒါကြောင့် နောက်ပိုင်းမှာ JSX လို့ခေါ်တဲ့ နည်းပညာကို တီထွင်ခဲ့တာပါ။ JSX ဆိုတာ တစ်ကယ်တော့ ရိုးရိုးလေးပါ။ HTML Code ကို JavaScript ထဲမှာ တိုက်ရိုက်ထည့်ရေးလို့ ရအောင် ထွင်ပေးလိုက်တဲ့ နည်းပညာလို့ အလွယ်မှတ်နိုင်ပါတယ်။ JSX ရေးထုံးဟာ အများအားဖြင့် HTML ရေးထုံးနဲ့ တူပါတယ်။ Element တိုင်းမှာ အပိတ်ပါရမယ်၊ `class` Attribute အစား `className` လို့ သုံးရမယ် စသဖြင့် ခြွင်းချက်တစ်ချို့ ရှိပေမယ့် ခေါင်းစားခံပြီး ကြိုမှတ်မနေပါနဲ့။ ကုဒ်နမူနာတွေ ရေးရင်းနဲ့ ဒီထူးခြားချက် လေးတွေက သူဘာသာ သတိပြုမိလာပါလိမ့်မယ်။ စောစောက ရေးခဲ့တဲ့ ကုဒ်ကို ဒီလို ပြင်ပြီး စမ်းကြည့်ပါ။

```
import React from 'react';

class App extends React.Component {
    render() {
        return (
            <div>
                <h1>Hello React</h1>
                <ul>
                    <li>Item One</li>
                    <li>Item Two</li>
                </ul>
            </div>
        )
    }
}

export default App;
```

`render()` Function က Return ပြန်ပေးတာ HTML Structure တစ်ခုဖြစ်သွားပါပြီ။ ထူးခြားချက် နှစ်ချက် ရှိပါတယ်။ ပထမတစ်ချက်က HTML (JSX) Structure ကို ရေးတဲ့အခါ တစ်ကြောင်းတည်းမရေးဘဲ ခွဲရေးချင်လို့ ဝိုက်ကွင်းအဖွင့်အပိတ်ထဲမှာ ရေးထားရပါတယ်။ မှတ်ထားပါ။ ဝိုက်ကွင်းအဖွင့်အပိတ်ထဲမှာမှ အခုလို ခွဲရေးလို့ ရပါတယ်။ ဒုတိယတစ်ခုကတော့ အထက်မှာ ပြောခဲ့ပြီးသားပါ။ React Component တစ်ခုဖြစ်ဖို့ Element တစ်ခုကို Return ပြန်ပေးရမယ်ဆိုတာ ပါပါတယ်။ တစ်ခုထက်ပိုလို့မရပါဘူး။ ဒါကြောင့် `<div>` တစ်ခုထဲမှာ အားလုံးကို စုရေးပြီး တစ်ခုတည်း အနေနဲ့ ပြန်ပေးထားရတာကို သတိပြုကြည့်ပါ။

Step 5 – Using Component

ဒီတစ်ခါ Component တစ်ခုတည်ဆောက်ပြီး နောက် Component တစ်ခုကနေ ယူသုံးတဲ့ ကုဒ်ကို ရေးကြည့်ပါမယ်။ ဒီလိုပါ –

```
import React from 'react';

class Item extends React.Component {
    render() {
        return <li>Content</li>;
    }
}

class App extends React.Component {
    render() {
        return (
            <div>
                <h1>Hello React</h1>
                <ul>
                    <Item />
                    <Item />
                </ul>
            </div>
        )
    }
}

export default App;
```

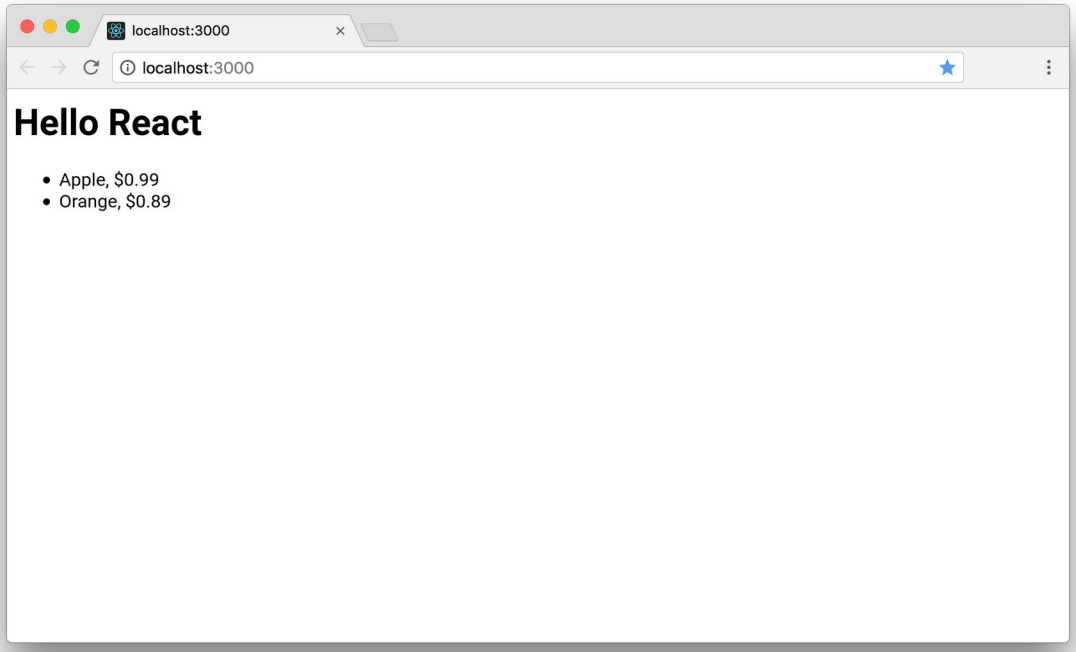
`Item` အမည်နဲ့ Component တစ်ခုကို အရင်ဆောက်ထားပြီးတော့ မှ `App` Component ထဲမှာ အဲ့ဒီ `Item` ကို ယူသုံးထားတာပါ။

Step 6 – props

နည်းနည်းပို အရေးကြီးတာလေး လာပါပြီ။ Component တစ်ခုနဲ့တစ်ခု ခေါ်ယူအသုံးပြုတဲ့အခါ Data ပေးလို့ရပါတယ်။ HTML Property အနေနဲ့ ပေးရပါတယ်။ ဒီလိုပေးလိုက်တဲ့ တန်ဖိုးတွေကို Property လို့ခေါ်ပြီး `props` ကနေတစ်ဆင့် ပြန်ယူသုံးလို့ရပါတယ်။ ဒီလိုပါ –

```
import React from 'react';

class Item extends React.Component {
    render() {
        return (
            <li>
                {this.props.name},
                ${this.props.price}
            </li>
        );
    }
}

class App extends React.Component {
    render() {
        return (
            <div>
                <h1>Hello React</h1>
                <ul>
                    <Item name="Apple" price="0.99" />
                    <Item name="Orange" price="0.89" />
                </ul>
            </div>
        )
    }
}

export default App;
```

App က Item Component ကို အသုံးပြုတဲ့အခါ `name` နဲ့ `price` ဆိုတဲ့ Property နှစ်ခုပေးထားသလို့ Item ကလည်း အဲ့ဒီ Property နှစ်ခုကို အသုံးပြုအလုပ်လုပ်ထားခြင်း ဖြစ်ပါတယ်။ JSX ရေးထုံးအရ HTML ထဲမှာ JavaScript Expression တွေ ထည့်ရေးချင်ရင် တွန့်ကွင်း အဖွင့်အပိတ်ထဲမှာ ရေးပေးရပါတယ်။ ဒါကြောင့် ဒီကုဒ်ရဲ့ ရလဒ်က အခုလိုဖြစ်မှာပါ –



Step 7 – state

နောက်ထပ် အရေးကြီးတဲ့ သဘောသဘာဝကတော့ `state` ဖြစ်ပါတယ်။ `state` ဆိုတာ Component အတွက် Data ပါ။ `state` မှာ သိမ်းထားတဲ့ Data တွေကို အသုံးပြုပြီး Component ကို ဖော်ပြစေနိုင်ပါတယ်။ ကုဒ်တွေများလာပြီမို့လို့ အကုန်လုံးကို ထပ်ခါထပ်ခါ ရေးမပြတော့ပါဘူး။ အပြောင်းအလဲ မရှိတဲ့ အပိုင်းတွေ ချန်ပြီး အပြောင်းအလဲရှိတဲ့အပိုင်းပဲ ရွေးထုတ်ပြပါတော့မယ်။ ဥပမာ –

```
class App extends React.Component {
    state = {
        items: [
            { id: 1, name: 'Apple', price: 0.99 },
            { id: 2, name: 'Orange', price: 0.89 },
        ]
    }

    render() {
        return (
            <div>
                <h1>Hello React</h1>
                <ul>
                    {this.state.items.map(i => {
                        return (
                  <Item
                      name={i.name}
                      price={i.price}
                  />
               )
                    })}
                </ul>
            </div>
        )
    }
}
```

နမူနာမှာ `state` လို့ခေါ်တဲ့ Class Field တစ်ခုပါဝင်လာပြီး၊ သူ့ရဲ့ `items` နေရာမှာ JSON Data Array တစ်ခုကို ပေးထားပါတယ်။ အဲ့ဒီ Array ကို `map()` နဲ့ Loop လုပ်ပြီး `<Item />` တွေကို ဖော်ပြထားတာကို တွေ့နိုင်ပါတယ်။ ဒီအထိ `state` ရဲ့ ထူးခြားချက်ကို မမြင်ရသေးပါဘူး။ ဒီအထိဆိုရင် `state` ဆိုတာ Variable တစ်ခုထက် မပိုသေးပါဘူး။

တစ်ခြား Variable တွေနဲ့ မတူဘဲ ထူးခြားတာကတော့ `state` တန်ဖိုးပြောင်းရင် Component က အလိုအလျောက်၊ ပြောင်းလဲသွားတဲ့ တန်ဖိုးနဲ့အညီ ဖော်ပြပေးသွားမှာ ဖြစ်ပါတယ်။ ဒါဟာ React ရဲ့ အဓိက အကျဆုံး Concept ပါ။ `state` ပြောင်းရင် Component က အလိုအလျောက် ပြောင်းလဲ ဖော်ပြပေးခြင်းပါပဲ။

လို - တို - ရှင်း ဆိုတဲ့အတိုင်း လိုရင်းကိုပဲ ပြောထားတာပါ။ ဒီအတိုင်းပဲ မှတ်ထားပေးပါ။ နောက်ကွယ်မှာ လေးနက်တဲ့ နည်းပညာသဘောသဘာဝတွေ ရှိနေပေမယ့် လိုရင်း အနှစ်ချုပ်ကတော့ ဒါပါပဲ။ အရေးကြီးလို့ ထပ်ပြောပါဦးမယ်။ `state` ပြောင်းရင် Component ရဲ့ ဖော်ပြပုံ လိုက်ပြောင်းပါတယ်။

Step 8 – Changing state

`state` ပြောင်းရင် Component ရဲ့ ဖော်ပြပုံပါ လိုက်ပြောင်းပုံကို နမူနာ စမ်းကြည့်ရအောင်။ ပထမဦးဆုံး အနေနဲ့ `App` Class အတွက် `add()` Method လေးတစ်ခု အခုလို ရေးပေးလိုက်ပါမယ်။

```
add = () => {
    let id = this.state.items.length + 1;

    this.setState({
        items: [
            ...this.state.items,
            { id, name: `Item ${id}`, price: 0.01 * id  }
        ]
    });
}
```

`state` ကို ပြင်ဖို့ `setState()` ကိုသုံးရခြင်း ဖြစ်ပါတယ်။ ပြောင်းစေလိုတဲ့ တန်ဖိုးကို ပေးရပါတယ်။ နမူနာမှာ `items` အတွက် Array တစ်ခုပေးထားပါတယ်။ နဂို `state` ထဲက `items` တွေကို Spread Operator အကူအညီနဲ့ အရင်ဖြန့်ထည့်လိုက် ပြီးမှ နောက်ကနေ ထပ်တိုးလိုတဲ့ Data ကို ထပ်တိုးထားတာ ဖြစ်ပါတယ်။ ဒါကြောင့် `add()` Method အလုပ်လုပ်တိုင်း `state` ရဲ့ `items` မှာ အသစ်အသစ် တိုးနေမှာ ဖြစ်ပါတယ်။

`add()` ဟာ Arrow Function တစ်ခုဖြစ်ပြီး၊ Property Shorthand ရေးထုံးကို ထည့်သုံးထားပါတယ်။ String Interpolation ရေးထုံးကို ထည့်သုံးထားတာလည်း သတိပြုပါ။ ဒီရေးထုံးတွေဟာ အရေးပါပါတယ်။ ဒီရေးထုံးတွေ မကြေညက်ရင် ကုဒ်နမူနာတွေက နားလည်ရ ခက်နေမှာပါ။ ဒါကြောင့် လိုအပ်ရင် အခန်း (၁) ကို သေသေချာချာ အရင်ပြန်လေ့လာထားပေးပါ။

တမင်ခွဲပြီး ရှင်းပြချင်လို့ `add()` ကုဒ်ကို အရင်ပြတာပါ။ `App` Class ရဲ့ လက်ရှိ ကုဒ်အပြည့်အစုံက ဒီလိုပါ

```
class App extends React.Component {
    state = {
        items: [
            { id: 1, name: 'Apple', price: 0.99 },
            { id: 2, name: 'Orange', price: 0.89 },
        ]
    }

    add = () => {
        let id = this.state.items.length + 1;
        this.setState({
            items: [
                ...this.state.items,
                { id, name: `Item ${id}`, price:  0.01 * id  }
            ]
        });
    }

    render() {
        return (
            <div>
                <h1>Hello React</h1>
                <ul>
                    {this.state.items.map(i => {
                        return (
                  <Item
                      name={i.name}
                      price={i.price}
                   />
                )
                    })}
                </ul>
                <button onClick={this.add}>Add</button>
            </div>
        )
    }
}
```

render() Method က Return ပြန်ပေးထားတဲ့ JSX ထဲမှာ <button> တစ်ခု ပါဝင်လာပြီး onClick မှာ add() ကို Assign လုပ်ထားတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။ Assign လုပ်ထားတယ်လို့ ပြောတာကို သတိပြုပါ။ <button onClick={**this.add()**}> လို့ရေးရင် React က လက်မခံပါဘူး။ Component ကို ဖော်ပြတိုင်း အဲ့ဒီ Method က Run နေမှာ မို့လို့ပါ။ <button onClick={**this.add**}> လို့ပဲ ရေးရပါတယ်။ ဒီတော့မှ Component ကိုဖော်ပြစဉ်မှာ add()

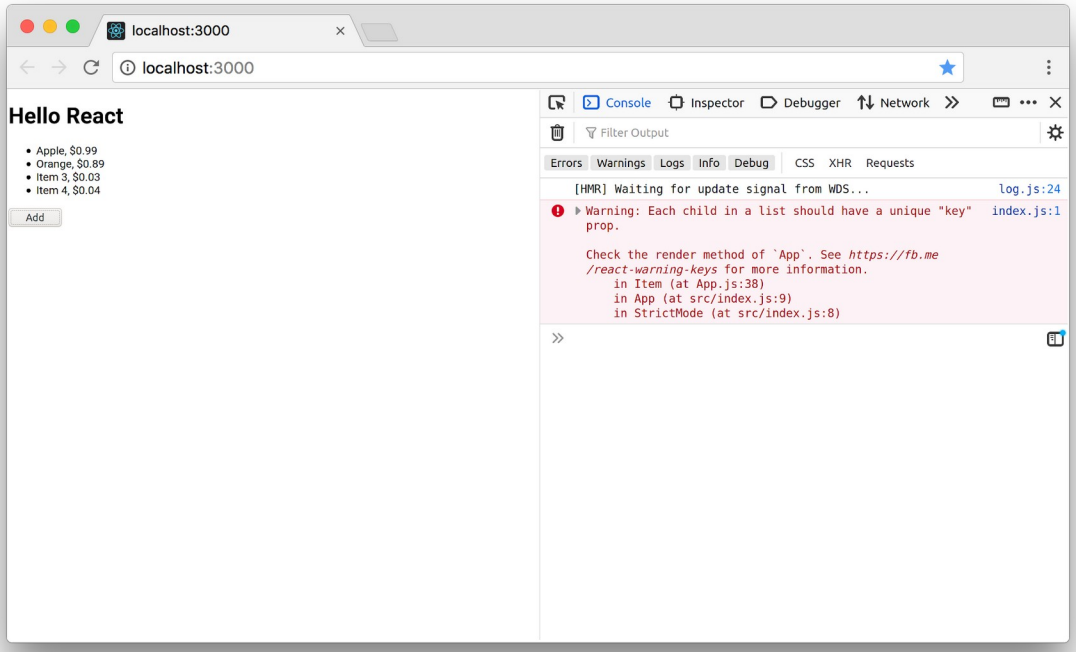
Method ကို `onClick` အတွက် သတ်မှတ်ယုံပဲ သတ်မှတ်ပြီး၊ Button ကို တစ်ကယ်နှိပ်လိုက်တော့မှ Run စေမှာဖြစ်ပါတယ်။ ဒါလေးက မျက်စိလည်ချင်စရာလေးမို့လို့ ဒီစာပိုဒ်ကို နောက်တစ်ခါ ပြန်ဖတ်ပေးပါ။

`<button onClick="this.add">` လို့ ရေးလို့မရတာကိုလည်း သတိပြုပါ။ Quote အဖွင့်အပိတ်နဲ့ မရေးပါဘူး။ တွန့်ကွင်း အဖွင့်အပိတ်နဲ့ ရေးပါတယ်။ ဒါလေးလည်းအရမ်းများတတ်ကြပါတယ်။ နောက်ထပ် သတိပြုစရာက ရိုးရိုး HTML မှာ Case Insensitive မို့လို့ `onclick`, `onClick` ကြိုက်သလို ရေးလို့ ရပါတယ်။ JSX မှာ မရပါဘူး။ `onClick` ပဲ ဖြစ်ရပါတယ်။

ဒီကုဒ်ကို စမ်းကြည့်တဲ့အခါ သတိထားကြည့်ပါ။ Button ကို နှိပ်လိုက်ရင် ဖော်ပြနေတဲ့ List ထဲမှာ Item အသစ်တိုးလာတာကို တွေ့ရပါလိမ့်မယ်။ ကျွန်တော်တို့ ရေးထားတဲ့ကုဒ်မှာ Item အသစ်တိုးပြီး List ကို ပြပေးလိုက်ပါဆိုတဲ့ကုဒ် မပါဝင်ပါဘူး။ `state` မှာ ပြောင်းလိုက်/တိုးလိုက်တဲ့ ကုဒ်ပဲပါပါတယ်။ `state` ပြောင်းသွားလို့ ပြောင်းသွားတဲ့ Data နဲ့အညီ React က အလိုအလျောက် ပြပေးသွားခြင်းပဲ ဖြစ်ပါတယ်။

Step 9 – key Property and Virtual DOM

အခုလက်ရှိရေးထားတဲ့ကုဒ်ကို Browser မှာစမ်းကြည့်တဲ့အခါ Console ကို ဖွင့်ကြည့်ပါ။ အခုလို Warning ကို တွေ့ရနိုင်ပါတယ်။



ဘာအဓိပ္ပါယ်လည်းဆိုတော့၊ Array ကို Loop လုပ်ပြီး Component ကို ဖော်ပြစေတဲ့အခါ `key` Property ပါဝင်သင့်ပါတယ်လို့ ပြောထားတာပါ။ ဘာကြောင့်လဲဆိုတာကို ရှင်းပြဖို့အတွက် Virtual DOM လို့ခေါ်တဲ့ နည်းစနစ်အကြောင်း နည်းနည်းပြောဖို့လိုပါတယ်။

Item (၅) ခုပါတဲ့ List တစ်ခု ရှိတယ် ဆိုကြပါစို့။ အရင်တုံးက စနစ်တွေမှာ Item အသစ်တစ်ခု တိုးလိုက်တယ်ဆိုတာ တစ်ကယ်တော့ Item (၆) ခုပါတဲ့ List တစ်ခုနဲ့ နဂို Item (၅) ခုပါတဲ့ List ကို အစားထိုးပစ်လိုက်ကြတာပါ။ ဆိုလိုတာက တစ်ခုခုပြင်လိုက်ရင် ပြင်တဲ့နေရာတင် ပြောင်းတာ မဟုတ်ဘဲ Component UI တစ်ခုလုံး ပြောင်းသွားတဲ့သဘော ရှိပါတယ်။

Virtual DOM ရဲ့ အလုပ်လုပ်ပုံကတော့၊ Item (၅) ခုပါတဲ့ List ရဲ့ Browser ပေါ်မှာ ဖော်ပြတဲ့ DOM Tree နဲ့ ပုံစံတူ Object တစ်ခုထုတ်ထားလိုက်တယ်။ အသစ်တစ်ခုတိုးလိုက်လို့ ဖော်ပြပုံ ပြောင်းရတော့မယ်ဆိုရင် ပြောင်းရမယ့် DOM Tree နဲ့ ပုံစံတူ Object တစ်ခု ထပ်ထုတ်တယ်။ အဲ့ဒီနှစ်ခု ဘယ်နေရာမှာ ဘာကွာလဲ တိုက်စစ်တယ်။ ပြီးတော့မှ ကွာသွားတဲ့ နေရာလေးတွေပဲ ရွှေးပြီးတော့ တစ်ကယ့် Browser မှာ ပြင်ပေးလိုက်တဲ့စနစ် မျိုးပါ။ ဆိုလိုတာက လုပ်စရာရှိတဲ့အလုပ်တွေကို Browser ပေါ်မှာ တိုက်ရိုက် မလုပ်ဘဲ JavaScript မှာပဲ အလုပ်လုပ်လိုက်တာ ဖြစ်သွားလို့ ပိုမြန်သွားမှာ ဖြစ်ပါတယ်။

ဒီထက်ပိုရှင်းချင်ရင်တော့ အရင်ကသုံးခဲ့ကြတဲ့ Template System တွေအကြောင်းနဲ့ Browser DOM Re-rendering ရဲ့ နှေးကွေးပုံတို့ကို ပြောရမှာပါ။ ထုံးစံအတိုင်း ဒီနေရာမှာ အဲ့ဒီလောက် အကျယ်မချဲ့ပါဘူး။ လိုရင်းကိုပဲ မှတ်ထားလိုက်ပါ။

`map()` နဲ့ Loop ပါတ်ပြီး ဖော်ပြထားတဲ့ ကုဒ်ကို ဒီလို ပြင်ပေးရမှာပါ။

```
<ul>
    {this.state.items.map(i => {
     return (
        <Item
           key={i.id}
           name={i.name}
           price={i.price}
        />
     )
    })}
</ul>
```

