Değişkenler

Değerleri üzerinde tutmamıza yararlar.

```
> var benimAdim = "Fatih"
< undefined
> benimAdim
< 'Fatih'
> console.log(benimAdim)
  Fatih
< undefined
> benimAdim+" Baytar";
< 'Fatih Baytar'
>
```

```
> var sehir = "İstanbul";
< undefined
> sehir = 15
< 15
> sehir = false
< false
>
```

Daha önce var ile tanımlanmış birşey let ile tekrar tanımlanamaz. Ancak let olarak tanımlanmış birşey tekrar tanımlanabilir. Aşağıdaki örnekte sehir isimli değişken var olarak

```
> let sehir = "Ankara";
✕ Uncaught SyntaxError: Identifier 'sehir' has already been declared
> let baskent = "Ankara";
< undefined
> let baskent = "İstanbul";
< undefined
> let baskent = "ankara";
< undefined
>
```

öncesinden tanımlanmıştı.

```
> const pi = 3.14
< undefined
> pi
< 3.14
> pi = 3.14
✕ ▸Uncaught TypeError: Assignment to constant variable.
      at <anonymous>:1:4
>
```

const ile tanımlanmış bir değişken sabittir. Tekrardan değeri değiştirilemez.

Koşul ifadeleri

Koşul ifadeleri diğer dillerde olduğu gibi yapılır.

```
> var sayi = 15;
< undefined
> if(sayi<12){
      console.log("Sayımız 12'den küçük");
  }else{
      console.log("Sayımız 12'ye eşit ya da büyük");
  }
  Sayımız 12'ye eşit ya da büyük
< undefined
>
```

```
> sayi
✕ ▸Uncaught ReferenceError: sayi is not defined
      at <anonymous>:1:1
> if(sayi)
      console.log("Sayı mevcut");
✕ ▸Uncaught ReferenceError: sayi is not defined
      at <anonymous>:1:1
> if(sayi)
      console.log("Sayı mevcut");
  else
      console.log("Sayı mevcut değil");
✕ ▸Uncaught ReferenceError: sayi is not defined
      at <anonymous>:1:1
```

Bir nesnenin varlığını kontrol etmek için de (undefined'tan farklı olup olmadığını) if yapısı aşağıdaki gibi kullanılabilir.

```
> var sayi;
< undefined
> if(sayi)
      console.log("Sayı mevcut");
  else
      console.log("Sayı mevcut değil");
  Sayı mevcut değil
< undefined
>
```

Dikkat edilmesi gereken konulardan birisi eşitlik operatörüdür.

```
> var sayi = 15;
< undefined
> sayi = 22;
< 22
> if(sayi=='22')
      console.log(`Sayi ${sayi} dir`);
  Sayi 22 dir
< undefined
>
```

Örnekte görüldüğü gibi == ifadesi tip kontrolü yapmadan işlem yapar. Tip kontrolü yapmak isterseniz === kullanılmalıdır. Yukarıdaki örnekte dikkat edilmesi gereken diğer husus string ifadeleri kullanırken string literal'in kullanılmasıdır. String literal'in bir başka kullanım örneği şudur:

```
> `Merhaba
  Nasılsın
  Nasıl gidiyor
  2 nin karesi ${Math.pow(2,2)} dir`
< 'Merhaba\nNasılsın\nNasıl gidiyor\n2 nin karesi 4 dir'
>
```

Veri yapıları

Javascript varsayılan olarak JSON (javascript object notation) gösterimini kullanılır. Burada süslü parantezler içerisinde key value çiftleri ile veriler gösterilir.

```
> var person1 = {Ad:"Fatih", Soyad:"Baytar", Yas : 40, Sinif:"5A"};
< undefined
> person1.Ad
< 'Fatih'
> person1
< ▶ {Ad: 'Fatih', Soyad: 'Baytar', Yas: 40, Sinif: '5A'}
> person1.Ad+" "+person1.Yas
< 'Fatih 40'
> person1