ဒီနည်းနဲ့ `key` Property ပါဝင်သွားသလို့ သူ့ရဲ့ တန်ဖိုးကလည်း Unique ဖြစ်သွားပါတယ်။ အဲ့ဒီလို `key` Property သာမပါခဲ့ရင် React က Virtual DOM လုပ်ဆောင်ချက်ကို အသုံးပြုပေးနိုင်မှာ မဟုတ်ဘဲ၊ တစ်ခုခု အပြောင်းအလဲရှိခဲ့ရင် Item အားလုံးကို အစအဆုံး တစ်ခေါက်ပြန်ဖော်ပြစေမှာ ဖြစ်ပါတယ်။ ဒါကြောင့် ဒီ `key` လေးထည့်လိုက်တာနဲ့ List တွေရဲ့ အလုပ်လုပ်ပုံ ပိုမြန်သွားမှာပဲ ဖြစ်ပါတယ်။

Step 10 – Input

ဆက်လက်ပြီး React မှာ Input တွေ စီမံပုံအကြောင်းကို ပြောပါမယ်။ React ကိုအသုံးပြုတဲ့ အခါမှာ HTML Element တွေကို တိုက်ရိုက်စီမံခြင်း မပြုရပါဘူး။ ဒါပေမယ့် Input တွေကတော့ ခြွင်းချက်နဲ့ လိုအပ်တဲ့အခါ တိုက်ရိုက်စီမံရပါတယ်။ ဒီလိုကိစ္စမျိုးအတွက် React မှာ `ref` လို့ခေါ်တဲ့ နည်းပညာတစ်ခု ပါဝင်ပါတယ်။ ဒီကုဒ် သုံးကြောင်းကို သီးခြားအရင် လေ့လာကြည့်ပါ။

```
nameRef = React.createRef();
<input type="text" ref={this.nameRef} />
let name = this.nameRef.current.value;
```

ပထမတစ်ကြောင်းက `React.createRef()` ကိုအသုံးပြုပြီး `nameRef` ကို တည်ဆောက်ပါတယ်။ နောက်တစ်ကြောင်းမှာ `<input />` Element ရဲ့ `ref` နေရာမှာ တည်ဆောက်ထားတဲ့ `nameRef` ကို ပေးလိုက်ပါတယ်။ `<input> = nameRef` လို့ ညွှန်းပေးလိုက်တဲ့ သဘောမျိုးပါ။ ဒီလိုညွှန်းပြီးပြီဆိုရင် `nameRef` ကို အသုံးပြုပြီး `<input>` ကို စီမံလို့ရသွားပါပြီ။ နောက်ဆုံးတစ်ကြောင်းက `nameRef` ကို အသုံးပြုပြီး `<input>` ရဲ့ value ကို ယူလိုက်တာပါ။ ဒီနည်းနဲ့ React မှာ Input တွေကို စီမံပါတယ်။ ရေးလက်စကုဒ်မှာ ဒီနည်းကို အခုလို ထည့်သွင်း အသုံးပြုလိုက်ပါ။

```
class App extends React.Component {
    state = {
        items: [
            { id: 1, name: 'Apple', price: 0.99 },
            { id: 2, name: 'Orange', price: 0.89 },
        ]
    }

    nameRef = React.createRef();
    priceRef = React.createRef();

    add = () => {
        let id = this.state.items.length + 1;
        let name = this.nameRef.current.value;
        let price = this.priceRef.current.value;

        this.setState({
            items: [
                ...this.state.items,
                { id, name, price }
            ]
        });
    }

    render() {
        return (
            <div>
                <h1>Hello React</h1>
                <ul>
                    {this.state.items.map(i => {
                        return (
                            <Item id={i.id} name={i.name} price={i.price} />
                        )
                    })}
                </ul>
                <input type="text" ref={this.nameRef} /><br />
                <input type="text" ref={this.priceRef} /><br />
                <button onClick={this.add}>Add</button>
            </div>
        )
    }
}
```

`render()` ရဲ့ Return ပြန်ပေးတဲ့ JSX မှာ `<input>` Element နှစ်ခု ထည့်သွင်းပေးပြီး `nameRef` နဲ့ `priceRef` တို့ကို အသုံးပြုထားပါတယ်။ ပြီးတဲ့အခါ `add()` Function မှာ အဲ့ဒီ Input တွေမှာ ရေးဖြည့်ထားတဲ့ တန်ဖိုးကို ယူပြီး အသုံးပြုသွားခြင်းပဲ ဖြစ်ပါတယ်။

တော်တော်ပြည့်စုံနေပါပြီ။ React ရဲ့ အခြေခံသဘောသဘာဝတွေလည်း စုံသလောက် ရှိနေပါပြီ။ ကျန်ရှိနေတဲ့ အကြောင်းအရာတွေကို နောက်တစ်ခန်း ခွဲပြီးတော့ ဆက်လေ့လာသွားကြရအောင်ပါ။

ဒီအခန်းမှာဖော်ပြခဲ့တဲ့ ကုဒ်တွေအပါအဝင် ဒီစာအုပ်မျာ နမူနာ ဖော်ပြထားတဲ့ ကုဒ်တွေအားလုံးကို အောက်ကလိပ်စာမှာ Download ရယူနိုင်ပါတယ်။

– https://github.com/eimg/react-book

အခန်း (၃) – React Data Flow

ပြီးခဲ့တဲ့အခန်းမှာ `props` နဲ့ `state` အကြောင်း လေ့လာခဲ့ကြပါတယ်။ ဒီအကြောင်းအရာတွေနဲ့ ပက်သက်ရင် သတိပြုရမယ့် အရေးကြီးတဲ့ အချက် (၅) ချက် ရှိပါတယ်။ နည်းနည်း ခေါင်းရှုပ်စရာလေးတွေ မို့လို့ သေချာ ဂရုစိုက်ဖတ်ကြည့်ပေးပါ။ လိုအပ်ရင် နှစ်ခါသုံးခါ ပြန်ဖတ်ပါ။

၁။ `props` ဟာ Read-only ဖြစ်ပါတယ်။ Component တွေဟာ `props` Data တွေကို အသုံးပြုလို့ ရပါတယ်။ ပြင်လို့ ပြောင်းလို့မရပါဘူး။

၂။ `state` ကတော့ ပြင်လို့ ပြောင်းလို့ ရပါတယ်။ `state` Data ပြောင်းရင် Component ဖော်ပြပုံ အလိုအလျောက် ပြောင်းလဲပုံကို လေ့လာခဲ့ကြပြီး ဖြစ်ပါတယ်။

၃။ Data ဟာ Parent to Child မြင့်ရာကနေ နိမ့်ရာကိုပဲ စီးဆင်းပါတယ်။ နိမ့်ရာကနေ မြင့်ရာကို ပြောင်းပြန်စီးဆင်းခြင်း မရှိပါဘူး။ ပြီးခဲ့တဲ့ နမူနာအရဆိုရင် Parent Component ဖြစ်တဲ့ `App` က Data တွေကို `props` အဖြစ်နဲ့ Child Component ဖြစ်တဲ့ `Item` ကို ပေးလို့ ရပါတယ်။ ဒါပေမယ့် `Item` Component က `App` Component ကို Data တွေ ပြန်ပေးလို့ မရပါဘူး။

၄။ Data ဟာ အဆင့်ဆင့်ပဲ လက်ဆင့်ကမ်းပြီး သွားလို့ရပါတယ်။ အဆင့်ကျော်လို့ မရပါဘူး။ ဥပမာ – App → List → Item ဆိုပြီး အဆင့်ဆင့် ရှိတယ်ဆိုရင် App က Item ကို အဆင့်ကျော်ပြီး Data ပေးလို့မရပါဘူး။ App က List ကို ပေးရပါတယ်။ List က လက်ဆင့်ကမ်း

ပြီးတော့ Item ကို ပေးလို့ပဲရပါတယ်။ ဒီသဘောသဘာဝကို မကြာခင် လက်တွေ့ စမ်းကြည့်ပါမယ်။ အဆင့်ကျော်ပြီး ရအောင်ပေးတဲ့ နည်းလည်း ရှိတော့ရှိပါတော့။ ဒါကိုတော့ နောက်တစ်ခန်းသပ်သပ်နဲ့ သီးခြား လေ့လာပါမယ်။

၅။ Child Component က Parent Component ရဲ့ Data ကို `props` Method တွေသုံးပြီး စီမံလို့ ရပါတယ်။ ဒါကိုတော့ အခုပဲ လက်တွေ့ကြည့်ကြပါမယ်။ ဆက်ကြည့်လိုက်ပါ။

`props` Methods

ပြီးခဲ့တဲ့နမူနာမှာ ရေးခဲ့တဲ့ ကုဒ်မှာ Input တွေ Button တွေကို သီးခြား Component အဖြစ် ခွဲထုတ်လိုက်ပါမယ်။ ဒီလို ရေးရမှာပါ။

```
class AddForm extends React.Component {
    nameRef = React.createRef();
    priceRef = React.createRef();

    render() {
        return (
            <div>
                <input type="text" ref={this.nameRef} /><br />
                <input type="text" ref={this.priceRef} /><br />
                <button onClick={this.add}>Add</button>
            </div>
        )
    }
}
```

App Component က ဒီအသစ်တည်ဆောက်လိုက်တဲ့ `AddForm` ကို ယူသုံးမှာပါ။ ပြဿနာက `AddForm` မှာပါတဲ့ `<button>` ရဲ့ `onClick` ကို သတိပြုကြည့်ပါ။ `add()` Method ကို အသုံးပြုထားပါတယ်။ သူ့မှာ `add()` Method မရှိပါဘူး။ မရှိလို့ ရေးလိုက်မယ်ဆိုရင်လည်း အဆင်မပြေသေးပါဘူး။ `add()` Method ဆိုတာ state → items မှာ Data အသစ် တိုးပေးရတာပါ။ သူ့မှာ state လည်း မရှိပါဘူး။ state → items အမှန်တကယ် ရှိနေတာက App Component မှာပါ။ သေချာစဉ်းစားကြည့်ပါ။ Child Component ဖြစ်တဲ့ `AddForm` က Parent Component ဖြစ်တဲ့ `App` ရဲ့ `state` ကို စီမံဖို့ လိုအပ်နေတာပါ။ ဒီ ပြဿနာကို ဖြေရှင်းလို့ရပါတယ်။ `App` Component ရဲ့ ကုဒ်ကို အခုလို ပြင်ပေးရမှာပါ။

```
class App extends React.Component {
    state = {
        items: [
            { id: 1, name: 'Apple', price: 0.99 },
            { id: 2, name: 'Orange', price: 0.89 },
        ]
    }

    add = (name, price) => {
        let id = this.state.items.length + 1;

        this.setState({
            items: [
                ...this.state.items,
                { id, name, price }
            ]
        });
    }

    render() {
        return (
            <div>
                <h1>Hello React</h1>
                <ul>
                    {this.state.items.map(i => {
                        return (
                            <Item
                                id={i.id}
                                name={i.name}
                                price={i.price}
                            />
                        )
                    })}
                </ul>
                <AddForm add={this.add} />
            </div>
        )
    }
}
```

ပထမဆုံး `add()` Method ကိုလေ့လာကြည့်ပါ။ Input စီမံတဲ့ကုဒ်တွေ မပါတော့ပါဘူး။ Parameter အနေနဲ့ `name` နဲ့ `price` တို့ကို လက်ခံပြီး အလုပ်လုပ်ထားပါတယ်။ နောက်ထပ် သတိပြုရမှာကတော့ `<AddForm>` ကို အသုံးပြုပုံပါ။ `add` Property အနေနဲ့ သူရဲ့ `add()` Method ကို ထည့်ပေးထားပါတယ်။ ဒါကြောင့် `AddForm` က `App` ရဲ့ `add()` Method ကို `add` props ကနေတစ်ဆင့် အသုံးပြုလို့ ရသွားပါပြီ။ `AddFrom` ရဲ့ ကုဒ် အပြည့်အစုံက ဒီလိုဖြစ်မှာပါ။

```
class AddForm extends React.Component {
    nameRef = React.createRef();
    priceRef = React.createRef();

    add = () => {
        let name = this.nameRef.current.value;
        let price = this.priceRef.current.value;

        this.props.add(name, price);
    }

    render() {
        return (
            <div>
                <input type="text" ref={this.nameRef} /><br />
                <input type="text" ref={this.priceRef} /><br />
                <button onClick={this.add}>Add</button>
            </div>
        )
    }
}
```

အခုတော့ AddForm မှာလည်း add() Method ရှိသွားပါပြီ။ သေချာဂရုစိုက်ကြည့်ပါ။ သူက this.props.add() ကို သုံးထားတယ်ဆိုတာ တွေ့ရပါလိမ့်မယ်။ ပြည့်စုံပါပြီ။ ဒီအတိုင်းစမ်းကြည့်ရင် အလုပ်လုပ်ပါတယ်။ ဒီနည်းနဲ့ Parent Component က Method တွေကို props အနေနဲ့ Child Component ကို ပေးလို့ရပါတယ်။ Child Component က လက်ခံရရှိတဲ့ props Method ကနေတစ်ဆင့် Parent Component ရဲ့ state ကို လှမ်းစီမံလို့ ရသွားပါတယ်။

ဒီသဘောသဘာဝကို ကောင်းကောင်းနားလည်ပြီဆိုရင် React ကို အတော်လေး ပိုင်နိုင်သွားပြီလို့ ပြောလို့ ရနိုင်ပါတယ်။

`props` Waterfall

နောက်ထပ်လေ့လာမှာကတော့ ဟိုးအပေါ်က နံပါတ် (၄) မှာ ပြောထားတဲ့ props ရဲ့ အဆင့်လိုက် စီးဆင်းပုံကို လေ့လာကြမှာပါ။ ပေးထားတဲ့ကုဒ်ကို လေ့လာကြည့်ပါ။ အောက်ကနေအပေါ်ကို ပြောင်းပြန်ကြည့်ပါ။

```
class Title extends React.Component {
    render() {
        return  <h1>{this.props.name}</h1>;
    }
}

class Header extends React.Component {
    render() {
        return (
            <div>
                <Title name={this.props.name} />
            </div>
        )
    }
}

class App extends React.Component {
    render() {
        return (
            <div>
                <Header name="App Title" />
                ...
            </div>
        )
    }
}
```

Component (၃) ခုပါဝင်ပါတယ်။ App, Header နဲ့ Title တို့ပါ။ App က Header ကို အသုံးပြုပြီး Header က Title ကို အသုံးပြုထားလို့ သူတို့ရဲ့ ဆက်စပ်မှုက App → Header → Title ဖြစ်ပါတယ်။ Title မှာ အသုံးပြုရမယ့် `name` ကို App က တစ်ဆင့်ကျော်ပြီး Title ကို လှမ်းပေးလို့ မရတဲ့အတွက် App ကနေ Header ကို ပေးပါတယ်။ Header ကနေမှ Title ကို ပေးထားတာကို သတိပြုကြည့်ရမှာပဲ ဖြစ်ပါတယ်။

React Component မှာ props Data တွေဟာ အခုလို တစ်ဆင့်ချင်းစီသာ အဆင့်ဆင့် လက်ဆင်ကမ်းပြီး ပေးသွားရပါတယ်။ အဆင့်ကျော်လို့ မရပါဘူး။ အဆင့်ကျော်ချင်ရင် Context လို နည်းပညာမျိုးကို သုံးရပါတယ်။ ဒီအကြောင်ကိုတော့ သီးခြား အခန်းတစ်ခန်းနဲ့ ဖော်ပြပေးသွားမှာပဲ ဖြစ်ပါတယ်။

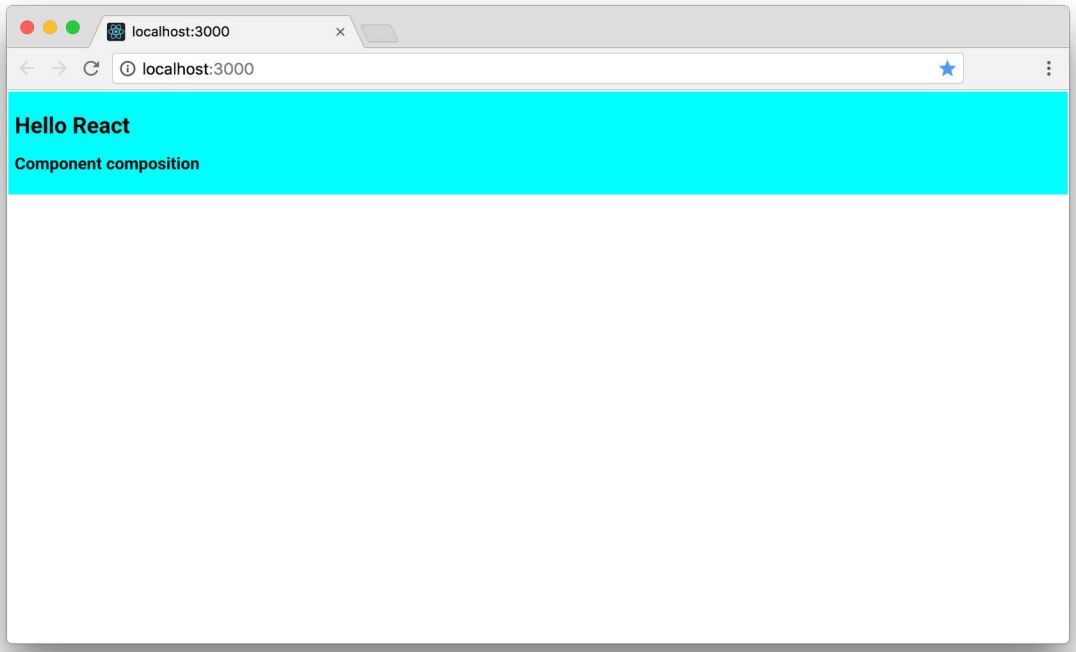
အခန်း (၄) – Composition and Code Splitting

Component Composition ဆိုတာ Component တွေကို လိုတဲ့နေရာက ခေါ်ယူ အသုံးပြုနိုင်ယုံသာမက ပေါင်းစပ်ပြီးတော့ပါ အသုံးပြုနိုင်တယ်ဆိုတဲ့ သဘောသဘာဝပါ။ ဒီကုဒ်ကို လေ့လာကြည့်ပါ။

```
class Toolbar extends React.Component {
    render() {
        return (
            <div style={{ background: 'cyan', padding: 10 }}>
                {this.props.children}
            </div>
        );
    }
}

class App extends React.Component {
    render() {
        return (
            <div>
                <Toolbar>
                    <h1>Hello React</h1>
                    <h2>Component composition</h2>
                </Toolbar>
            </div>
        )
    }
}
```

နမူနာမှာ Toolbar Component ကို App က ခေါ်သုံးထားပါတယ်။ ဒီအတိုင်းသုံးတာ မဟုတ်ပါဘူး။ Toolbar Component အတွင်းမှာ ရှိရမယ့် Element တွေကို ထည့်သွင်း သတ်မှတ်ပေးထားတာကို

သတိပြုပါ။ Toolbar Component ရေးထားပုံကို ပြန်လေ့လာပါ။ `props.children` လို့ခေါ်တဲ့ အထူး တန်ဖိုးတစ်ခုကို အသုံးပြုပြီး ပေါင်းစပ်သုံးဖို့ ပေးလာတဲ့ Element တွေကို အသုံးပြု ဖော်ပြထားတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။ ရလဒ်က ဒီလိုဖြစ်မှာပါ –



ဒီနည်းနဲ့ React မှာ Component တွေကို ပေါင်းစပ် အသုံးပြုနိုင်ပါတယ်။ လက်တွေ့ပရောဂျက်တွေမှာ ကုဒ်တွေရဲ့ဖွဲ့စည်းပုံက ဒီလိုမျိုး ဖြစ်နိုင်ပါတယ်။

```
<Toolbar>
  <Logo image="/path/to/image" />
  <Title>
     <Heading>App Title</Heading>
     <SubHeading>App Tag Line</SubHeading>
  </Title>
  <Menu>
     <MenuItem value="Home" />
     <MenuItem value="Users" />
  </Menu>
</Toolbar>
```

Code Splitting

အခုနမူနာတွေမှာ ကုဒ်အားလုံးကို `App.js` တစ်ဖိုင်ထဲမှာ အကုန်စုရေးထားတာပါ။ ဒီလိုအကုန်စုရေးမယ့် အစား လက်တွေ့မှာ Component တစ်ခုကို ဖိုင်တစ်ခုနဲ့ ခွဲရေးသင့်ပါတယ်။ ခွဲရေးပုံရေးနည်းက အသစ်အဆန်းတော့ မဟုတ်ပါဘူး။ ES6 Module ရေးထုံးကိုပဲ အသုံးပြုရမှာ ဖြစ်ပါတယ်။ ဥပမာ - Toolbar Component ကုဒ်တွေကို `Toolbar.js` ထဲမှာ အခုလို ရေးလို့ရနိုင်ပါတယ်။

```
import React from 'react';

class Toolbar extends React.Component {
    render() {
        return (
            <div style={{ background: 'cyan', padding: 10 }}>
                {this.props.children}
            </div>
        );
    }
}

export default Toolbar;
```

နောက်ဆုံးမှာ Toolbar ကို Export လုပ်ပေးထားတာလေး မမေ့ပါနဲ့။ တစ်ခြား သတိပြုစရာတွေ အနေနဲ့ Component ရဲ့အမည်ကို Capital Case နဲ့ ပေးရတယ် ဆိုတဲ့ အချက်နဲ့ Component ဖိုင်အမည်ဟာ Component အမည်နဲ့ တူသင့်တယ် ဆိုတဲ့ အချက်ပါပဲ။ မတူလည်း ဘာမှတော့ မဖြစ်ပါဘူး။ ဒါပေမယ့် တူ အောင်ပေးမှသာ Consistence ဖြစ်မှာပါ။ ဒီလို ဖိုင်ခွဲရေးထားတဲ့ Component ကို `App.js` မှာ အခုလို ခေါ်သုံးနိုင်ပါတယ်။

```
import React from 'react';
import Toolbar from './Toolbar';

class App extends React.Component {
    render() {
        return (
            <div>
                <Toolbar>
                    <h1>Hello React</h1>
                    <h2>Component composition</h2>
                </Toolbar>
            </div>
        )
    }
}

export default App;
```

Toolbar ကို Import လုပ်ပြီး ဆက်သုံးသွားယုံပါပဲ။ Import လုပ်တဲ့အခါ အမည်ကို တစ်ခြားအမည်နဲ့ ပြောင်းပေးမယ်ဆိုလည်း ရနိုင်ပါတယ်။ ဥပမာ –

```
import MyBar from './Toolbar';
```

ဒါဆိုရင် အသုံးပြုတဲ့အခါ <MyBar> လို့ သုံးပေးရမှာပါ။ ရတယ်လို့ပြောတာပါ၊ ကောင်းတာတော့ သူ့မူရင်း Component အမည်အတိုင်း အသုံးပြုနိုင်ရင် အကောင်းဆုံးပါပဲ။

ကုဒ်နမူနာတွေကို ရေးပြီးသားအသင့်လိုချင်ရင် https://github.com/eimg/react-book မှာ Download ရယူနိုင်ပါတယ်။

အခန်း (၅) – Component Style

React Component တွေရဲ့ Style အတွက် နည်းပညာအမျိုးမျိုး ရှိပါတယ်။ CSS Module, CSS in JS, Styled Component စသဖြင့် ရှိကြပါတယ်။ CSS Module လုပ်ဆောင်ချက်ကတော့ ပရောဂျက်ထဲမှာ ပါဝင်ပြီးသားပါ။ ဥပမာ - `Toolbar.css` အမည်နဲ့ အခုလို ရေးထားတယ်ဆိုကြပါစို့ -

```
.toolbar {
  background: cyan;
  padding: 10;
}
```

ဒီ CSS ကုဒ်ဖိုင်ကို Module တစ်ခုလို သဘောထားပြီး ES6 Import နဲ့ Import လုပ်ယူလို့ ရပါတယ်။ `Toolbar.js` ရဲ့ကုဒ်က ဒီလိုဖြစ်သွားမှာပါ။

```
import React from 'react';
import './Toolbar.css';

class Toolbar extends React.Component {
    render() {
        return (
            <div className="toolbar">
                {this.props.children}
            </div>
        );
    }
}

export default Toolbar;
```

CSS ဖိုင်ကို တိုက်ရိုက် Import လုပ်ထားတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။ သတိပြုစရာကတော့ ရိုးရိုး HTML မှာလို Element Class အတွက် `class` Attribute ကိုတော့ သုံးလို့ မရပါဘူး။ `className` လို့ သုံးပေးရပါတယ်။ `id` နဲ့ တစ်ခြား Attribute ကိုတော့ ရိုးရိုး HTML မှာလိုပဲ သုံးလို့ရပါတယ်။

CSS in JS

CSS in JS နည်းပညာကလည်း ပရောဂျက်ထဲမှာ ပါပြီးသားပါပဲ။ ရေးနည်း (၂) နည်းနဲ့ ရေးနိုင်ပါတယ်။ ပထမနည်းကတော့ Inline Style ကို အသုံးပြုခြင်းဖြစ်ပါတယ်။ ဒီလိုပါ –

```
render() {
    return (
        <div style={{ background: 'cyan', padding: 10 }}>
            {this.props.children}
        </div>
    );
}
```

`style` Attribute ကိုသုံးပြီးတော့ CSS Style တွေကို JSON အနေနဲ့ ပေးလိုက်တာပါ။ တွန့်ကွင်း နှစ်ထပ် ဆိုတာသတိပြုပါ။ ပထမတွန့်ကွင်းက JSX ကို JavaScript Expression ဖြစ်ကြောင်းသိစေပြီး ဒုတိယတွန့်ကွင်းက JSON အတွက်ပါ။

CSS Style Code တွေ ရေးတဲ့အခါ ရိုးရိုး CSS ရေထုံးအတိုင်း ရေးယုံပါပဲ။ ခြွင်းချက်တွေရှိပေမယ့် အများကြီးခေါင်းစားခံပြီး မှတ်မနေပါနဲ့ဦး။ သုံးခုမှတ်ထားရင် ရပါပြီ။ CSS Property တွေကို Camel Case နဲ့ ရေးရပါတယ်။ `px` Unit တွေ ထည့်ရေးစရာ မလိုပါဘူး။ JSON Format နဲ့ ရေးရတာဖြစ်လို့ Style Property တစ်ခုနဲ့တစ်ခုကို ရိုးရိုး CSS မှာလို Semi-colon နဲ့ မခြားဘဲ Comma နဲ့ ခြားပေးရတယ် ဆိုတဲ့အချက်တွေ မှတ်ထားရင် ရပါပြီ။ ဥပမာ ဒီလိုပါ –

```
render() {
    let parent = 200;
    let height = 150;
    return (
        <div style={{ marginBottom: parent - height,
              border: '1px solid red' }}>
            {this.props.children}
        </div>
    );
}
```

CSS မှာ `margin-bottom` လို့ရေးပေမယ့် CSS in JS မှာ `marginBottom` လို့ရေးပါတယ်။ `px` Unit ထည့်စရာ မလိုတဲ့အတွက် လိုအပ်ရင် တန်ဖိုးတွေကို ပေါင်းနှုတ်မြှောက်စား လုပ်လို့ရပါတယ်။ `border` အတွက်တော့ `1px solid red` လို့ ပေးထားပါတယ်။ `px` Unit ထည့်ပေးထားပါတယ်။ ပေးလို့ရပါတယ်။ Quote အဖွင့်အပိတ်ထဲမှာတော့ ဖြစ်ရပါတယ်။

CSS တွေ များလာတဲ့အခါ ပြင်ဆင်ထိမ်းသိမ်းရလွယ်အောင် ဒုတိယ ရေးနည်းအနေနဲ့ အခုလို ခွဲရေးထားသင့်ပါတယ်။

```
const styles = {
    toolbar: {
        marginBottom: 20,
        border: '1px solid red',
    }
}
```

ပြီးတော့မှ လိုတဲ့နေရာမှာ အခုလို ထည့်သုံးလိုက်တာ ပိုကောင်းပါတယ်။

```
render() {
    return (
        <div style={styles.toolbar}>
            {this.props.children}
        </div>
    );
}
```

style အတွက် တွန့်ကွင်း နှစ်ထပ်မလိုတော့တာကို သတိပြုပါ။ တစ်ခုထက်ပိုတဲ့ Style သတ်မှတ်ချက်ကို Component မှာ တွဲသုံးချင်လည်း ရပါတယ်။ ဥပမာ – Style က ဒီလိုသတ်မှတ်ထားတယ် ဆိုပါစို့။

```
const styles = {
    toolbar: {
        marginBottom: 20,
        border: '1px solid red',
    },
    dark: {
        background: 'purple',
        color: 'white',
    }
}
```

Component မှာ အခုလို အလွယ်တစ်ကူ ယူသုံးလိုက်လို့ ရပါတယ်။

```
render() {
    return (
        <div style={styles.toolbar, styles.dark}>
            {this.props.children}
        </div>
    );
}
```

တွဲသုံးချင်တဲ့ Style တွေကို Comma လေးခံပြီး ထည့်သွားလိုက်တာပါပဲ။

နောက်တစ်နည်းဖြစ်တဲ့ Styled Component ကတော့ ပရောဂျက်ထဲမှာ အသင့်မပါပါဘူး။ Package ထပ်ထည့်ပြီးမှ အသုံးပြုလို့ရမှာဖြစ်လို့ ဒီတစ်နည်းကိုတော့ ချန်ထားလိုက်ပါမယ်။ React နဲ့ပက်သက်ပြီး ကောင်းကောင်း ကျင်လည်လာပြီဆိုတော့မှ ကိုယ့်ဘာသာ ဆက်လေ့လာရမှာပါ။

အခန်း (၆) – Functional Components

ဟိုးရှေ့ပိုင်းမှာ ပြောထားပါတယ်။ React Component တစ်ခုတည်ဆောက်ဖို့ အခြေခံလိုအပ်ချက် အနေနဲ့ `React.Component` ကို Extend လုပ်ထားတဲ့ Class တစ်ခုဖြစ်ရမယ်၊ `render()` Method ပါရမယ်၊ Element တစ်ခုကို Return ပြန်ပေးရမယ်လို့ ပြောခဲ့ပါတယ်။ လက်တွေ့မှာ React Component တစ်ခုတည်ဆောက်ဖို့ နည်းလမ်း အမျိုးမျိုး ရှိပါတယ်။ ဒီတိုင်းရေးရင်တောင် ရပါတယ်။

```
const Header = <div><h1>Header</h1></div>;
```

ဒါလည်း လက်တွေ့အသုံးချလို့ရတဲ့ `<Header>` Component ဖြစ်သွားတာပါပဲ။ ဒီထက်ပိုအရေးပါတဲ့ နည်းလမ်းကတော့ Function ကို အသုံးပြုပြီး Component တွေ တည်ဆောက်တဲ့နည်း ဖြစ်ပါတယ်။ သူမှာလည်း လိုအပ်ချက် (၃) ချက် ရှိပါတယ်။

၁။ React Namespace အောက်မှာ ရှိရပါမယ်။
၂။ `props` ကို Argument အနေနဲ့ လက်ခံရပါမယ်။
၃။ Element တစ်ခုကို Return ပြန်ပေးရပါမယ်။

React Namespace အောက်မှာ ရှိရမယ်ဆိုတာ React ကို Import လုပ်ထားရမယ်လို့ ပြောတာပါ။ ဒီကုဒ်ကို လေ့လာကြည့်ပါ။

```
import React from 'react';

function Header(props) {
  return (
     <div>
        <h1>{props.name}</h1>
     </div>
  );
}

export default Header;
```

React ကို Import လုပ်ထားပါတယ်။ `Header` Function က `props` ကို လက်ခံထားပါတယ်။ ပြီးတော့ Element တစ်ခုကို Return ပြန်ပေးပါတယ်။ ဒါကြောင့် ဒါဟာ ပြည့်စုံတဲ့ React Component တစ်ခု ဖြစ်ပါတယ်။ `props` ကို အသုံးပြုတဲ့အခါ `this.props` လို့ပြောစရာမလိုဘဲ တိုက်ရိုက် သုံးထားတာကို သတိပြုပါ။ တစ်လက်စတည်း Export လုပ်ပုံလုပ်နည်း ဟာလည်း အတူတူပဲဆိုတာ သိစေချင်လို့ တစ်ခါတည်း ထည့်ပြီး Export လုပ်ပြထားပါတယ်။

ရိုးရိုး Function ကို မသုံးဘဲ Arrow Function ကို သုံးမယ်ဆိုရင်လည်း ရပါတယ်။ ရေးနည်းက ဒီလိုဖြစ်သွားမှာပါ။

```
const Header = props => {
  return (
     <div>
        <h1>{props.name}</h1>
     </div>
  );
}
```

ဒီထက်ကုဒ်ပိုတိုချင်ရင် ဒီတိုင်းရေးလည်း ရပါတယ်။

```
const Header = props => (
  <div>
     <h1>{props.name}</h1>
  </div>
);
```

Arrow Function မှာ တစ်ကြောင်းတည်းဆိုရင် တွန့်ကွင်းနဲ့ `return` မလိုဘူးလေ။ သုံးထားတာ တွန့်ကွင်း မဟုတ်ပါဘူး။ ဝိုက်ကွင်းပါ။ အဲ့ဒါလေးကို သေချာကြည့်ဖို့လိုပါတယ်။

Function Component တွေဟာ Functional Programming ကပေးတဲ့ အားသာချက်တွေကို ရပါတယ်။ အဲ့ဒါတွေအကြောင်း အသေးစိတ်တော့ ထည့်မပြောတော့ပါဘူး။ တစ်ခြားမြင်သာတဲ့ အားသာချက်ကတော့ ကုဒ်ရဲ့ တိုတောင်းကျစ်လစ်မှုပဲ ဖြစ်ပါတယ်။ ဒါကြောင့် ဒီရေးနည်းကို သိသွားပြီဆိုရင် Class တွေ မသုံးချင်တော့ဘဲ Function ကိုပဲ သဘောကျပြီး ဆက်သုံးသွားကြလေ့ ရှိပါတယ်။

React Basic အခန်းမှာ ပေးခဲ့တဲ့ ကုဒ်နမူနာကို Function တွေနဲ့ ပြောင်းရေးကြည့်ရအောင်။ `Item` Component ကို အရင်ရေးပါမယ်။ ဒီလိုပါ –

```
const Item = props => {
    return (
        <li>
            {props.name},
            ${props.price}
        </li>
    )
}
```

ဒီကုဒ်ကို ဒီထက်တိုသွားအောင် အခုလိုရေးလို့လည်း ရပါသေးတယ်။

```
const Item = ({ name, price }) => (
    <li>{name}, ${price}</li>
)
```

ကုဒ်ရဲ့ တိုတောင်းကျစ်လစ်မှုက ဘာမှမဆိုင်တော့ပါဘူး။ မလိုအပ်တဲ့ ဖွဲ့စည်းပုံကုဒ်တွေနဲ့ ရှုပ်နေစရာ မလိုတော့ပါဘူး။ Object Destructure ရေးနည်းကိုသုံးပြီး `props` ကို စလက်ခံကတည်းက `name` နဲ့ `price` အဖြစ် ဖြန့်ပြီးလက်ခံထားတာပါ။ ဒီရေးနည်းကို ပိုပြီးတော့ အသုံးပြုဖို့ တိုက်တွန်းပါတယ်။ ဆက်ရေးတဲ့ ကုဒ် တိုတောင်းကျစ်လစ် သွားယုံသာမက၊ ဒီ နမူနာကုဒ်ကို တစ်ချက်ကြည့်လိုက်တာနဲ့ `name` နဲ့ `price` လိုအပ်ပါလား ဆိုတာကို အလွယ်တစ်ကူ သိနိုင်လို့ ရေရှည်ပြုပြင်ထိမ်းသိမ်းရ လွယ်ကူတဲ့ကုဒ် ဖြစ်သွားစေမှာပါ။

App Component ကုဒ်ကိုတော့ ဒီလိုရေးရမှာပါ။

```
const App = props => {
    return (
        <div>
            <ul>
                <Item name="Apple" price="0.99" />
                <Item name="Orange" price="0.89" />
            </ul>
        </div>
    )
}
```

သူကိုတော့ တိုအောင် ထပ်မချဲ့တော့ပါဘူး။ တွန့်ကွင်းနဲ့ `return` နဲ့ အပြည့်အစုံပဲ ရေးထားပါတော့မယ်။ နောက်ထပ် ထပ်ဖြည့်ရမယ့် ကုဒ်တွေ ရှိသေးလို့ပါ။

`state` in Function Component

လက်ရှိဒီစာရေးနေချိန်မှာ နောက်ဆုံး React Version က 16.3.1 ဖြစ်ပါတယ်။ React 15.8.0 မတိုင်ခင်ထိ Function Component တွေမှာ `state` ကို အသုံးပြုလို့မရပါဘူး။ ဒါကြောင့် `state` အသုံးပြုဖို့လိုရင် Class Component ကိုပဲ သုံးရပါတယ်။ ဒါမှမဟုတ် Function Component ကို Redux လို State နည်းပညာနဲ့ တွဲသုံးရပါတယ်။ Redux အကြောင်းကို တစ်ခန်းသပ်သပ် နောက်မှ ပြောပါမယ်။

အခုချိန်မှာတော့ Hook လို့ခေါ်တဲ့ နည်းပညာတွေရဲ့ အကူအညီနဲ့ Function Component တွေမှာလည်း `state` ကို အသုံးပြုလို့ရပါပြီ။ `useState()`, `useReducer()`, `useEffect()` စသဖြင့် Build-in Hook Function တွေရှိပါတယ်။ အဲ့ဒီထဲက `useState()` ကိုအသုံးပြုပြီး `state` တွေစီမံပုံကို ဖော်ပြသွားပါမယ်။

ဒီကုဒ်ကို လေ့လာကြည့်ပါ။

```
let [ name, setName ] = React.useState('Bob');
let [ age, setAge ] = React.useState(22);
```

`React.useState()` Function က တန်ဖိုး (`state`) နဲ့ အဲ့ဒီတန်ဖိုးကို ပြင်လို့ရတဲ့ Function (`setState`) ကို ပြန်ပေးပါတယ်။ Array တစ်ခုအနေနဲ့ ပြန်ပေးလို့ Destructure လုပ်ပြီး လက်ခံရပါတယ်။ `state` နဲ့ `setState` လို့ အမည်ကိုပေးလို့ရသလို တစ်ခြား ကြိုက်တဲ့အမည်လည်း ပေးလို့ရပါတယ်။ ပေးထားတဲ့ နမူနာ ပထမတစ်ကြောင်းမှာ `name` ဆိုတဲ့ `state` တန်ဖိုးနဲ့ `setName()` ဆိုတဲ့ အဲ့ဒီတန်ဖိုးကို ပြင်လို့ရတဲ့ Function ကိုရပြီး၊ ဒုတိယ တစ်ကြောင်းမှာ `age` ဆိုတဲ့ `state` တန်ဖိုးနဲ့ `setAge()` ဆိုတဲ့ Function ကို ရပါတယ်။ `useState()` ကို ခေါ်ယူစဉ်မှာ ပေးလိုက်တဲ့ Argument က `state` အတွက် Default Value ဖြစ်သွားပါတယ်။ ဒါကြောင့် လက်ရှိ `name` ရဲ့တန်ဖိုးဟာ Bob ဖြစ်နေပြီး `age` ရဲ့တန်ဖိုးကတော့ 22 ဖြစ်နေပါတယ်။

`name` တန်ဖိုးပြောင်းချင်ရင် အခုလို အလွယ်တစ်ကူပြောင်းနိုင်ပါတယ်။

```
setName('Tom');       // => name = Tom
```

`age` ကို ပြောင်းချင်ရင်လည်း ဒီလိုပါပဲ၊ အလွယ်တစ်ကူ ပြောင်းနိုင်ပါတယ်။

```
setAge(23);     // => age = 23
```

ဒီနည်းကို အသုံးပြုပြီး App Component ကို ပြန်ရေးကြည့်ပါမယ်။ ဒီလိုပါ –

```
const App = props => {
    let [ items, setItems ] = React.useState([
        { id: 1, name: 'Apple', price: 0.99 },
        { id: 2, name: 'Orange', price: 0.89 },
    ]);

    return (
        <div>
            <ul>
              {items.map(i => <Item name={i.name} price={i.price} />)}
            </ul>
        </div>
    )
}
```

ဒီအဆင့်မှာ `useState()` ရဲ့ အကူအညီနဲ့ `items` နဲ့ `setItems()` ရသွားပါပြီ။ `items` အတွက် JSON Array တစ်ခုကိုလည်း Default Value အနေနဲ့ပေးထားပါတယ်။ ပြီးတဲ့အခါ `items` ကို `map()` လုပ်ပြီး ဖော်ပြစေပါတယ်။ `add()` Function တစ်ခု ရေးကြည့်ပါမယ်။ ဒီလိုဖြစ်မှာပါ။

```
let add = () => {
    setItems([
        ...items,
        { id: 3, name: 'Banana', price: 0.75 }
    ]);
}
```

`setItems()` ကို Class Component ရဲ့ `setState()` အသုံးပြုသကဲ့သို့ပဲ အသုံးပြုသွားတာပါ။ ဒါကို Input Ref တွေဘာတွေ အပြည့်အစုံနဲ့ ရေးလိုက်မယ်ဆိုရင် အခုလို ရပါလိမ့်မယ်။

```
import React, { createRef, useState } from 'react';

const Item = ({ name, price }) => (
    <li>{name}, ${price}</li>
)

const App = props => {
    let [ items, setItems ] = useState([
        { id: 1, name: 'Apple', price: 0.99 },
        { id: 2, name: 'Orange', price: 0.89 },
    ]);

    let nameRef = createRef();
    let priceRef = createRef();

    let add = () => {
        let id = items.length + 1;
        let name = nameRef.current.value;
        let price = priceRef.current.value;

        setItems([
            ...items,
            { id, name, price }
        ]);
    }
```

```
    return (
        <div>
            <ul>
                {items.map(i => (
                    <Item
                        key={i.id}
                        name={i.name}
                        price={i.price}
                    />
                ))}
            </ul>
            <input type="text" ref={nameRef} /><br />
            <input type="text" ref={priceRef} /><br />
            <button onClick={add}>Add</button>
        </div>
    )
}

export default App;
```

အစကနေအဆုံးပါတဲ့ ကုဒ်အပြည့်အစုံပါ။ ကူးယူရေးစမ်းကြည့်ရင် ဟိုး React Basic အခန်းမှာ ရေးခဲ့တဲ့ ကုဒ်နဲ့ တူညီတဲ့ရလဒ်ကို ရမှာပါ။ ရလဒ်တူသလို ရေးထားတဲ့ ကုဒ်ရဲ့ သဘောသဘာဝကလည်း ခပ်ဆင်ဆင်ပဲ ဖြစ်ပါတယ်။ Class တွေအစား Function တွေဖြစ်သွားတာပဲ ကွာပါတယ်။

ဟိုးအပေါ်ဆုံးက Import မှာ `createRef` နဲ့ `useState` ကို တစ်ခါတည်း Destructure လုပ်ပြီး Import လုပ်ထားတာကို သတိပြုပါ။ ဒီနည်းနဲ့ Import လုပ်တော့ ဆက်ရေးတဲ့ကုဒ်မှာ တိုပြီး ရှင်းသွားတာပေါ့။

အခန်း (၇) – Context

React Component တွေမှာ ပုံမှန်အားဖြင့် Data တွေဟာ Parent to Child လက်ဆင့်ကမ်းပြီး `props` အနေနဲ့ စီးဆင်းရတာကို ဖော်ပြခဲ့ပြီးဖြစ်ပါတယ်။ နောက်တစ်ခေါက်လောက် ပြန်ကြည့်ရအောင်ပါ။

```
const App = props => {
    return <Header name="Hello React" />
}

const Header = props => {
    return <Title name={props.name} />
}

const Title = props => {
    return <h1>{props.name}</h1>
}
```

App Component က Hello React ဆိုတဲ့ တန်ဖိုးကို Header Component ထံ `name` props အနေနဲ့ ပေးပါတယ်။ Header က လက်ခံရရှိတဲ့ `name` ကို Title Component ရဲ့ props အဖြစ် လက်ဆင့်ကမ်း ပေးပါတယ်။ နောက်ဆုံးမှာ Title Component က လက်ခံရရှိတဲ့ `name` props ကို အသုံးပြုပါတယ်။ ဒီလို တစ်ဆင့်ချင်းသာ သွားရပြီး App က Title အကို အဆင့်ကျော်ပြီး တိုက်ရိုက် Data ပေးလို့မရသလို၊ Title ကလည်း App ဆီက Data တွေကို အဆင့်ကျော်ပြီး လှမ်းယူလို့ မရပါဘူး။

Context ဆိုတာ လိုရင်းကတော့ Data တွေ အဆင့်ကျော် ပေးလို့ရအောင်၊ ယူလို့ရအောင် ဖန်တီးပေးထားတဲ့ နည်းပညာတစ်ခုပါ။ ပြီးခဲ့တဲ့ နမူနာကို Context သုံးပြီး အခုလို ရေးနိုင်ပါတယ်။

```
const MyContext = React.createContext();

const App = props => {
    return (
        <MyContext.Provider name="Hello React" />
            <Header />
        </MyContext.Provider>
    )
}

const Header = props => {
    return <Title />
}

const Title = props => {
    return (
        <MyContext.Consumer>
            { name => <h1>{name}</h1> }
        </MyContext.Consumer>
    )
}
```

ပထမဆုံးအနေနဲ့ App Component က `name` ကို Context Provider မှာပေးလိုက်ပါတယ်။ Header ကို မပေးတော့ပါဘူး။ Header ကလည်း Title ကို ဘာမှမပေးတော့ပါဘူး။ Title က Context Consumer ကို သုံးပြီးတော့ `name` ကို တိုက်ရိုက် ရယူလိုက်ခြင်းပဲ ဖြစ်ပါတယ်။ ဒါကြောင့် လိုအပ်တဲ့ Data ကို အဆင့်ဆင့် လက်ဆင့်ကမ်းနေဖို့ မလိုတော့ပဲ တိုက်ရိုက်ရရှိသွားပါပြီ။ နမူနာမှာက Component သုံးခုတည်းမို့လိုပါ။ လက်တွေ့မှာ Component ပေါင်းအဆင့်ဆင့် ဖြစ်လာမယ်ဆိုရင် ဒီနည်းက သိသိသာသာ အသုံးဝင်မှာပဲ ဖြစ်ပါတယ်။

ဒီကုဒ်ကို နောက်တစ်မျိုးပြောင်းရေးပြပါဦးမယ်။

```
const MyContext = React.createContext("Hello React");

const App = props => {
    return <Header />
}

const Header = props => {
    return <Title />
}

const Title = props => {
    const name = React.useContext(MyContext)
    return <h1>{name}</h1>
}
```

Context Provider တွေ Context Consumer တွေ မပါတော့ပါဘူး။ ပထမဆုံး `createContext()` မှာ ကတည်းက Default Value ပေးခဲ့ပါတယ်။ React က Context Provider မရှိရင် Default Value ကို သုံးပေးသွားမှာပါ။ Title မှာ Context ကို အသုံးပြုဖို့အတွက် Hook တွေထဲက တစ်ခုဖြစ်တဲ့ Context Hook ကို အသုံးပြုလိုက်လို့ Context Consumer ကိုလည်း သုံးစရာ မလိုတော့ပါဘူး။ ကုဒ်က နည်းနည်းပိုပြီး တော့ ရှင်းသွားပါတယ်။

အကယ်၍ Class Component တွေကို အသုံးပြုပြီး ရေးလိုတယ်ဆိုရင်လည်း အခုလို ရေးနိုင်ပါတယ်။

```
const MyContext = React.createContext("Hello React");

class App extends React.Component {
    render() {
        return <Header />
    }
}
class Header extends React.Component {
    render() {
        return <Title />
    }
}
class Title extends React.Component {
    static contextType = MyContext;

    render() {
        return <h1>{this.context}</h1>
    }
}
```

အသုံးပြုလိုတဲ့ Title Component မှာ `contextType` ကို Static Class Field အနေနဲ့ ကြေငြာပေးပြီးမှ သုံးရပါတယ်။ သုံးတဲ့အခါမှာ `this.context` ကနေ တစ်ဆင့်သာ အသုံးပြုရလို့ မျက်စိလည်ချင်စရာပါ။ Function Component ရဲ့ ရေးထုံးကတော့ ပိုပြီး နားလည်ရရော အသုံးပြုရပါလွယ်မယ်လို့ ထင်ပါတယ်။

အခန်း (၈) – Redux

Redux ဟာ State Container နည်းပညာလို့ ခေါ်ပါတယ်။ တနည်းအားဖြင့် `state` Data တွေကို စီမံတဲ့ နေရာမှာ အသုံးပြုရတဲ့ နည်းပညာပါပဲ။ ဒီနည်းပညာဟာ နားလည်သွားရင် ရိုးရှင်းပေမယ့်၊ အစပိုင်းမှာ နားလည်ရခက်ပြီး မျက်စိလည်စေနိုင်တဲ့ နည်းပညာတစ်ခုပါ။ အတတ်နိုင်ဆုံး ကြိုးစားပြီးတော့ ရှင်းအောင် ပြောသွားမှာ ဖြစ်ပါတယ်။

ပထမဦးဆုံး `state` Date တွေကို စီမံတဲ့ Function တစ်ခုအကြောင်း ပြောပါမယ်။ Reducer Function လို့ ခေါ်ပါတယ်။ Redux က `state` Data တွေကို စီမံမပေးပါဘူး။ ကိုယ်တိုင်ပဲ စီမံရတာပါ။ အဲ့ဒီလို စီမံနိုင်ဖို့ လိုအပ်တဲ့ Container Framework ကိုသာ ပေးထားတဲ့သဘောပါ။ Reducer Function ရဲ့ အခြေခံဖွဲ့စည်းပုံက ဒီလိုပါ။

```
function reducer(state, action) {
  return state;
}
```

နမူနာမှာ Function Name ကို `reducer` လို့ပေးထားပေမယ့် ဒီနာမည်က အရေးမကြီးပါဘူး။ ကြိုက်တဲ့ နာမည် ပေးလို့ရပါတယ်။ အရေးကြီးတာကတော့ `state` နဲ့ `action` ဆိုတဲ့ Parameter နှစ်ခုပါဖို့ပါပဲ။ မဖြစ်မနေ ပါဖို့လိုအပ်ပါတယ်။ နောက်ထပ်အရေးကြီးတာကတော့ `state` ကို Return ပြန်ပေးဖို့လိုအပ်ခြင်း ဖြစ်ပါတယ်။ `state` Data အကုန်လုံးကို ပြန်တာဖြစ်နိုင်တယ်၊ တစ်စိတ်တစ်ပိုင်းကို ပြန်တာ ဖြစ်နိုင်တယ်။ ဒါကတော့ ကိုယ် ဒီ Function ကို ဘယ်လို အလုပ်လုပ်စေချင်သလဲဆိုတဲ့ပေါ်မှာ မူတည်ပါတယ်။ နည်းနည်း ပြင်လိုက်ပါမယ်။

```
function reducer(state = [], action) {
  if(action.type === "ADD") {
     return [ ...state, action.name ];
  }

  return state;
}
```

state ရဲ့ Default Value ကို Array အလွတ်တစ်ခုလို့ သတ်မှတ်ထားတာကို သတိပြုပါ။ ပြီးတဲ့အခါ state ကို Return ပြန်ပုံပြန်နည်း ပြောင်းသွားပါပြီ။ action ရဲ့ Property type က ADD ဖြစ်ခဲ့မယ်ဆိုရင် မူလ state မှာ action ရဲ့ name ကို ပေါင်းထည့်ပြီးမှ Return ပြန်ပေးသွားတာ ဖြစ်ပါတယ်။ တစ်ခြား Delete, Update, Filter စတဲ့ ကုဒ်တွေကို ဒီနည်းအတိုင်း ဆက်ရေးသွားရမှာပါ။ ဒါတွေက Programming Logic တွေမို့လို့ ဒီနေရာမှာ နမူနာ ရေးမပြတော့ပါဘူး။ ကုဒ်ရှည်ပြီး ကြည့်ရခက် နားလည်ရခက်သွားမှာ စိုးလို့ပါ။ ဒီ Logic တွေကိုတော့ လက်ရှိသိထားတဲ့ ဗဟုသုတပေါ်မှာ မူတည်ပြီး ကိုယ်တိုင်ကြံဆပြီး ရေးရတော့မှာပါ။ ဒီနေရာမှာတော့ Redux ရဲ့ အလုပ်လုပ်ပုံကိုသာ ဆက်ရှင်းပြသွားမှာပါ။

Redux ရဲ့ createStore() လို့ ခေါ်တဲ့ Function ကို အသုံးပြုပြီး State Container တစ်ခုတည်ဆောက် နိုင်ပါတယ်။ Reducer Function ကို Argument အနေနဲ့ ပေးရပါတယ်။ ဒီလိုပါ။

```
const store = Redux.createStore(reducer);
```

ပြီးတဲ့အခါ dispatch() လို့ခေါ်တဲ့ Function တစ်ခုထပ်မှတ်ရပါမယ်။ State Container ထဲက state ကို ပြင်ချင်ရင် dispatch() နဲ့ ပြင်ရပါတယ်။ သူကိုယ်တိုင် ပြင်တာ မဟုတ်ပါဘူး။ သူက Reducer ကို သုံးပြီး အလုပ်လုပ်ပေးတာပါ။ တနည်းအားဖြင့် dispatch() ဆိုတာ reducer() ကို ခေါ်ပေးတဲ့ Function လို့ ဆိုနိုင်ပါတယ်။ dispatch() ကို ခေါ်တဲ့အခါ action ကို Argument အနေနဲ့ ထည့်ပေးရပါတယ်။ ဥပမာ –

```
store.dispatch({  type: "ADD", name: "Apple" });

// => state = [ Apple ]
```

```
store.dispatch({  type: "ADD", name: "Orange" });

// => state = [ Apple, Orange ]
```

```
store.dispatch({  type: "ADD", name: "Mango" });

// => state = [ Apple, Orange, Mango ]
```

`dispatch()` Function ကို ခေါ်စဉ်မှာ ပေးလိုက်တဲ့ `type: ADD` ကြောင့် `state` ထဲမှာ `name` တွေ တိုးတိုးသွားတဲ့ သဘောကို တွေ့ရမှာ ဖြစ်ပါတယ်။ `type: ADD` ဆိုရင် `state` ထဲမှာ `name` တိုးပြီး ပြန် ပေးဖို့ကို ကျွန်တော်တို့ကိုယ်တိုင် `reducer()` ထဲမှာ ရေးထားခဲ့တယ်လေ။

တစ်ကယ်တော့ ဒါပါပဲ။ Redux ရဲ့ အခြေခံ ပြီးသွားပါပြီ။ မျက်စိလည်နေသေးရင် နောက်တစ်ခေါက် လောက် ပြန်ဖတ်ကြည့်ပါ။ လက်တွေ့ စမ်းကြည့်လို့တော့ ရဦးမှာ မဟုတ်ပါဘူး။ စမ်းလို့ရတဲ့ ကုဒ် အပြည့်အစုံကို ပေးထားတာ မဟုတ်ဘဲ အလုပ်လုပ်ပုံ သဘောသဘာဝကို အရှင်းဆုံးနားလည်စေမယ့် နမူနာကို ပေးထားတာမို့လို့ပါ။

React & Redux

ဒီတစ်ခါတော့ လက်တွေ့စမ်းလို့ရတဲ့ ကုဒ်အပြည့်စုံကို နမူနာပေးပါတော့မယ်။ ပထမဆုံးအနေ `create-react-app` နဲ့ React ပရောဂျက် အသစ်တစ်ခု ဆောက်လိုက်ပါ။ ပြီးတဲ့အခါ သူ့ထဲမှာ NPM ရဲ့ အကူအညီနဲ့ `redux` နဲ့ `react-redux` တို့ကို Install လုပ်ပေးရပါမယ်။

```
>> create-react-app hello-redux
>> cd hello-redux
>> npm i redux react-redux
```

ပြီးတဲ့အခါ `index.js` ကို ဖွင့်ပြီး ဒီကုဒ်ကို ရေးပေးရပါမယ်။

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';

import { createStore } from 'redux';
import { Provider } from 'react-redux';

const store = createStore((state = [], action) => {
  if(action.type === "ADD") return [ ...state, action.item ];
  return state;
});

ReactDOM.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>,
  document.getElementById('root')
);

// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: https://bit.ly/CRA-PWA
serviceWorker.unregister();
```

အကုန်ပြန်ရေးစရာမလိုပါဘူး။ နမူနာမှာ Bold လုပ်ပေးထားတဲ့ အပိုင်းပဲ ဖြည့်ရမှာပါ။ ကျန်တာက နဂို ကတည်းက ပါလာတဲ့ ကုဒ်တွေပါ။ နည်းနည်း ရှင်းပြပါမယ်။ ပထမဆုံး `createStore` Function ကို `redux` က နေ Import လုပ်ယူပါတယ်။ ပြီးတဲ့အခါ Provider လို့ ခေါ်တဲ့ Component ကို `react-redux` ကနေ Import လုပ်ယူပါတယ်။ React မှာ Design Pattern တွေ ရှိပါတယ်။ Container Component ဆိုတာ လည်း တစ်ခုအပါအဝင် ဖြစ်ပါတယ်။ လေးနက်တဲ့ အကြောင်းအရာတွေမို့လို့ အသေးစိတ်ကိုတော့ နောက်မှ ဆက်လေ့လာပါ။ လောလောဆယ်မှာ ဒီလို အလွယ်မှတ်နိုင်ပါတယ်။ React Component တွေ တည်ဆောက်တဲ့အခါ အခုလို ပုံစံတည်ဆောက်ဖို့ကို အားပေးကြပါတယ်။

```
<Container>
  <UI />
</Container>
```

Container Component ထဲမှာ UI Component ကို ထားပါတယ်။ state အပါအဝင် Data နဲ့ပက်သက်တဲ့ ကိစ္စအားလုံးကို Container မှာပဲ လုပ်ပြီး UI မှာ အသွင်းအပြင်ပိုင်းပဲ ပါပါမယ်။ UI မှာ Data စီမံမှုကို လုံးဝ မလုပ်တဲ့ ရေးဟန် ဖြစ်ပါတယ်။ ဒီနည်းနဲ့ Data နဲ့ UI ကို ခွဲထုတ်လိုက်လို့ ပိုပြီးတော့ ပြုပြင်ထိန်းသိမ်းရ လွယ်ကူတဲ့ ကုဒ်ကို ရမှာဖြစ်ပါတယ်။ react-redux ကနေ ယူထားတဲ့ Provider ဆိုတာ Container Component တစ်မျိုး ဖြစ်ပါတယ်။ ဘယ်လိုသုံးထားလဲဆိုတာ အောက်နားမှာ ကြည့်လိုက်ပါ။

```
<Provider store={store}>
  <App />
</Provider>
```

ဒီသဘောပါပဲ။ store ဆိုတဲ့ Data ကို Provider ကပဲ စီမံမှာဖြစ်ပြီး App မှာ Data စီမံတဲ့အလုပ်ကို မလုပ်တော့ဘူးဆိုတဲ့ သဘောပါ။ ဒီနေရာမှာ store ဆိုတာလေး လိုလာလို့ သူ့အပေါ်မှာ createStore() Function ရဲ့ အကူအညီနဲ့ store တစ်ခုတည်ဆောက်ထားတာကို တွေ့နိုင်ပါတယ်။ Reducer Function ကို သပ်သပ်ရေးမနေတော့ပဲ createStore() ထဲမှာပဲ Arrow Function နဲ့ တစ်ခါတည်းရေးထားပါတယ်။ ပြီးတဲ့အခါ App.js မှာ ရေးရမယ့်ကုဒ်က ဒီလိုပါ။

```
import React, { createRef } from 'react';
import { connect } from 'react-redux';


const Item = ({ name, price }) => (
    <li>{name}, ${price}</li>
)


const App = props => {
    let nameRef = createRef();
    let priceRef = createRef();

    const add = () => {
        props.add(
     props.items.length + 1,
     nameRef.current.value,
     priceRef.current.value
      );
    }
```

```
    return (
        <div>
            <ul>
                {props.items.map(i => (
           <Item
              key={i.id}
              name={i.name}
              price={i.price}
           />
         )}
            </ul>
            <input type="text" ref={nameRef} /><br />
            <input type="text" ref={priceRef} /><br />
            <button onClick={add}>Add</button>
        </div>
    )
}

const stateToProps = state => {
    return {
        items: state
    };
}

const dispatchToProps = dispatch => {
    return {
        add: (id, name, price) => {
            dispatch({
        type: 'ADD',
        item: { id, name, price }
        });
        }
    }
}

const ReduxApp = connect(stateToProps, dispatchToProps)(App);

export default ReduxApp;
```

ရှိသမျှကုဒ်အကုန်ပြထားလို့ မျက်စိနောက်သွားနိုင်ပေမယ့် ကုဒ်အများစုက အရင်ရေးနေကြကုဒ်နဲ့ သိပ်မကွာပါဘူး။ ထူးခြားချက်တွေကို ရွေးထုတ်ပြောပြပါမယ်။ ပထမဆုံး `connect()` Function ကို `react-redux` ဆီကနေ Import လုပ်ထားပါတယ်။ `connect()` Function ဟာ Higher Order Function လို့ ခေါ်တဲ့ Function တစ်မျိုးပါ။ လွယ်လွယ်မှတ်ချင်ရင် Function တစ်ခုကို Return ပြန်ပေးတဲ့ Function လို့ မှတ်နိုင်ပါတယ်။ ဥပမာလေးတစ်ခု ပေးပါမယ် –

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

နမူနာမှာ ပေးထားတဲ့ `add()` Function ဟာ Higher Order Function တစ်ခုပါ။ သူက Function တစ်ခုကို Return ပြန်ပေးထားပါတယ်။ အခုနေ `add(2)` လို့ခေါ်လိုက်ရင် ပြန်ရမှာက –

```
function (b) {
  return 2 + b;
}
```

ဆိုတဲ့ Function ကို ပြန်ရမှာပါ။ ဒါကြောင့် အပြည့်အစုံရေးမယ်ဆိုရင် အခုလို ဖြစ်နိုင်ပါတယ်။

```
const addTwo = add(2);

addTwo(3)       // => 5
```

`add()` Function ကို ခေါ်လိုက်လို့ ပြန်ရလာတဲ့ `addTwo()` Function ကို နောက်တစ်ခါ ပြန်ခေါ်ပြထားတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။ အဲ့ဒီလို နှစ်ကြောင်းမရေးချင်ရင် ဒီလိုရေးလို့ ရပါတယ်။

```
add(2)(3)       // => 5
```

`add(2)` ကပြန်ပေးတဲ့ Function ကို (3) ချက်ခြင်းထည့်ပေးလိုက်တာ ဖြစ်ပါတယ်။ အခုပြောနေတဲ့ connect() Function ဟာ ဒီလို Function အမျိုးအစားပါ။ သူကိုခေါ်တဲ့အခါ ဒီလိုခေါ်ရပါတယ်။

```
connect(function(state), function(dispatch))(Component)
```

`connect()` Function ကို ခေါ်တဲ့အခါ Argument နှစ်ခုပေးပြီးတော့ ခေါ်ရတာပါ။ နှစ်ခုလုံးက

Function တွေ ဖြစ်ရပါမယ်။ ပထမ Function က state Data ကို props အဖြစ်ပြောင်းပေးတဲ့ Function ဖြစ်ရပါမယ်။ ဒုတိယ Function ကတော့ dispatch() ကို အသုံးပြုပြီး state ကို စီမံတဲ့ props တွေ ဖန်တီးပေးတဲ့ Function ဖြစ်ရပါမယ်။ ခေါင်းမူးသွားပြီထင်ပါတယ်။ မရပ်ဘဲ နည်းနည်းဆက် ဖတ်ပေးပါ။

သူ့အပေါ်နားမှာ ရေးထားတဲ့ stateToProps() Function ကိုလေ့လာကြည့်ပါ။ props တွေကို Return ပြန်ပေးထားတာကို တွေ့ရပါမယ်။ နမူနာအရ items props ရဲ့တန်ဖိုးအဖြစ် state ထဲမှ ရှိသမျှ ကို ပေးလိုက်တာ ဖြစ်ပါတယ်။

နောက် Function တစ်ခုဖြစ်တဲ့ dispatchToProps() ကို ဆက်လေ့လာကြည့်ပါ။ သူလည်းပဲ props တွေကို Return ပြန်ပေးတာပါပဲ။ သူမှာတော့ add props ရဲ့ တန်ဖိုးအဖြစ် dispatch() Function ကို ပြန်ပေးထားပါတယ်။

connect() Function ကိုခေါ်ထားတဲ့ လက်တွေ့ကုဒ်က ဒီလိုပါ။

```
const ReduxApp = connect(stateToProps, dispatchToProps)(App);
```

connect() Function ကို ခေါ်ပြီး ပြန်ရလာတဲ့ Function ကိုတော့ App Component ပေးလိုက်ပါ တယ်။ ဒါကြောင့် connect() Function က App Component မှာ items နဲ့ add ဆိုတဲ့ props နှစ် ခုကို တွဲပေးလိုက်မှာပါ။ items က stateToProps ကနေ ရပါတယ်။ add က dispatchToProps ကနေ ရပါတယ်။

ဒါကြောင့် App Component ကုဒ်ကို လေ့လာကြည့်လိုက်ရင် props.items ကို map() လုပ်ပြီး အသုံးပြု ထားတာကို တွေ့ရမှာဖြစ်ပါတယ်။ ပြီးတဲ့အခါ add() Function က props.add() ကို ခေါ် သုံးသွားတာကို တွေ့ရနိုင်ပါတယ်။

နောက်ဆုံးအနေနဲ့ သတိပြုရမှာကတော့ Export လုပ်ထားတာ App မဟုတ်ပါဘူး။ လိုအပ်တဲ့ props တွေ ကို တွဲထားပြီးဖြစ်တဲ့ ReduxApp ကို Export လုပ်ပေးထားတာပါ။

ပြောမယ်ဆိုရင် နည်းနည်းကျန်သေးပေမယ့် ဒီအဆင့်မှာ ဒီလောက်နဲ့ပဲ ရပ်ထားကြပါစို့။ ဆန်းကြယ်တဲ့ ရေးဟန်တွေကို အသုံးပြုထားတဲ့ နည်းပညာဖြစ်နေလို့ အချိန်တိုအတွင်း အရမ်းစွတ်သွားလိုက်ရင် ခေါင်းတွေမူး၊ မျက်စိတွေလည်ပြီး ဘာမှသေချာ မရလိုက် ဖြစ်သွားပါမယ်။ အခုပြောပြထား သလောက်ကို ကောင်းကောင်း သဘောပေါက်အောင်သာ အခါခါ ပြန်ကြည့်ပါ။ ရတယ်ဆိုရင် နောက်အဆင့်တွေက ဆက်လုပ်ရတာ လွယ်သွားပါလိမ့်မယ်။

ထုံးစံအတိုင်း ရေးပြီးသားကုဒ်အပြည့်အစုံလိုချင်ရင် https://github.com/eimg/react-book မှာ ယူလို့ရပါတယ်။

အခန်း (၉) – React Router

လက်တွေ့ပရောဂျက်တွေမှာ Screen (သို့မဟုတ်) Page တွေ အများကြီးပါဝင်နိုင်ပါတယ်။ ဥပမာ – Login, Register, Profile, Home, Dashboard, Setting စသဖြင့်ပေါ့။ React မှာလည်း Component တွေကို အဲ့ဒီလို Page တွေခွဲပြီး စီမံလို့ရပါတယ်။ ဒီအတွက် React Router လို နည်းပညာမျိုးကိုသုံးနိုင်ပါတယ်။ ဥပမာ – အခုလို Component နှစ်ခုရှိတယ်ဆိုကြပါတယ်။

```
const users = [
    { id: 1, name: 'Alice', gender: 'f' },
    { id: 2, name: 'Bob', gender: 'm' },
    { id: 3, name: 'Tom', gender: 'm' },
    { id: 4, name: 'Mary', gender: 'f' },
];
```

```
const Male = props => {
    return (
        <ul>
            {users.filter(u => u.gender === 'm')
            .map(u => <li key={u.id}>{u.name}</li>)}
        </ul>
    );
}
```

```
const Female = props => {
    return (
        <ul>
            {users.filter(u => u.gender === 'f')
           .map(u => <li key={u.id}>{u.name}</li>)}
        </ul>
    );
}
```

ဒီ Component နှစ်ခုကို အသွားအပြန် Navigate လုပ်လို့ရတဲ့ ခလုပ်လေးတွေနဲ့ Page ခွဲပြီး ပြချင်တယ်ဆိုရင် ပြလို့ရပါတယ်။ ပထမဆုံးအနေနဲ့ `react-router-dom` ကို NPM နဲ့ Install လုပ်လိုက်ပါ။

```
>> npm i react-router-dom
```

ပြီးရင်သူ့ထဲက ဒီလုပ်ဆောင်ချက်တွေကို Import လုပ်ယူရမှာပါ။

```
import {
    BrowserRouter as Router,
    Switch,
    Route,
    Link,
} from "react-router-dom";
```

BrowserRouter ကို Router လို့ အမည်ပြောင်းပေးပြီး Import လုပ်ထားပါတယ်။ Page တွေ ခွဲပြချင်တဲ့ App က အဲ့ဒီ `<Router>` Component အတွင်းမှာ ရှိရမှာ ဖြစ်ပါတယ်။ Route ဆိုတာကတော့ ပြချင်တဲ့ Page တွေပါပဲ။ `<Route>` နှစ်ခုရှိရင် Page နှစ်ခုရှိတယ်ဆိုတဲ့ သဘောပါ။ Switch ကတော့ ရှိနေတဲ့ Route တွေထဲက တစ်ကြိမ်မှာ တစ်ခုပဲပြအောင် စစ်ပေးတဲ့ Component ပါ။ ဒါကြောင့် တစ်ကြိမ်မှာ တစ်ခုပဲ ပြစေချင်တဲ့ `<Route>` တွေအားလုံးကို `<Switch>` ထဲမှာ ထားပေးရပါတယ်။ Link ကတော့ Navigation ခလုပ်အတွက်ပါ။ ဒီ Component တွေကိုသုံးပြီး App Component ကို အခုလို ရေးနိုင်ပါတယ်။

```
const App = props => {
    return (
        <Router>
            <div>
                <ul>
                    <li><Link to="/male">Male</Link></li>
                    <li><Link to="/female">Female</Link></li>
                </ul>
                <div style={{background: 'cyan', padding: 20}}>
                    <Switch>
                        <Route path="/male"><Male /></Route>
                        <Route path="/female"><Female /></Route>
                    </Switch>
                </div>
            </div>
        </Router>
    );
}
```

လေ့လာကြည့်ပါ။ အသုံးပြုရ လွယ်ကူပါတယ်။ သိပ်ပြီးတော့ ရှုပ်ထွေးမှု မရှိလှပါဘူး။ App Component တစ်ခုလုံးက `<Router>` အတွင်းမှာ ရှိပါတယ်။ နှိပ်လို့ရတဲ့ Navigation ခလုပ်ဖြစ်တဲ့ `<Link>` နှစ်ခု ပါဝင်ပြီး တစ်ခုရဲ့လိပ်စာကို `/male` လို့ ပေးထားပါတယ်။ သူ့ကို နှိပ်လိုက်ရင် –

http://localhost:3000/male

ဆိုတဲ့ လိပ်စာကို သွားပေးမှာပါ။ နောက် `<Link>` တစ်ခုကလည်း အလားတူပါပဲ။ `/female` ဆိုတဲ့ လိပ်စာကို သွားပေးတဲ့ ခလုပ်ပါ။

`<Switch>` ထဲမှာ `<Route>` နှစ်ခုရှိပါတယ်။ တစ်ခုက လိပ်စာ `/male` ဆိုရင် ပြဖို့ပါ။ နောက်တစ်ခုက `/female` ဆိုရင်ပြဖို့ပါ။ ရေးထားတဲ့ကုဒ်အရ `/male` ဆိုရင် `<Male>` Component ကို ဖော်ပြပြီး `/female` ဆိုရင်တော့ `<Female>` Component ကို ဖော်ပြပေးမှာပါ။ ဒီနည်းနဲ့ Component တွေကို စာမျက်နှာတွေ ခွဲပြီး ရေးသားစီမံနိုင်ပါတယ်။

ပေးထားတဲ့ နမူနာအရ လိပ်စာက `/male` ဆိုရင် `<Male>` Component ဖော်ပြသလို `/male-users`, `/male/123`, `/male/users` ဘယ်လိုပဲလာလာ လိပ်စာက `/male` နဲ့စနေသ၍ `<Male>`

Component ကိုပဲ ပြပေးမှာပါ။ အဲ့ဒီလိုမဟုတ်ဘဲ `/male` ဆိုတဲ့လိပ်စာအတိအကျဖြစ်မှ `<Male>` Component ကို ပြစေချင်ရင် သူရဲ့ `<Route>` ကို အခုလိုရေးပေးရမှာပါ။

```
<Route path="/male" exact><Male /></Route>
```

`exact` Property ပါသွားတာပါ။

Dynamic URL

တစ်ချို့ အခြေအနေပေါ်မူတည်ပြီး တန်ဖိုးပြောင်းလဲနေတဲ့ URL တွေကို အသုံးပြုဖို့ လိုအပ်ရင်လည်း သုံးနိုင်ပါတယ်။ `/user/1`, `/user/2`, `/user/3` စသဖြင့် လုပ်ဆောင်ချက်က အတူတူပဲ၊ 1, 2, 3 စသဖြင့် Parameter တန်ဖိုးပဲ ပြောင်းသွားတဲ့သဘောပါ။ ဒီကုဒ်ကိုလေ့လာကြည့်ပါ။

```
const User = props => {
    const { name } = useParams();

    return (
        <h1>Profile - {name}</h1>
    )
}

const App = props => {
    return (
        <Router>
            <div>
                <ul>
                    <li><Link to="/user/Alice">Alice</Link></li>
                    <li><Link to="/user/Bob">Bob</Link></li>
                </ul>
                <div style={{background: 'cyan', padding: 20}}>
                    <Switch>
                        <Route path="/user/:name"><User /></Route>
                    </Switch>
                </div>
            </div>
        </Router>
    );
}
```

App Component မှာ `<Link>` နှစ်ခုပါပေမယ့် `<Route>` Component က တစ်ခုပဲ ရှိပါတယ်။ ထူးခြားချက်အနေနဲ့ `<Route>` ရဲ့ path မှာ `:name` ဆိုတဲ့ Parameter တစ်ခုပါဝင်ပါတယ်။ `:name` နေရာမှာ ပါဝင်လာတဲ့ တန်ဖိုးတွေ ပြောင်းလဲနေရင်လည်း လက်ခံအလုပ် လုပ်ပေးနိုင်တဲ့ Route ဖြစ်သွားပါတယ်။ ဒါကြောင်း Link နှစ်ခုမှာ ပေးထားတဲ့ `/user/Alice` ရာ `/user/Bob` ရော နှစ်ခုလုံးအတွက် ဒီ Route က အလုပ်လုပ်မှာ ဖြစ်ပါတယ်။

User Component ကိုလေ့လာကြည့်လိုက်ပါ။ `useParams()` လို့ ခေါ်တဲ့ Hook ရဲ့ အကူအညီနဲ့ URL ထဲက `:name` တန်ဖိုးကို ယူထားပါတယ်။ ဒီနည်းနဲ့ URL မှာပါဝင်လာတဲ့ ပြောင်းလဲနေတဲ့ Parameter တန်ဖိုးတွေကို Component က ရယူအလုပ်လုပ်နိုင်သွားပါတယ်။ Import ကုဒ်ကို နမူနာထပ်မပြတော့ပေမယ့် `useParams` ကိုလည်း ထည့် Import လုပ်ရလိမ့်မယ်ဆိုတာကို သတိပြုပါ။

လိုအပ်ရင်တော့ ကုဒ်အပြည့်အစုံကို ဒီကယူကြည့်ပါ – https://github.com/eimg/react-book

အခန်း (၁၀) – React Native

React Native ဟာ React ကို အသုံးပြုထားတဲ့ Cross-platform Development နည်းပညာတစ်ခုပါ။ Mobile အတွက်ရော Desktop အတွက်ပါ သုံးကြပါတယ်။ React Native ကိုသုံးပြီး Android App တွေ iOS App တွေ Windows App တွေ ရေးလို့ရပါတယ်။ ရေးတဲ့အခါ React (JavaScript) နဲ့ ရေးရလို့ ရေးရတာ မြန်သလို၊ လက်တွေ့ အလုပ်လုပ်တဲ့အခါ Native UI တွေကို အသုံးပြုပြီး အလုပ်လုပ်လို့ စွမ်းဆောင်ရည် အမြန်နှုန်းလည်း ကောင်းပါတယ်။ ကြိုက်တဲ့သူ ရှိသလို မကြိုက်တဲ့သူတွေလည်း ရှိပါတယ်။ ဒီနေရာမှာ အကြောင်း အကျိုး အကောင်း အဆိုးတွေကို နှိုင်းယှဉ်ပြီး ပြောမနေတော့ပါဘူး။ ပေရှည်ပါလိမ့်မယ်။ ဘယ်လိုသုံးရလဲ၊ ဘယ်လိုရေးရလဲ ဆိုတာကိုပဲ လိုတိုရှင်း ပြောပြသွားမှာပါ။

React ပရောဂျက်တွေကို `create-react-app` အသုံးပြု တည်ဆောက်ရသလို့ React Native ပရောဂျက်တွေကိုတော့ `react-native` (သို့မဟုတ်) Expo လို့ခေါ်တဲ့ နည်းပညာကို အသုံးပြု တည်ဆောက်ရပါတယ်။ Mobile App တွေ ရေးရ စမ်းရတာ နည်းနည်း အလုပ်များပါတယ်။ စက်ထဲမှာ သက်ဆိုင်ရာ SDK တွေ Build Tool တွေ ရှိမှ စမ်းလို့ရကြပါတယ်။ Expo လို့ခေါ်တဲ့ နည်းပညာက React Native ပရောဂျက်တွေကို ကိုယ့်စက်ထဲမှာ တစ်ခြားဘာမှ ရှိစရာမလိုဘဲ စမ်းလို့ရအောင် လုပ်ပေးထားလို့ အတော်အဆင်ပြေပါတယ်။ React Native Documentation ကလည်း Expo ကို ဦးစားပေး ဖော်ပြထားသလို ဒီနေရာမှာလည်း Expo ကိုပဲ အသုံးပြုပြီး ရှေ့ဆက်သွားကြမှာပါ။ ပထမဆုံးအနေနဲ့ `expo-cli` ကို NPM နဲ့ Install လုပ်ပေးရပါမယ်။

```
>> npm I expo-cli
```

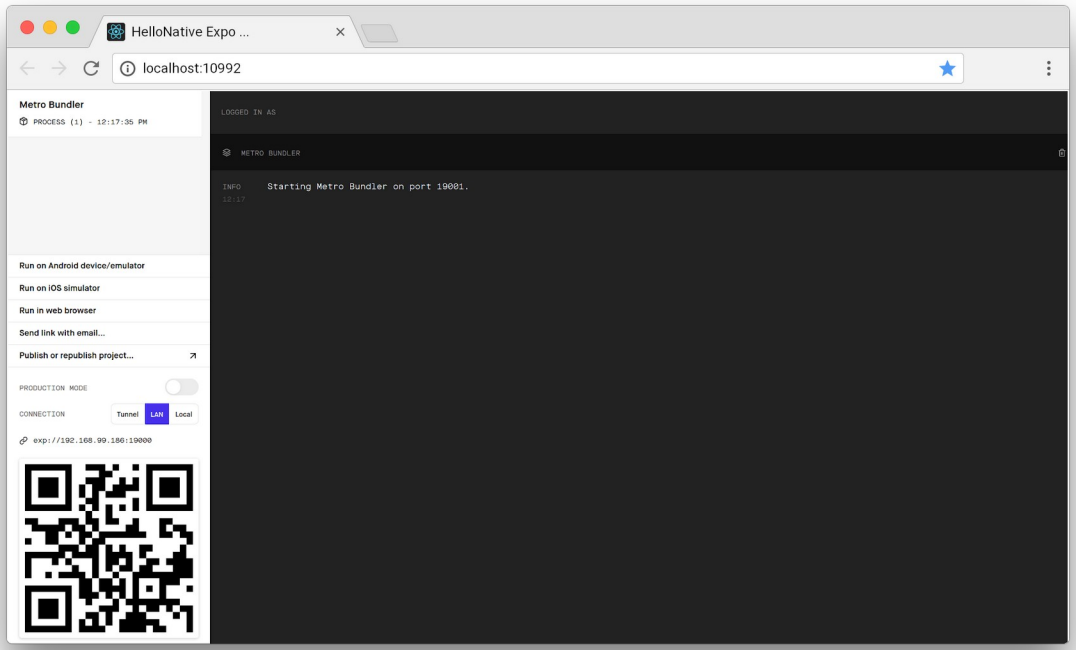
ပြီးတဲ့အခါ React Native ပရောဂျက်တစ်ခုကို အခုလို တည်ဆောက်ပေးရပါတယ်။

```
>> npx expo init HelloNative
```

`HelloNative` ဆိုတာ ပရောဂျက်ဖိုဒါအမည်ဖြစ်ပြီး Capital Case နဲ့ ပေးရတယ်ဆိုတာကို သတိပြုပါ။ ပရောဂျက်တည်ဆောက်စဉ်မှာ blank လား TypeScript လား tabs လားစသဖြင့် Template ရွေးခိုင်းရင် ရွေးမနေဘဲ သူပေးထားတဲ့အတိုင်း blank မှာပဲ Enter နှိပ်လိုက်ပါ။ ပြီးရင် အခုလို Run ကြည့်လို့ရပါပြီ။

```
>> cd HelloNative
>> npm start
```

ပုံမှာပြထားသလို Dashboard တစ်ခုကို ရလဒ်အနေနဲ့ Browser မှာ မြင်ရပါလိမ့်မယ်။



နောက်တစ်ဆင့်အနေနဲ့ ကိုယ့်ဖုံးထဲမှာ **Expo** App ကို Install လုပ်ထားဖို့ လိုပါတယ်။ App Store တို့ Play Store တို့မှာ ရှာပြီးထည့်လိုက်ပါ။ မှားမှာစိုးလို့ တိုက်စစ်ချင်ရင် Link တွေ ထည့်ပေးလိုက်ပါမယ်။

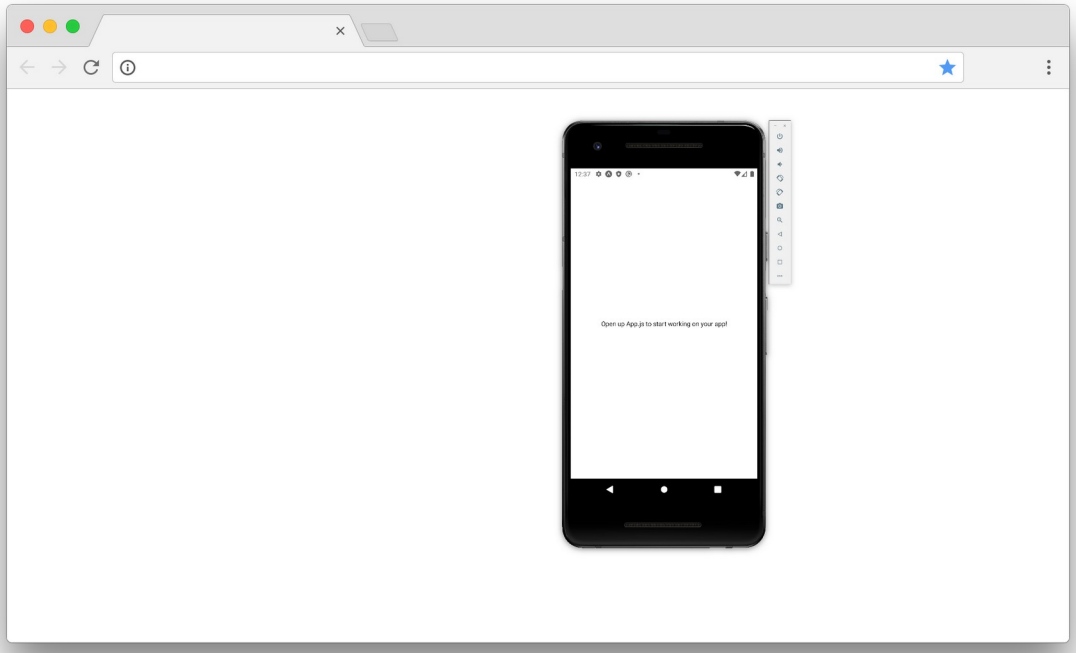
Google Play Store – https://play.google.com/store/apps/details?id=host.exp.exponent&hl=en

Apple App Store – https://apps.apple.com/us/app/expo-client/id982107779

Expo App ကို Install လုပ်ပြီးရင် Android မှာ App ကိုဖွင့်ပါ။ သူ့မှာပါတဲ့ Scan QR Code ခလုပ်ကနေ တစ်ဆင့် စောစောက Dashboard မှာပါတဲ့ QR Code ကို Scan လုပ်ပေးလိုက်ပါ။ Run ထားတဲ့ App ရဲ့ ရလဒ်ကို ကိုယ့်ဖုန်းမှာ တွေ့မြင်ရမှာ ဖြစ်ပါတယ်။ iOS ဆိုရင် Camera App ကို သုံးပြီးတော့ပဲ QR Code ကို Scan လုပ်နိုင်ပါတယ်။ အရေးကြီးတာ တစ်ခုကတော့ စမ်းမယ့်ဖုန်းနဲ့ ရေးနေတဲ့ ကွန်ပျူတာဟာ Wifi Network တစ်ခုတည်းမှာ ရှိရမှာဖြစ်ပါတယ်။ ဒါမှအလုပ်လုပ်မှာပါ။ Wifi မတူရင် အလုပ်မလုပ်ပါဘူး။

နောက်တစ်နည်းအနေနဲ့ ကိုယ့်စက်ထဲမှာ Android Emulator တို့ iOS Simulator တို့ ရှိရင်လည်း အဲ့ဒီ Simulator ထဲမှာ Run ခိုင်းလို့ရပါတယ်။ Android App တွေ iOS App တွေ စမ်းရေးဖူးလို့ စက်ထဲမှာ Simulator တွေရှိတဲ့သူက Simulator ကို ဖွင့်ထားလိုက်ပါ။ ပြီးရင် Dashboard ရဲ့ ဘယ်ဘက်ခြမ်းက Sidebar မှာ Open in Simulator ခလုပ်တွေ ပါပါတယ်။ နှိပ်လိုက်ရင် ရပါပြီ။

အခုလိုရလဒ်မျိုးကို ရမှာပါ။

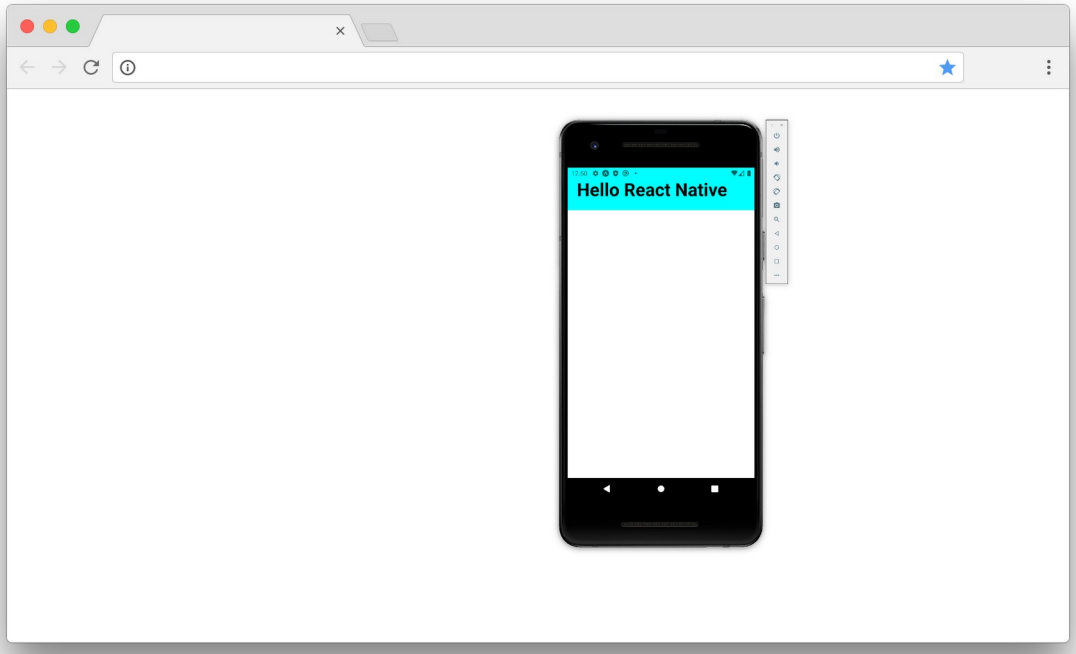


ဒီအဆင့်ထိ စမ်းသပ်အောင်မြင်ပြီဆိုရင် နောက်ပိုင်းက လွယ်သွားပါပြီ။ အများအားဖြင့် ရှေ့ပိုင်းမှာ လေ့လာခဲ့ပြီးဖြစ်တဲ့ React ရေးထုံးတွေအတိုင်းပဲ ဆက်ရေးသွားရမှာပါ။ Component တည်ဆောက်ပုံ၊ state တွေ props တွေ စီမံပုံ၊ အားလုံးအတူတူပါပဲ။ Input စီမံပုံလေး နည်းနည်း ကွဲသွားပြီး `<div>` တို့ `<h1>` တို့ `<ul>` တို့လို HTML Element တွေအစား `<View>` တို့ `<Text>` တို့လို React Native က ဖန်တီးပေးထားတဲ့ Element တွေကို အစားထိုး သုံးပေးရမှာပါ။

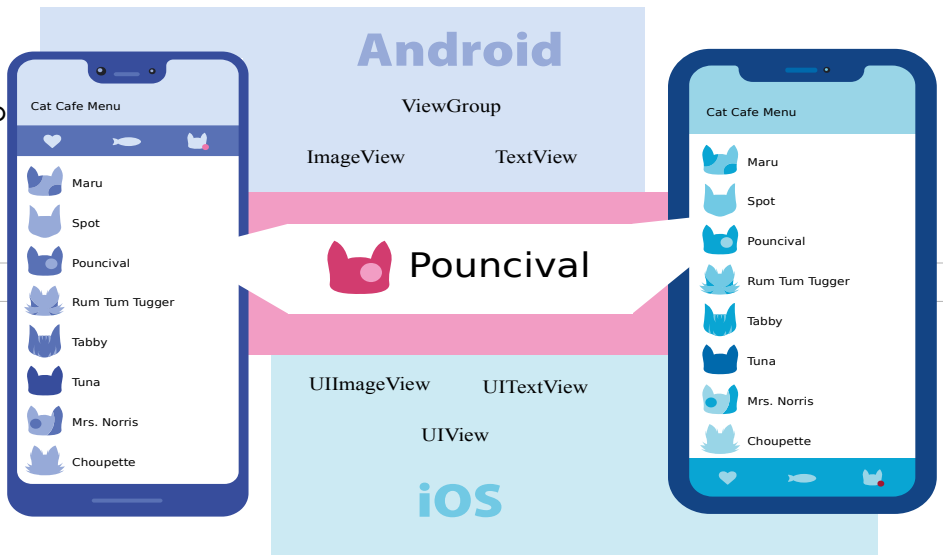
နမူနာအနေနဲ့ ပရောဂျက်ဖိုဒါထဲက `App.js` ကိုဖွင့်ပြီး အခုလိုရေးစမ်းကြည့်ပါ။

```
import React from 'react';
import { Text, View } from 'react-native';

const App = props => {
    return (
        <View style={{ backgroundColor: 'cyan', padding: 20 }}>
            <Text style={{ fontSize: 40, fontWeight: 'bold' }}>
                Hello React Native
            </Text>
        </View>
    )
}

export default App;
```

ရိုးရိုး React ကုဒ်နဲ့ သိပ်မကွာဘဲ `<div>` တွေဘာတွေအစား React Native ကနေ Import လုပ်ယူထားတဲ့ `<View>` နဲ့ `<Text>` တို့ကို သုံးသွားတာကို တွေ့ရမှာပါ။ `<View>` ကို `<div>` နဲ့ တူတယ်လို့ သဘောထားပြီး စာမှန်သမျှ `<Text>` ထဲမှာ ရှိသင့်တယ်လို့ မှတ်ထားပေးပါ။ နောက်ထပ်ထူးခြားချက်ကတော့ `style` အနေနဲ့ သုံးထားတဲ့ CSS ကုဒ်မှာ `background` အစား `backgroundColor` လို့ အပြည့်အစုံ ရေးရတာကိုလည်း သတိပြုပါ။ Style တွေအတွက် ရေးထုံးက CSS ရေးထုံးအတိုင်းပဲ ရေးရပေမယ့် အခုလို ကွဲလွဲမှုလေးတစ်ချို့တော့ ရှိပါတယ်။ ရလဒ်ကို အခုလို ဖြစ်မှာပါ။



တစ်ခြား အသုံးဝင်တဲ့ အခြေခံ Component တွေကတော့ `<ScrollView>` `<Image>` `<TextInput>` `<Button>` `<Picker>` `<Switch>` `<FlatList>` တို့ပဲဖြစ်ပါတယ်။ ဒါတွေက React Native နဲ့ ပါတဲ့ Cross-Platform Component တွေကို ပြောတာပါ။ ဆိုလိုတာက ဒီ Component တွေကို Android App တွေ မှာပဲဖြစ်ဖြစ် iOS App တွေမှာပဲဖြစ်ဖြစ် အသုံးပြုလို့ ရပါတယ်။ ဘယ်လိုနည်းလမ်းမျိုးနဲ့ အသုံးပြုလို့ရသလဲဆိုတာ ဒီပုံလေးကိုကြည့်ပါ။



React Native မှာ –

```
<View>
  <Image /><Text />
</View>
```

လို့ရေးထားပေးမယ်၊ လက်တွေ့အလုပ်လုပ်တဲ့အခါ Android မှာဆိုရင် ViewGroup, ImageView နဲ့ TextView ဆိုတဲ့ Android Native UI တွေကို သုံးပြီး အလုပ်လုပ်ပေးမှာပါ။ iOS မှာဆိုရင်တော့ UIView, UIImageView နဲ့ UITextView ဆိုတဲ့ iOS Native UI တွေကို သုံးသွားမှာပါ။ ဒီနည်းနဲ့ တစ်ကြိမ်ရေးယုံပြီး Platform နှစ်ခုလုံးအတွက် App တွေကို ထုတ်ပေးနိုင်မှာ ဖြစ်ပါတယ်။

Cross-Platform မဟုတ်တဲ့ Android သီးသန့်၊ iOS သီးသန့် Component တွေလည်း ရှိပါသေးတယ်။ ဥပမာ - `<DrawerLayoutAndroid>` `<ToastAndroid>` စတာတွေဟာ Android သီးသန့် Component တွေပါ။

`<View>` `<Text>` `<TextInput>` `<Button>` နဲ့ `<FlatList>` တို့ကို အသုံးပြုပြီး နမူနာတစ်ခု ရေးပြပါမယ်။

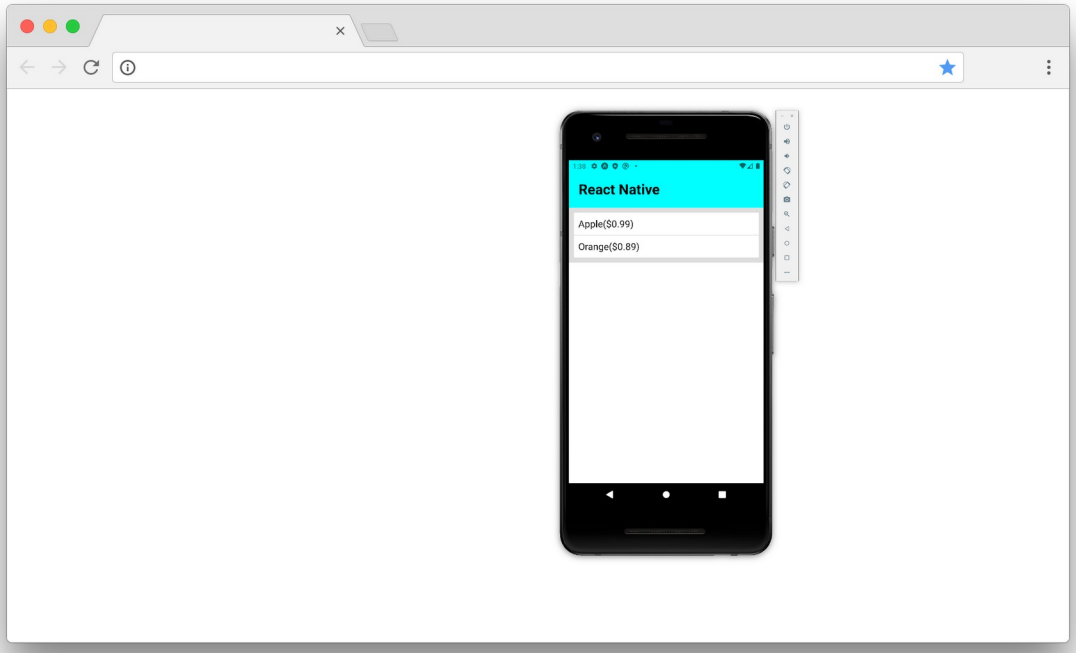
```
import React, { useState } from 'react';
import {
    StyleSheet,
    FlatList,
    Button,
    TextInput,
    Text,
    View
} from 'react-native';

const styles = StyleSheet.create({
    container: {
        backgroundColor: '#ddd',
    },
    appbar: {
        paddingTop: 40,
        paddingBottom: 20,
        paddingLeft: 20,
        paddingRight: 20,
        backgroundColor: 'cyan',
    },

    title: {
        fontSize: 30,
        fontWeight: 'bold'
    },
```

```
    content: {
        margin: 10,
        backgroundColor: 'white',
    },
    item: {
        padding: 10,
        borderBottomWidth: 1,
        borderColor: '#ddd'
    },
    itemText: {
        fontSize: 20
    }
});

const Item = props => {
    return (
        <View style={styles.item}>
            <Text style={styles.itemText}>
                {props.name}
                (${props.price})
            </Text>
        </View>
    )
}

const App = props => {
    let [ items, setItem ] = useState([
        { id: '1', name: 'Apple', price: 0.99 },
        { id: '2', name: 'Orange', price: 0.89 },
    ]);

    return (
        <View style={styles.container}>
            <View style={styles.appbar}>
                <Text style={styles.title}>React Native</Text>
            </View>
            <View style={styles.content}>
                <FlatList
                    data={items}
                    renderItem={({ item }) => (
                        <Item
                            name={item.name}
                            price={item.price}
                        />
                    )}
                    keyExtractor={i => i.id}
                />
            </View>
        </View>
    )
}

export default App;
```

ကုဒ်တွေများလို့ လန့်မသွားပါနဲ့။ Style တွေ များနေတာပါ။ Style တွေကို ဒီတိုင်းရေးရင်လည်း ရပေမယ့် React Native ကပေးတဲ့ Stylesheet လုပ်ဆောင်ချက်ကို သုံးထားပါတယ်။ `create()`,

`compose()`, `flatten()` စသဖြင့် ရေးထားတဲ့ Style တွေကို စီမံတဲ့အခါ အထောက်အကူပြုတဲ့ လုပ်ဆောင်ချက်တွေ ရနိုင်ပါတယ်။ ပြီးတော့ အဲ့ဒီ Style တွေက လက်တွေ့မှာ အခုလို ကိုယ်တိုင် အကုန် ရေးစရာ မလိုပါဘူး။ အသင့်သုံး UI Framework တွေ အများကြီးရှိပါတယ်။ လိုချင်တဲ့ UI ကို ယူသုံးယုံပါပဲ။ ဒီနေရာမှာသာ ဒီလို Framework တွေကို သုံးပြီး နမူနာမပြချင်လို့ ကိုယ့်ဘာသာ ရေးထားရတာပါ။

ကျန်တဲ့ JavaScript ကုဒ်က သိပ်တောင် ရှင်းပြစရာ မရှိပါဘူး။ ဖတ်ကြည့်ပါ။ ရိုးရိုး React ကုဒ်အတိုင်းပဲ ရေးထားတာပါ။ ထူးခြားချက်အနေနဲ့ `<FlatList>` အကြောင်းလောက်ပဲ ပြောစရာရှိပါတယ်။ FlatList အတွက် props (၃) ခု လိုပါတယ်။ `data` က List အနေနဲ့ဖော်ပြစေလိုတဲ့ စာရင်းပါ။ `renderItem` ကတော့ List Item တစ်ခုချင်းစီကို ဘယ်လိုဖော်ပြရမလဲ ကြိုတင်သတ်မှတ်ထားတဲ့ Component ပါ။ `keyExtractor` ကတော့ React မှာ ထည့်ပြောခဲ့တဲ့ `key` props နဲ့ သဘောသဘာဝ တူပါတယ်။ ဒီ (၃) ခုစုံရင် သူ့ဘာသာ ဖော်ပြသွားလို့ `map()` တွေဘာတွေနဲ့ Loop လုပ်နေစရာ မလိုတော့ပါဘူး။ ရလဒ်က အခုလိုပုံစံဖြစ်မှာပါ။



နောက်တစ်ဆင့်အနေနဲ့ App Component ရဲ့ ဖွဲ့စည်းပုံက ဒီပုံစံဖြစ်သွားပါလိမ့်မယ်။

```
const App = props => {
    const [ items, setItem ] = useState([
        { id: '1', name: 'Apple', price: 0.99 },
        { id: '2', name: 'Orange', price: 0.89 },
    ]);

    const [name, setName] = useState('Name');
    const [price, setPrice] = useState('Price');

    const add = () => {
        setItem([
            ...items,
            { id: items.length + 1, name, price }
        ])
    }

    return (
        <View style={styles.container}>
            <View style={styles.appbar}>
                <Text style={styles.title}>React Native</Text>
            </View>
            <View style={styles.content}>
                <FlatList
                    data={items}
                    renderItem={({ item }) => (
                        <Item name={item.name} price={item.price} />
                    )}
                    keyExtractor={i => i.id}
                />
            </View>
            <View style={styles.content}>
                <TextInput
                    style={styles.input}
                    onChangeText={text => setName(text)}
                    value={name}
                    />
                <TextInput
                    keyboardType="numeric"
                    style={styles.input}
                    onChangeText={text => setPrice(text)}
                    value={price}
                    />
                <Button title="ADD" onPress={add} />
            </View>
        </View>
    )
}
```

ဒီနေရာမှာ Input တွေစီမံပုံအကြောင်း ပြောရပါမယ်။ ရိုးရိုး React ပရောဂျက်တွေမှာ Input တွေစီမံဖို့

ref ကို သုံးပါတယ်။ React Native မှာ ref လုပ်ဆောင်ချက် မရှိပါဘူး။ ဒါကြောင့် Input တွေကို state နဲ့ စီမံပါတယ်။ Input မှာ ရိုက်လိုက်သမျှ တန်ဖိုးအားလုံးကို state မှာ သွားသိမ်းလိုက်တာပါ။ ဒါကြောင့် Input ရဲ့ တန်ဖိုးကို လိုချင်ရင် state ကနေ ပြန်ယူရပါတယ်။ ref မပေါ်ခင်က ရိုးရိုး React ပရောဂျက်တွေမှာလည်း ဒီလိုပဲ သွားရပါတယ်။

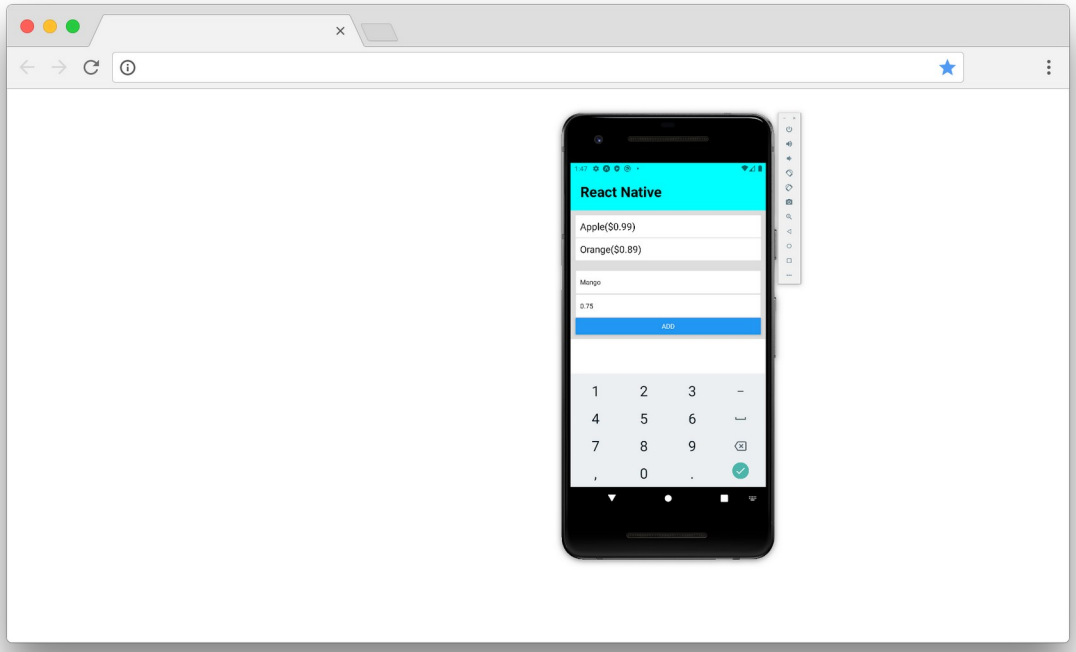
ရေးထားတဲ့ ကုဒ်ကို ကြည့်လိုက်ရင် useState() Hook ကို အသုံးပြုပြီး name, setName, price, setPrice စသဖြင့် state နဲ့ state ကို စီမံနိုင်တဲ့ Function တွေ တည်ဆောက်ပါတယ်။ ပြီးတော့မှ <TextInput> Component တွေရဲ့ onChangeText မှာ တန်ဖိုးပြောင်းတိုင်း name, price စတဲ့ state တွေကို ပြောင်းဖို့ ရေးထားပါတယ်။ value မှာတော့ လက်ရှိ state Data ကို ပြန်သတ်မှတ်ပေးထားပါတယ်။ ဒီနည်းနဲ့ ရိုက်လိုက်သမျှ သွားသိမ်းပြီး သိမ်းထားသမျှ ပြန်ပြတဲ့ သဘောသဘာဝကို ရရှိသွားပါတယ်။

input ဆိုတဲ့ Style ကိုလည်း သုံးထားလို့ ဒီ Style ကုဒ်လေးကို Style စာရင်းထဲမှာ ဖြည့်ပေးဖို့ လိုပါလိမ့်မယ်။

```
input: {
    borderWidth: 1,
    borderColor: '#ddd',
    padding: 10,
}
```

<Button> Component ကိုလည်းသုံးထားပြီး နှိပ်လိုက်ရင် add() ကို အလုပ်လုပ်ပေးပါတယ်။ add() မှာရေးထားတဲ့ ကုဒ်ကတော့ ထူးခြားချက်အသစ် မရှိပါဘူး။ name နဲ့ price ကို state ကနေ တိုက်ရိုက် ယူသုံးထားတာပဲ ရှိပါတယ်။ Button ရဲ့ onPress ကိုတော့ သတိပြုပါ။ ရိုးရိုး React မှာ onClick လို့ သုံးပေမယ့် React Native မှာ onPress လို့ သုံးပါတယ်။

ရလဒ်ကတော့ အခုလိုဖြစ်မှာပါ။



ဒီလောက်သိသွားပြီဆိုရင် React Native ကို စအသုံးပြုလို့ရနေပါပြီ။ Component တွေကို CSS Flexbox သုံးပြီး Layout ဖန်တီးပုံတွေ၊ Device ရဲ့ Status Bar တို့ Back Button တို့ Storage တို့ကို စီမံပုံတွေ၊ Page တွေခွဲပြီး Navigation တွေ ဘာတွေနဲ့ ရေးပုံရေးနည်းတွေ၊ Native API တွေ ခေါ်သုံးပုံသုံးနည်းတွေ၊ ပြောမယ်ဆိုရင်တော့ အများကြီး ကျန်ပါသေးတယ်။

ဒါပေမယ့် ရှေ့အခန်းတွေမှာ ပြောခဲ့သလိုပါပဲ။ React အခြေခံတွေ ကိုကောင်းကောင်း ကြေညက်ပိုင်နိုင်ဖို့ သာအဓိကပါ။ React ကို ပိုင်နိုင်ရင် React Native ကို ဆက်လေ့လာရတာ မခက်တော့ပါဘူး။

အခန်း (၁၁) – Promises

စစချင်းမှာ ES6 အကြောင်းပြောခဲ့ပေမယ့် Promise အကြောင်း ထည့်မပြောခဲ့ပါဘူး။ Promise ဟာ လည်း ES6 ရဲ့ လုပ်ဆောင်ချက်တွေထဲက တစ်ခုပါပဲ။ ပုံမှန်အားဖြင့် JavaScript မှာ Asynchronous ကုဒ်တွေရေးဖို့ Callback တွေကို အသုံးပြုကြပါတယ်။ Asynchronous ကုဒ်မဟုတ်ပေမယ့် Callback ရဲ့ သဘောသဘာဝကို မြင်သာစေမယ့် ဥပမာလေးတစ်ခု ပေးပါမယ်။ ဒီလိုပါ –

```
function add(nums, resolve, reject) {
    if(Array.isArray(nums)) {
        let result = nums.reduce((a, b) => a+ b);
        resolve(result);
    } else {
        reject();
    }
}
```

နမူနာအရ `add()` Function ဟာ Argument (၃) ခု လက်ခံပါတယ်။ `nums` အတွက် Array ရယ် `resolve` အတွက် Function ရယ် `reject` အတွက် Function ရယ် ဖြစ်ပါတယ်။ သူ့ရဲ့အလုပ်လုပ်ပုံကတော့ ပေးလာတဲ့ `nums` က Array ဟုတ်မဟုတ် စစ်ပြီးတော့ မှန်ရင် `resolve()` ကို Run ပေးတယ်၊ မမှန်ရင်တော့ `reject()` ကို Run ပေးပါတယ်။ ခေါ်ယူ အသုံးပြုလိုတဲ့အခါ ဒီလို ခေါ်ရမှာပါ။

```
add(1, result => {
    console.log(`Result: ${result}`);
}, () => {
    console.log('Something wrong!');
});

// => Something wrong
```

```
add([1, 2], result => {
    console.log(`Result: ${result}`);
}, () => {
    console.log('Something wrong!');
});

// => Result: 3
```

တွေ့ရတဲ့အတိုင်း `nums` တန်ဖိုးမမှန်တဲ့အခါ `reject()` Function အလုပ်လုပ်သွားပြီး `nums` တန်ဖိုးမှန်တဲ့အခါ `resolve()` Function အလုပ်လုပ် သွားပါတယ်။ ဒီကုဒ်ကိုပဲ ES6 ရဲ့ Promise ကို အသုံးပြုပြီး ပြန်ရေးပါမယ်။

```
function add(nums) {
    return new Promise((resolve, reject) => {
        if(Array.isArray(nums)) {
            let result = nums.reduce((a, b) => a + b);
            resolve(result);
        } else {
            reject();
        }
    });
}
```

`add()` Function ဟာ `nums` Array တစ်ခုတည်းကိုပဲ လက်ခံပါတော့တယ်။ ဒါပေမယ့် Promise တစ်ခုကို Return ပြန်ပေးထားပြီး မှန်ရင် `resolve()` ကို Run ပါတယ်။ မမှန်ရင်တော့ `reject()` ကို Run ပါတယ်။ စောစောက Callback ကုဒ်နဲ့ သဘောသဘာဝတူပြီး ရေးပုံကွာသွားတာပါ။ ဒီ Promise ပြန်ပေးတဲ့ Function ကို ခေါ်သုံးပုံက ဒီလိုပါ –

```
const result = add([1, 2]);
```

`add()` Function ခေါ်လိုက်တဲ့အခါ Promise တစ်ခုကို ပြန်ရမှာပါ။ ပြန်ရတဲ့ Promise ပေါ်မှာ `then()` Function ရဲ့ အကူအညီနဲ့ `resolve` ကို ပေးရပါတယ်။

```
result.then( result => {
    console.log(`Result: ${result}`);
});

// => 3
```

ဒီလိုနှစ်ဆင့် ခွဲရေးမနေဘဲ တစ်ဆက်တည်း ရေးလိုက်မယ်ဆိုရင်လည်း ရပါတယ်။ ဒီလိုပါ –

```
add([1, 2]).then( r => {
    console.log(`Result: ${r}`);
}).catch(() => {
    console.log('Something wrong!');
});

// => Result: 3
```

`resolve` ကို `then()` နဲ့ပေးရပြီး `reject` ကိုတော့ `catch()` နဲ့ ပေးရတာပါ။ Promise ရဲ့ ထူးခြားချက်က `then()` Function တစ်ခါ Run တိုင်း Promise ကို Return ပြန်ရလို့ `then()` Function တွေကို အခုလို အခါခါ Chain လုပ်ပြီး ရေးလို့ရပါတယ်။

```
add([1, 2]).then( a => {
    return a + 1;
}).then(b => {
    return b + 1;
}).then(c => {
    console.log(`Result: ${c}`);
}).catch(() => {
    console.log('Something wrong!');
});

// => Result: 5
```

ပထမ `then()` Function က ရလဒ်ကို တစ်တိုးပြီး Return ပြန်ပေးပါတယ်။ ဒုတိယ `then()` Function က ပထမ `then()` Function ပြန်ပေးတဲ့ တန်ဖိုးကို လက်ခံယူပြီး ဆက်အလုပ်လုပ်ပါတယ်။ တတိယ `then()` Function က ဒုတိယ `then()` Function ပြန်ပေးတဲ့ တန်ဖိုးကို လက်ခံယူပြီး ဆက်အလုပ်လုပ်ပေးပါတယ်။ ဒီနည်းနဲ့ ရလာတဲ့ တန်ဖိုးပေါ်မှာ လုပ်စရာရှိတဲ့ အလုပ်တွေကို ဆက်တိုက် အဆင့်ဆင့် လုပ်သွားလို့ ရတဲ့သဘော ဖြစ်ပါတယ်။

နောက်ထပ်ထူးခြားချက်ကတော့ ဒီကုဒ်ကို တမင်ရည်ရွယ်ပြီး Asynchronous ကုဒ်အနေနဲ့ ရေးထားတာမဟုတ်ပေမယ့် Promise ကြောင့် Asynchronous အလိုလို ဖြစ်နေပါတယ်။ ဒီလိုစမ်းကြည့်လို့ ရပါတယ်။

```
add([1, 2]).then(result => {
    console.log(`Result: ${result}`);
}).catch(() => {
    console.log('Something wrong!');
});

console.log('This works ahead');

// => This works ahead
// => Result: 3
```

`add()` Function ကို အရင် Run ပေမယ့် ပြီးအောင် မစောင့်ဘဲ လုပ်စရာရှိတာကို ဆက်လုပ်သွားလို့ သူ့အောက်မှာ ဆက်ရေးထားတဲ့ကုဒ်ရဲ့ ရလဒ်ကို အရင်မြင်ရတာကို တွေ့ရမှာ ဖြစ်ပါတယ်။

async, await

Promise တွေနဲ့အတူ async, await လို့ခေါ်တဲ့ ရေးထုံးတစ်မျိုးကိုလည်း ပူးတွဲ အသုံးပြုနိုင် ပါသေးတယ်။ ဒီလိုပါ –

```
async function sum(nums) {
    let result = await add(nums);
    console.log(`Result: ${result}`);
}

sum([1, 2]);
console.log('This works ahead');

// => This works ahead
// => Result: 3
```

`sum()` Function ဟာ သူအလုပ်လုပ်တာကို စောင့်စရာမလိုတဲ့ `async` Function တစ်ခုပါ။ ဒါကြောင့် အောက်မှာ `sum()` Function ကို ခေါ်ထားပေမယ့် သူ့ကိုမစောင့်ဘဲ သူ့နောက်မှရေးထားတဲ့ ကုဒ်က အရင် အလုပ်လုပ်နေတာကို တွေ့ရနိုင်ပါတယ်။

`sum()` Function ထဲမှာ `await add()` လို့ရေးထားတဲ့အတွက် `add()` Function အလုပ်လုပ်လို့ မပြီး မချင်း သူကစောင့်နေမှာပါ။ `add()` Function ကတော့ မူလက Promise ကို Return ပြန်ပေးတဲ့ Function ပါပဲ။ `then()` တွေ `catch()` တွေမပါတော့ဘဲ `resolve` ဖြစ်ပြီး နောက်ဆုံးရလဒ်ကိုပဲ တစ်ခါတည်း တန်းယူပေးသွားမှာ ဖြစ်ပါတယ်။

Promise ရဲ့ `reject` ကို ဖမ်းချင်ရင်တော့ ဒီလို ရေးနိုင်ပါတယ်။

```
async function sum(nums) {
    try {
        let result = await add(nums);
        console.log(`Result: ${result}`);
    } catch {
        console.log('Something wrong');
    }
}

sum(1);
console.log('This works ahead');

// => This works ahead
// => Something wrong
```

နောက်တစ်ခန်းမှာ React App တွေမှာ API နဲ့ ဘယ်လိုဆက်သွယ် အလုပ်လုပ်သလဲဆိုတာကို လေ့လာကြ မှာပါ။ `fetch()` လို့ခေါ်တဲ့ နည်းပညာကို အသုံးပြုမှာဖြစ်ပြီး Promise အကြောင်း သိထားဖို့ လိုအပ်လို့ အခုလို ကြိုတင်ပြီး ကြားဖြတ် ရှင်းပြထားတာပါ။

အခန်း (၁၂) – Working with API

React ဟာ Client-side နည်းပညာတစ်ခုဖြစ်သလို ရှေ့ပိုင်းမှာ ပြောခဲ့သမျှတောက်လျှောက်ကလည်း Client App တစ်ခုအနေနဲ့သာ ပြောခဲ့တာပါ။ Server-side API တွေနဲ့ ချိတ်ဆက် အလုပ်လုပ်ပုံတွေ မပါသေးပါဘူး။ ဒီစာအုပ်မှာ Server-side API ဖန်တီးမှုအကြောင်းတော့ မထည့်နိုင်ပါဘူး။ အသင့်ရှိနေတဲ့ Test API တစ်ခုကို အသုံးပြုပြီး React ပရောဂျက်ကနေ ဘယ်လိုဆက်သွယ် အသုံးပြုရသလဲ ဆိုတာကို ဖော်ပြသွားပါမယ်။

ပထမဆုံးအနေနဲ့ ဒီ Class Component ကို ပြန်လေ့လာကြည့်ပါ။

```
class App extends React.Component {
    state = {
        users: [
            { id: 1, first_name: 'Alice' },
            { id: 2, first_name: 'Bob' },
        ]
    }

    render() {
        return (
            <ul>
                {this.state.users.map(u =>
            <li key={u.id}>{u.first_name}</li>)}
            </ul>
        )
    }
}
```

`users` ကို `map()` လုပ် ဖော်ပြထားတဲ့ Component လေးတစ်ခုပါပဲ။

Class Component တွေမှာ Life-Cycle Methods ဆိုတာ ရှိပါတယ်။ Component ကိုမဖော်ပြခင်မှာ ဘာလုပ်ရမယ်၊ ဖော်ပြပြီးရင် ဘာလုပ်ရမယ်၊ Component ကို ဖျက်လိုက်ရင် ဘာလုပ်ရမယ် စသဖြင့် သတ်မှတ်ထားနိုင်ပါတယ်။ အဲ့ဒီထဲ `componentDidMount()` ဆိုတဲ့ Life-cycle Method ကို အသုံးပြုပါမယ်။ Component ကို ဖော်ပြပြီးတာနဲ့ ဒီ Method ကို React က အလုပ်လုပ်ပေးသွားမှာပါ။ App Class ထဲမှာ ဒီ Method ကို ရေးပေးပါ။

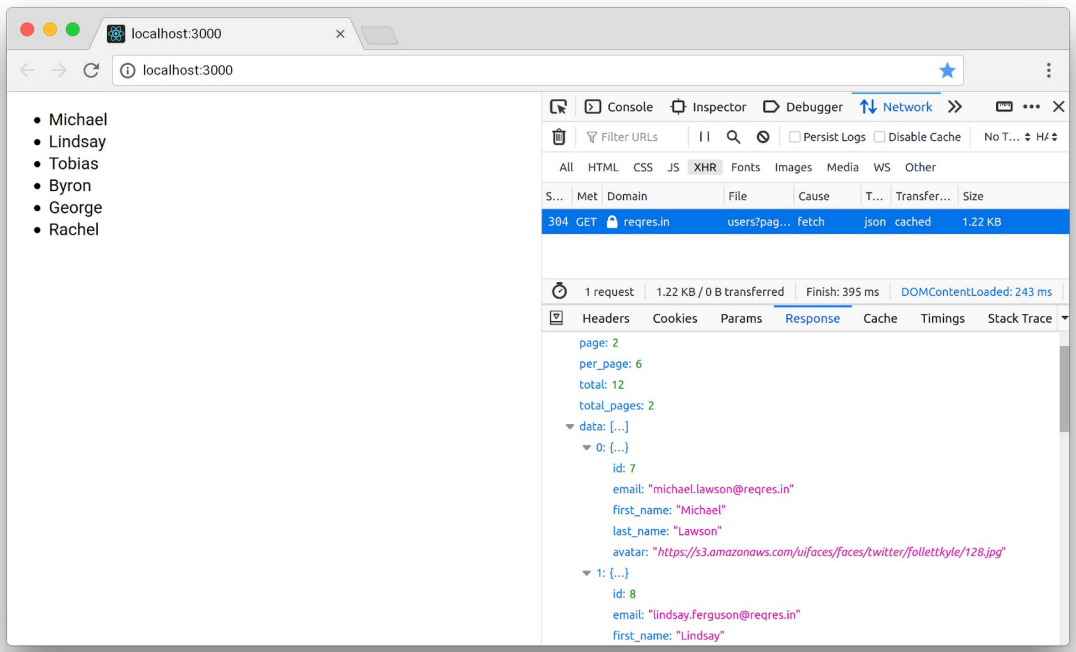
```
componentDidMount() {
    fetch('https://reqres.in/api/users')
      .then(res => res.json())
      .then(json => {
            this.setState({ users: json.data });
        });
}
```

Test API URL အနေနဲ့ `reqres.in/api/users` ကိုအသုံးပြုထားပါတယ်။ ပြန်ရမယ့် Response Body ရဲ့ ဖွဲ့စည်းပုံက ဒီလိုဖြစ်မှာပါ။

```
{
    "page": 2,
    "per_page": 6,
    "total": 12,
    "total_pages": 2,
    "data": [{
        "id": 7,
        "email": "michael.lawson@reqres.in",
        "first_name": "Michael",
        "last_name": "Lawson"
    }, {
        "id": 8,
        "email": "lindsay.ferguson@reqres.in",
        "first_name": "Lindsay",
        "last_name": "Ferguson"
    }
}
```

`page`, `per_page`, `total` စသဖြင့် Meta Information လဲပါသလို နမူနာ User စာရင်းကိုလည်း `data` အနေနဲ့ ထည့်ပေးထားပါတယ်။ ဒီ API ကို Request ပြုလုပ်ဖို့အတွက် `fetch()` JavaScript Function ကို အသုံးပြု ထားပါတယ်။ အရင်ကတော့ အခုလို API Request တွေအတွက် Ajax တို့ jQuery တို့ကို အသုံးပြုရပါတယ်။ အခုတော့ မလိုတော့ပါဘူး။ JavaScript မှာ ပါဝင်လာတဲ့ `fetch()` Function နဲ့တင် အဆင်ပြေသွားပါပြီ။

`fetch()` Function က API URL ကို Argument အနေနဲ့ ပေးရပြီး Promise တစ်ခုကို Response ပြန်ပေးပါတယ်။ Request/Response Error တွေရှိရင် `reject` လုပ်မှာဖြစ်ပြီး၊ Error မရှိရင်တော့ resolve လုပ်ပေးမှာပါ။ နမူနာမှာ ပြန်ရလာမယ့် Response ကို လက်ခံပြီး JSON ပြောင်းပါတယ်။ ပြောင်းထားတဲ့ JSON ရဲ့ `data` ကို state ရဲ့ `users` အဖြစ် သတ်မှတ်လိုက်တဲ့အတွက် API က ပြန်ပေးတဲ့ User စာရင်းကို Component က ဖော်ပြပေးသွားမှာ ဖြစ်ပါတယ်။



Function Component တွေမှာ Life-Cycle Methods တွေမရှိလို့ အရင်က အခုလို API တွေဘာတွေနဲ့ ချိတ်သုံးလို့ အဆင်မပြေပါဘူး။ ဒါပေမယ့် အခုတော့ Hook တွေပါဝင်လာတဲ့အတွက် အဆင်ပြေသွားပါပြီ။ စောစောက Component ကို Function နဲ့ ရေးမယ်ဆိုရင် ဒီလိုရေးရမှာပါ။

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

const App = props => {
    const [ users, setUsers ] = useState([]);

    useEffect(() => {
        fetch('https://reqres.in/api/users?page=2')
     .then(res => res.json())
     .then(json => {
            setUsers(json.data);
        });
    }, []);

    return (
        <ul>
            {users.map(u => <li key={u.id}>{u.first_name}</li>)}
        </ul>
    );
}

export default App;
```

Import ကနေ Export ထိ ကုဒ်အပြည့်စုံပေးထားပါတယ်။ `useEffect` လို့ခေါ်တဲ့ Hook ကို Import လုပ်ထားတာကို မသိလိုက်မှာ စိုးလို့ပါ။ Component ကုဒ်မှာ `useEffect()` Hook ကို အသုံးပြုပြီး API ကို လှမ်းခေါ်ထားပါတယ်။ `useEffect()` အတွက် Argument နှစ်ခု ပေးထားပါတယ်။ ပထမ Argument က Function တစ်ခုဖြစ်ပြီး Component ကိုဖော်ပြပြီးတဲ့အခါ ဒီ Function ကို `useEffect()` က Run ပေးမှာပါ။ ဒုတိယ Argument ကတော့ ဘယ်အချိန်မှာ `useEffect()` ကို ထပ် Run ပေးရမှာလဲဆိုတာကို သတ်မှတ်ပေးတာပါ။ နမူနာမှာ Array အလွတ်ကို ပေးထားလို့ ထပ် Run စရာမလိုဘူးဆိုတဲ့ သဘောပါ။ ဒါကြောင့် Component ကို ဖော်ပြပြီး တစ်ကြိမ်တည်းပဲ `useEffect()` အလုပ်လုပ်မှာ ဖြစ်ပါတယ်။

နောက်ထပ်နမူနာအနေနဲ့ `add()` လုပ်ဆောင်ချက်ကို ထပ်ဖြည့်ကြည့်ပါမယ်။ ဒီလိုရေးရပါတယ်။

```
const add = () => {
    fetch('https://reqres.in/api/users', {
        method: 'POST',
        headers: {
            'content-type': 'application/json'
        },
        body: JSON.stringify({ first_name: 'Tom' })
    }).then(res => res.json()).then(tom => {
        setUsers([ ...users, tom ]);
    });
}
```

ဒီတစ်ခါတော့ `fetch()` အတွက် Argument နှစ်ခု ဖြစ်သွားပါပြီ။ ပထမ Argument က API URL ပါ။ ဒုတိယ Argument ကတော့ Request Options တွေပါ။ Request Method ကို POST လို့ သတ်မှတ်ထားပါတယ်။ Header Content Type ကို `application/json` လို့သတ်မှတ်ထားပါတယ်။ ကျွန်တော်တို့ လက်ရှိသုံးနေတဲ့ Test API အပါအဝင် API အများစုက ဒီလို Content Type ပါလာမှပဲ လက်ခံကြပါတယ်။ Request Body က String ဖြစ်ရမှာပါ။ ဒါကြောင့် `JSON.stringify()` နဲ့ ပေးပို့ချင်တဲ့ Data ကို String ပြောင်းပြီး သတ်မှတ်ပေးထားပါတယ်။

Request အောင်မြင်ရင်တော့ ပြန်ရလာတဲ့ Response ကို `state` ထဲမှာ ထပ်တိုးပေးလိုက်မှာပဲ ဖြစ်ပါတယ်။ ပြန်လေ့လာချင်ရင် အစအဆုံး ပြန်လေ့လာနိုင်ဖို့အတွက် ရေးရမယ့် ကုဒ်အပြည့်အစုံကို ထပ်ပြီးတော့ ဖော်ပြပေးလိုက်ပါတယ်။

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


const App = props => {
    const [ users, setUsers ] = useState([]);

    useEffect(() => {
        fetch('https://reqres.in/api/users')
     .then(res => res.json())
     .then(json => {
        setUsers(json.data);
        });
    }, []);
```

```
    const add = () => {
        fetch('https://reqres.in/api/users', {
            method: 'POST',
            headers: {
                'content-type': 'application/json'
            },
            body: JSON.stringify({ first_name: 'Tom' })
        }).then(res => res.json()).then(tom => {
            setUsers([ ...users, tom ]);
        });
    }

    return (
        <div>
            <ul>
                {users.map(u =>
               <li key={u.id}>{u.first_name}</li>)}
            </ul>
            <button onClick={add}>New User</button>
        </div>
    );
}

export default App;
```

React Native ပရောဂျက်တွေမှာလည်း ဒီနည်းနဲ့ပုံ API တွေနဲ့ ချိတ်ဆက် အသုံးပြုရပါတယ်။ ဒီလောက် လေ့လာထားလိုက်ရင် API အကြောင်း တီးမီးခေါက်မိရှိသူ တစ်ယောက်အနေနဲ့ Update Request, Delete Request စသဖြင့် ကျန်နေတဲ့ Request အမျိုးအစားတွေကို ဆက်လက်ပြုလုပ်သွားနိုင်မှာပါ။

လိုအပ်ရင်ကုဒ်အပြည့်အစုံကို ဒီကနေယူလိုက်ပါ – https://github.com/eimg/react-book

အခန်း (၁၃) – Next.js

Next.js ဆိုတာ React အတွက် Server-side rendering နည်းပညာ တစ်ခုပါ။ ရိုးရိုး React မှာ Component တည်ဆောက်ပုံ၊ ဖော်ပြပုံ၊ ပြောင်းလဲပုံတွေ အကုန်လုံးက Browser ထဲမှာပဲ လုပ်သွားတာပါ။ Next.js မှာတော့ Component တည်ဆောက်တဲ့ ကိစ္စကို Server-side မှာ လုပ်ပြီး နောက်ဆုံးရလဒ်ကိုသာ Browser ကို ပေးပို့ဖော်ပြစေမှာပါ။

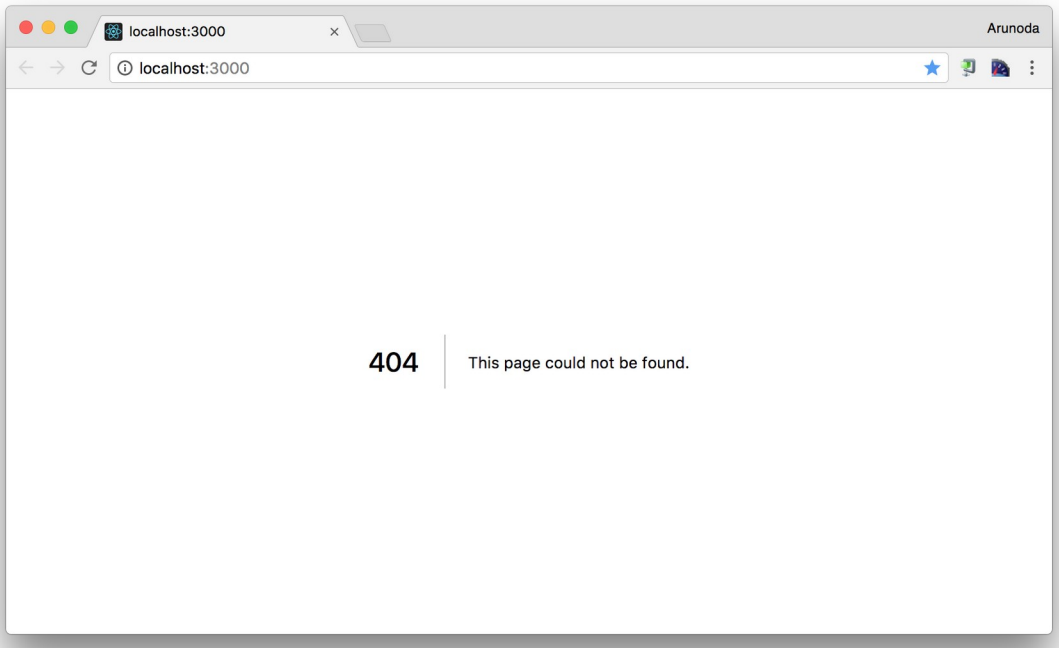
စတင်စမ်းသပ် အသုံးပြုနိုင်ဖို့အတွက် ပရောဂျက်ဖိုဒါတစ်ခု မိမိဘာသာ တည်ဆောက်ပါ။ အခဲ့ဒီ ပရောဂျက်ဖိုဒါထဲမှာ React နဲ့ Next.js တို့ကို အခုလို Install လုပ်ပေးရပါမယ်။

```
>> npm init -y
>> npm i react react-dom next
```

`npm init -y` ရဲ့ အဓိပ္ပါယ်က ဒီဖိုဒါကို NPM Package တစ်ခုအဖြစ် ကြေညာလိုက်တာပါ။ ပရောဂျက်ဖိုဒါထဲမှာ `package.json` ဆိုတဲ့ ဖိုင်တစ်ခု ဝင်သွားပြီး `name`, `version`, `license` စတဲ့ အချက်အလက်တွေ ပါဝင်ပါလိမ့်မယ်။ ဆန္ဒရှိရင် ကိုယ့်ဘာသာ ဖွင့်ပြင်လို့ ရပါတယ်။ အရင်က `create-react-app` ကိုသုံးနေလို့ `react` တို့ `react-dom` တို့ကို ကိုယ်တိုင် Install လုပ်စရာမလိုတာပါ။ အခုတော့ ထည့်ပြီး Install လုပ်ထားပါတယ်။ `next` ကိုလည်း Install လုပ်ထားပါတယ်။ အခုလို Run ကြည့်လို့ ရပါတယ်။

```
>> npx next
```

ရလဒ်က အခုလိုပုံစံဖြစ်မှာပါ။



ဘာကုဒ်မှ မရေးရသေးတဲ့အတွက် 404 ပြနေတာပါ။ ကုဒ်တွေစရေးနိုင်ဖို့အတွက် ပရောဂျက်ဖိုဒါထဲမှာပဲ pages အမည်နဲ့ ဖိုဒါတစ်ခု ထပ်ဆောက်ပေးပါ။ ပြီးရင် Home Page တစ်ခု သတ်မှတ်တဲ့အနေနဲ့ index.js အမည်နဲ့ ကုဒ်ဖိုင်တစ်ခုကို pages ဖိုဒါထဲမှာ အခုလို ရေးပြီးစမ်းကြည့်နိုင်ပါတယ်။

```
const Home = props => {
    return (
        <div>
            <h1>Welcome to Next.js</h1>
            <ul>
                <li>Alice</li>
                <li>Bob</li>
            </ul>
        </div>
    )
}

export default Home;
```

ရိုးရိုး React Component တစ်ခုပါပဲ။ React ကို Import လုပ်စရာ မလိုတာကိုတော့ သတိပြုပါ။ ရလဒ်က

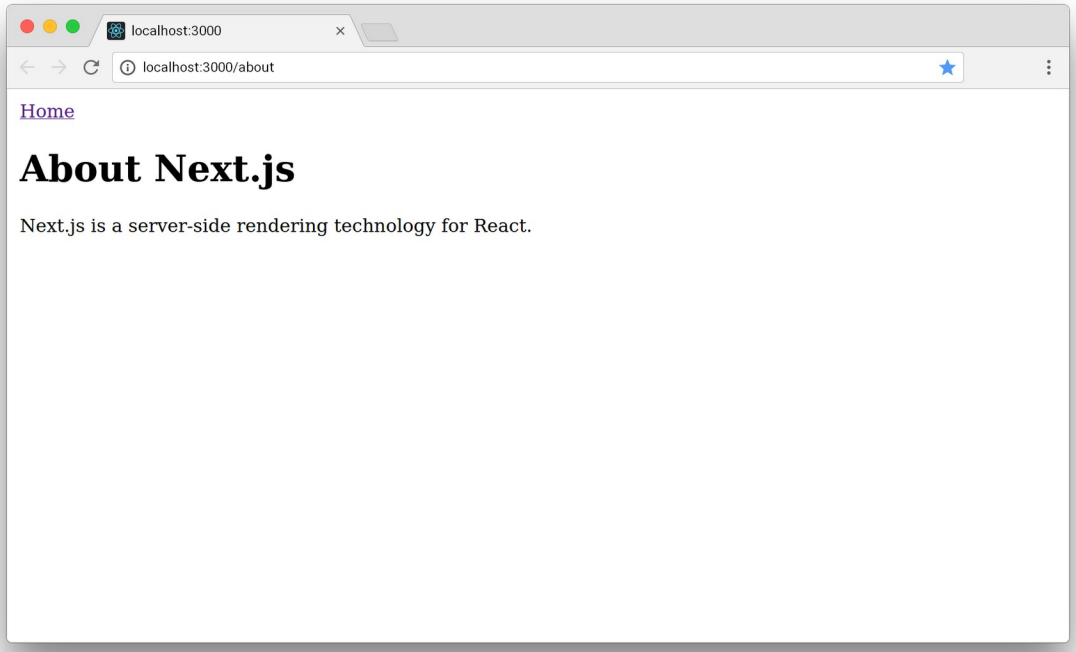
ရိုးရိုး React နဲ့ အတူတူပါပဲ။ ရလဒ်ခြင်းအတူတူ အရင်က Browser ထဲမှာပဲ အလုပ်လုပ်ပြီး ပြတာနဲ့ အခု က Server Render လုပ်ပြီး နောက်ဆုံးရလဒ် သက်သက်ကိုသာ ဖော်ပြသွားတာပါ။ နောက်ထပ် `about.js` အမည်နဲ့ ကုဒ်ဖိုင်တစ်ခုလောက် pages ဖိုဒါထဲမှာ ပဲ ထပ်ရေးပေးပါ။

```
import Link from 'next/link';

const About = props => {
    return (
        <div>
            <Link href="/">
                <a>Home</a>
            </Link>
            <h1>About Next.js</h1>
            <p>Next.js - Server-side rendering for React.</p>
        </div>
    )
}

export default About;
```

ထူးခြားချက်အနေနဲ့ `Link` ကို Import ထားတာကို သတိပြုပါ။ ပြီးတဲ့အခါ `<Link>` Component ကို လည်း အသုံးပြုထားပါတယ်။ `href` အနေနဲ့ `/` ကိုပေးထားလို့ Home ဆိုတဲ့အဓိပ္ပါယ်ပါ။ တစ်နည်းအားဖြင့် `index.js` ကို ညွှန်းထားတာပါ။ သူ့အထဲက `<a>` Element မှာတော့ `href` ထပ်ပေး စရာ မလိုတော့ပါဘူး။ စမ်းကြည့်နိုင်ဖို့ Browser URL Bar မှာ အခုလို ကိုယ်ဘာသာ ရိုက်ထည့်လိုက်ပါ။

http://localhost:3000/about

ရလဒ်က ဒီလိုဖြစ်မှာပါ။



Home ကို သွားလို့ရတဲ့ Link တစ်ခုပါပြီး နှိပ်လိုက်ရင် Home Page ကို ပြန်ရောက်သွားတယ် ဆိုတာကို တွေ့ရနိုင်ပါတယ်။ ပိုအဆင်ပြေသွားအောင် `Nav.js` အမည်နဲ့ ဖိုင်တစ်ခုကို pages ဖိုဒါထဲမှာပဲ ဒီလိုရေးပေးပါ။

```
import Link from 'next/link';

const Nav = props => {
    return (
        <ul>
            <li><Link href="/"><a>Home</a></Link></li>
            <li><Link href="/about"><a>About</a></Link></li>
        </ul>
    );
};

export default Nav;
```

ဒါဟာ Home (index.js) နဲ့ About (about.js) တို့ကို အပြန်အလှန် သွားလို့ရတဲ့ Menu တစ်ခု ဖြစ်သွားတာပါ။ သူ့ကို `index.js` နဲ့ `about.js` တို့က အခုလို ခေါ်သုံးလို့ရပါတယ်။

```
// index.js
import Nav from './Nav';

const Home = props => {
    return (
        <div>
            <Nav />
            <h1>Welcome to Next.js</h1>
            <ul>
                <li>Alice</li>
                <li>Bob</li>
            </ul>
        </div>
    )
}

export default Home;
```

```
// about.js
import Nav from './Nav';

const About = props => {
    return (
        <div>
            <Nav />
            <h1>About Next.js</h1>
            <p>Next.js - Server-side rendering for React.</p>
        </div>
    )
}

export default About;
```

ပိုပြည့်စုံသွားအောင် အဲ့ဒီ Nav Menu ကို Header Layout Component လေးနဲ့ ပြဖို့အတွက် `Header.js` အမည်နဲ့ ဖိုင်တစ်ခုကို `pages` ဖိုဒါထဲမှာပဲ ထပ်ဆောင်းပြီး ဒီကုဒ်ကို ရေးပါမယ်။

```
const styles = {
    header: {
        padding: 10,
        background: 'cyan'
    }
}

const Header = props => {
    return (
        <div style={styles.header}>
            {props.children}
        </div>
    )
}

export default Header;
```

ဘာမှမဟုတ်ပါဘူး။ `props.children` ကို `<div>` တစ်ခုနဲ့ `style` တွေဘာတွေနဲ့ ထည့်ပြလိုက်တာပါပဲ။ သူ့ကို အသုံးပြုဖို့ဆိုရင် `index.js` ရဲ့ ကုဒ်ဖွဲ့စည်းပုံက ဒီလိုဖြစ်သွားပါလိမ့်မယ်။

```
import Header from './Header';
import Nav from './Nav';

const Home = props => {
    return (
        <div>
            <Header>
                <Nav />
            </Header>
            <h1>Welcome to Next.js</h1>
            <ul>
                <li>Alice</li>
                <li>Bob</li>
            </ul>
        </div>
    )
}

export default Home;
```

ဒါပါပဲ။ ဒီနည်းနဲ့ Next.js ကို အသုံးပြုပြီး Page တွေဆောက်လို့ရမယ်။ Component တွေ ခွဲထားပြီး လိုတဲ့အခါ ယူသုံးလို့ရမယ်။ Layout Component တွေဘာတွေ ဖန်တီးချင်ရင်လည်းရမှာဖြစ်ပါတယ်။ React ကို

သိထားပြီးသူတွေအတွက် ခက်ခက်ခဲခဲ ထပ်လေ့လာစရာ မလိုဘဲ အသုံးပြုနိုင်စေမယ့် နည်းပညာတစ်ခုပါ။

URL Parameter လို ကိစ္စတွေကျန်သေးသလို၊ Next.js ကို အသုံးပြုပြီး API တည်ဆောက်နိုင်ပုံတွေ ကျန်ပါသေးတယ်။ အခု ထည့်မပြောတော့ပါဘူး။ ထုံးစံအတိုင်း လိုအပ်တဲ့အခါ ကိုယ်တိုင်ဆက်လေ့လာမယ်ဆိုရင် React သာ ကြေညက်ပါစေ၊ အလွယ်တစ်ကူ ဆက်လေ့လာသွားလို့ ရနိုင်ပါလိမ့်မယ်။

ဒီအခန်းအတွက်လည်း ကုဒ်အပြည့်အစုံ ဒီမှာပေးထားပါတယ် – https://github.com/eimg/react-book

အခန်း (၁၄) – What's Next

အခုဆိုရင် ဒီစာအုပ်မှာ ထည့်သွင်းဖော်ပြလိုတဲ့ အကြောင်းအရာ စုံသလောက် ရှိသွားပါပြီ။ လို တို ရှင်း ဆို ပေမယ့် စာမျက်နှာတော့ (၁၀၀) ကျော်သွားပါတယ်။ ဒီအထိရောက်အောင် စိတ်ဝင်တစ်စား ဖတ်ရှုလေ့လာ ခဲ့တဲ့အတွက် ဂုဏ်ယူပါတယ်။ ဒီစာအုပ်ကနေ အသုံးဝင်ပြီး အကျိုးရှိတဲ့ အကြောင်းအရာတွေကို ရရှိလိမ့် မယ်လို့ မျှော်လင့်ပါတယ်။

ဒီအခန်းက နောက်ဆုံးပါ။ ရှေ့အခန်းတွေမှာ အကြောင်းကြောင်းကြောင့် ထည့်မဖော်ပြဖြစ်ခဲ့ပေမယ့် ထည့်သွင်းသတိပြုသင့်တာလေးတစ်ချို့ကို လက်စသတ် ဖော်ပြပေးချင်ပါတယ်။

Conditional Rendering

React Component တွေမှာ Loop တွေလုပ်ဖို့ `map()` တို့ `filter()` တို့ကို သုံးခဲ့ကြပါတယ်။ အခြေအနေပေါ် မူတည်ပြီး အလုပ်လုပ်ဖို့အတွက်တော့ `if()` Statement လို့ ရေးထုံးမျိုးကို ထည့်သွင်း ဖော်ပြခဲ့ခြင်း မရှိပါဘူး။ ထည့်ရေးလို့လည်း မရပါဘူး။ `if()` Statement အစား လိုအပ်တဲ့အခါ အခြေအနေပေါ် မူတည်ဖော်ပြ စေလိုရင် Ternary Operator ကို အသုံးပြုနိုင်ပါတယ်။ `condition ? true : false` ဆိုတဲ့ ရေးထုံးပါ။ Condition နောက်က Question Mark လိုက်ရပြီး True ဆိုရင် Question Mark နောက်က အလုပ်ကို လုပ်မယ်။ False ဆိုရင်တော့ Colon နောက်က အလုပ်ကို လုပ်မှာ ဖြစ်ပါတယ်။ ဒီရေးထုံးက Language အများစုမှာပါသလို JavaScript မှာလည်း ပါပါတယ်။ React Component တွေမှာ ဆိုရင်တော့ ဒီလိုဖြစ်မှာပါ။

```
<div>
  {
     type === 1
     ? <Button primary>Button</Button>
     : <Button secondary>Button</Button>
  }
</div>
```

Ternary Operator ကိုပဲ ဖတ်ရလွယ်အောင် ခွဲရေးလိုက်တာပါ။ နမူနာအရ `type` တန်ဖိုး 1 ဆိုရင် **<Button>** ကို `primary` **props** နဲ့ ဖော်ပြပြီး မဟုတ်ရင်တော့ `secondary` **props** နဲ့ ဖော်ပြစေထားတာ ဖြစ်ပါတယ်။ ဒီနည်းနဲ့ အခြေအနေပေါ်မူတည်ပြီး ပြစေချင်တာတွေကို စစ်ပြီးမှ ပြလို့ ရပါတယ်။

Fragments

Component တွေ တည်ဆောက်တဲ့အခါ Element တစ်ခုတည်းကိုသာ Return ပြန်ပေးရတယ်လို့ ပြောခဲ့ပါတယ်။ ဒါကြောင့် နှစ်ခုသုံးခုရှိလာတဲ့အခါ `<div>` တစ်ခုနဲ့ စုပြီးတော့ ပြန်ပေးကြပါတယ်။ `<div>` မသုံးချင်ရင် React ရဲ့ Fragment လို့ခေါ်တဲ့ လုပ်ဆောင်ချက်ကို သုံးနိုင်ပါတယ်။ ရေးနည်း နှစ်နည်း ရှိပါတယ်။ ဒီလိုပါ –

```
const App = props => {
  return (
     <React.Fragment>
        <h1>Title</h1>
        <p>Content</p>
     </React.Fragment>
  );
}
```

```
const App = props => {
  return (
     <>
        <h1>Title</h1>
        <p>Content</p>
     </>
  );
}
```

`<React.Fragment>` ကို သုံးလို့ရသလို ဘာမှမပါတဲ့ `<>` ကိုလည်း အတိုကောက်အနေနဲ့ သုံးလို့ရတဲ့ သဘောပါ။

state CRUD

နောက်တစ်ခုအနေနဲ့ state Data တွေ စီမံတဲ့အခါ `map()` နဲ့ `filter()` ကို Create, Read, Update, Delete လုပ်ငန်းတွေအတွက် အသုံးပြုနိုင်ပုံကို ဖော်ပြပေးပါမယ်။ ဒါကတော့ React နဲ့ တိုက်ရိုက်ဆိုင်တာ မျိုး မဟုတ်ဘဲ ရေးထုံးပိုင်းဆိုင်ရာ အကြံပြုချက် တစ်ခုပါ။ ဥပမာ – ဒီလို Data ရှိတယ်ဆိုကြပါစို့။

```
const users = [
  { id: 1, name: 'Alice', age: 22 },
  { id: 2, name: 'Bob', age: 23 },
];
```

ဒီထဲက `name` တွေကိုချည်းပဲ လိုချင်တယ်ဆိုရင် အခုလို ယူလို့ရနိုင်ပါတယ်။

```
const names = users.map(u => u.name);  // => [ Alice, Bob ]
```

တစ်ခုတည်းကို လိုချင်တယ်ဆိုရင် အခုလို ယူလို့ရနိုင်ပါတယ်။

```
const bob = users.filter(u => u.id === 2);

// => [{ id: 2, name: 'Bob', age: 23 }]
```

အသစ်ထပ်တိုးချင်ရင် Spread Operator အကူအညီနဲ့ အလွယ်တစ်ကူ တိုးလို့ရပါတယ်။ ဒါကိုတော့ ရှေ့ပိုင်းမှာလည်း ခဏခဏ တွေ့ခဲ့ပြီးသားပါ။ တစ်ခုပဲ သတိထားပါ၊ မူလ Data ထဲမှာ ထပ်တိုးလိုက်တာ မဟုတ်ပါဘူး။ ထပ်တိုးထားတဲ့ Data အသစ်ကို ပြန်ပေးတာပါ။

```
const result = [ ...users, { id: 3, name: 'Tom', age: 24 };
```

ပြန်ဖျက်ချင်တယ်ဆိုရင် `filter()` နဲ့ပဲ ဖျက်လို့ရပါတယ်။ ဒီမှာလည်း သတိထားပါ။ မူလ Data ထဲက ဖျက်တာ မဟုတ်ပါဘူး။ ဖျက်ထားတဲ့ Data Set အသစ်ကို ပြန်ပေးတာပါ။

```
const result = users.filter(u => u.id !== 2);
```

`id: 2` တန်ဖိုးရှိတဲ့ `user` ကို ချန်ပြီး ကျန်တာတွေ Filter လုပ်ယူမယ်ဆိုတဲ့ အဓိပ္ပါယ်ပါ။ Update လုပ်ချင်ရင်တော့ `map()` ကိုပဲ အသုံးပြုနိုင်ပါတယ်။

```
const result = users.map(u => {
  if(u.id === 1) u.age = 21;
  return u;
});
```

`id: 1` တန်ဖိုးရှိတဲ့ `user` ရဲ့ `age` ကို `21` လို့ပြင်လိုက်တာပါ။ ဒီရေးနည်းတွေကို အရမ်းအသုံးဝင်ပါတယ်။ `state` Data တွေကို Create, Read, Update, Delete လုပ်ငန်းတွေ လုပ်ဖို့လိုတိုင်း ဒီရေးနည်းတွေကိုသာ အသုံးပြုဖို့ အကြံပြုပါတယ်။

Build System

React ကုဒ်တွေကိုရေးဖို့အတွက် ပရောဂျက်ကို `create-react-app` နဲ့ တည်ဆောက်ပါတယ်။ `create-react-app` က ဘာတွေလုပ်ပေးသွားတာလည်း သိချင်တယ်ဆိုရင် သူ့ကိုမသုံးဘဲ အလားတူ စနစ်တစ်ခုကို ကိုယ်ဘာသာတစ်ခုလောက် အစအဆုံး တည်ဆောက်ကြည့်သင့်ပါတယ်။ ဒီစာအုပ်မှာတော့ စာမျက်နှာများနေလို့ ဒီအကြောင်းကို ထည့်မပြောတော့ပါဘူး။ ဒီလိပ်စာမှာ ရေးပြီးတင်ထားပေးပါတယ်။ လေ့လာကြည့်ဖို့ တိုက်တွန်းပါတယ်။

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

Deployment

React နဲ့ ကုဒ်တွေရေးပြီးနောက် အများသုံးဖို့စပေးတော့မယ်ဆိုရင် ဒီ Command လေး Run လိုက်ယုံပါပဲ။

```
>> npm run build
```

ဒါဆိုရင် `create-react-app` ပရောဂျက်ထဲမှာ `build` ဆိုတဲ့အမည်နဲ့ ဖိုဒါတစ်ခု ဝင်သွားပါလိမ့်မယ်။ လိုအပ်တာ အားလုံးပါဝင်ပြီး အသင့်သုံး ဖိုင်နယ်ရလဒ်ကို ရိုးရိုး HTML, CSS, JavaScript အနေနဲ့ ရပါတယ်။ အသုံးပြုနိုင်ဖို့ React လည်း ထပ်ထည့်စရာ မလိုပါဘူး။ NPM တွေဘာတွေလည်း မလိုတော့ပါဘူး။ အဲ့ဒီ `build` ဖိုဒါထဲက ဖိုင်တွေကို Publish လုပ်လိုက်ယုံပါပဲ။ React Native မှာဆိုရင်တော့ အရင်ဆုံး ပရောဂျက်ဖိုဒါထဲက `app.json` မှာ App အမည်တို့ Version နံပါတ်တို့ကို စိတ်တိုင်းကျ ပြင်ပါ။ ပြီးရင် ဒီ Command တွေကို Run ပေးလိုက်ရင် ရပါပြီ။

```
>> expo build:android
>> expo build:ios
```

ဒါပေမယ့် Mobile App တွေ Build လုပ်ရတာက နည်းနည်းအလုပ်ရှုပ်ပါတယ်။ Play Store တို့ App Store တို့မှာ တင်လို့ရဖို့အတွက် နောက်ဆက်တွဲ လုပ်ပေးရမှာတွေ ရှိလာနိုင်လို့ အသေးစိတ်ကို ဒီမှာ ဆက်လေ့လာရမှာပါ။

– https://docs.expo.io/versions/latest/distribution/building-standalone-apps/

Expo နဲ့ ပက်သက်ရင် သတိပြုသင့်တာကတော့ Build လုပ်လိုက်တဲ့အခါ UI နဲ့ JavaScript Bundle ခေါ် နောက်ကွယ်က အလုပ်လုပ်တဲ့ကုဒ်ကို ခွဲမြင်ဖို့ လိုပါတယ်။ UI ကိုသာ App အနေနဲ့ ထုတ်လိုက်ပြီး JavaScript Bundle ကိုတော့ Expo ရဲ့ CND Cloud Server ပေါ်မှာ တင်ပေးလိုက်မှာပါ။ ဒါကြောင့် နောက်ပိုင်း ကုဒ်တွေ ပြင်လိုက်ရင် User က App ကို Update လုပ်စရာမလိုဘဲ ပြင်ဆင်မှုကို အလိုအလျောက် ရရှိနိုင်ပါတယ်။ အလုပ်လုပ်ပုံအသေးစိတ်ကိုတော့ ဒီမှာ ဆက်လေ့လာနိုင်ပါတယ်။

– https://docs.expo.io/versions/latest/workflow/how-expo-works/

UI Frameworks

ဒီစာအုပ်မှာသာ အခြေခံတွေနားလည်အောင် ဖော်ပြခဲ့ပေမယ့် လက်တွေ့မှာ Component အားလုံးကို ကိုယ်တိုင် ရေးစရာမလိုပါဘူး။ အသင့်သုံး UI Framework တွေ ရှိပါတယ်။ UI Framework တွေက ပေးတဲ့ အသင့်သုံး Component တွေကို ပေါင်းစပ်ပြီးတော့ ကိုယ်လိုချင်တဲ့ App ကို အလွယ်တကူ တည်ဆောက်နိုင်ပါတယ်။

လက်ရှိဒီစာရေးနေချိန်အထိ React အတွက် အသုံးအများဆုံး UI Framework တွေကတော့ Material UI နဲ့ Ant UI တို့ဖြစ်ပါတယ်။

– Material UI – https://material-ui.com/
– Ant UI – https://ant.design/

React Native အတွက်ဆိုရင်တော့ React Native Element, Native Base နဲ့ React Native Design System (RNDS) တို့ရှိပါတယ်။

– Elements – https://react-native-elements.github.io/react-native-elements/
– Native Base – https://nativebase.io/
– RNDS – https://nativebase.io/

လေ့လာကြည့်ပါ။ အသုံးပြုရ လွယ်ကူပြီး အများကြီး အသုံးဝင်တယ်ဆိုတာကို တွေ့ရပါလိမ့်မယ်။

နိဂုံးချုပ်

ဒီစာအုပ်ဟာ မှတ်မှတ်ရရ ကူးစက်မြန်ကပ်ရောဂါတစ်ခုဖြစ်တဲ့ Covid-19 ကိုရိုနာဗိုင်းရပ် ဖြစ်ပွားနေလို့ ရောဂါကူးစက်မှု ကာကွယ်တာဆီးနိုင်ရေးအတွက် တစ်ကမ္ဘာလုံးအတိုင်းအတာနဲ့ Social Distancing နဲ့ Work From Home လှုပ်ရှားမှုကို ဆောင်ရွက်နေစဉ်ကာလ၊ အလုပ်နဲ့ သင်တန်းကျောင်းကို ခဏပိတ်ထားပြီး အိမ်ထဲမှာပဲ နေစဉ်မှာ ရေးဖြစ်ခဲ့တာပါ။

ဒီစာအုပ်ကို ရေးမယ်ဆိုတော့ စိတ်ဝင်စားကြသူများက ကြိုတင်အမှာစာတွေ ပေးပို့ကြတာ အမှာစာပေါင်း (၁၉၀၀) ကျော် လက်ခံရရှိလို့ ရေးရတဲ့သူအတွက် တစ်အားပါပဲ။ ဒီစာအုပ်ကို PDF Ebook အနေနဲ့သာ ထုတ်ဝေဖို့ မူလကရည်ရွယ်ခဲ့ပေမယ့် ကြိုတင်အမှာစာ ပေးပို့ကြသူတွေရဲ့ အကူအညီနဲ့ ပုံနှိပ်စာအုပ်အဖြစ်ပါ ထုတ်ဝေဖြစ်သွားပါတယ်။

တစ်ဦးချင်းစီကို ကျေးဇူးတင်စကားမပြောနိုင်ပေမယ် ဒီနေရာကနေပဲ ဝိုင်းဝန်းပံ့ပိုးကြသူ အားလုံးကို ကျေးဇူးတင်ပါကြောင်းပြောရင် နိဂုံးချုပ်အပ်ပါတယ်ဗျာ။ အားလုံးပဲ ကပ်ဘေးတွေကို ကျော်လွှားနိုင်ပြီး ကိုယ်စိတ်နှစ်ဖြာ ကျန်းမာချမ်းသာ ကြပါစေ။

အိမောင် (Fairway)

၂၀၂၀ ခုနှစ်၊ ဧပြီလ (၃) ရက်နေ့တွင် ရေးသားပြီးစီးသည်။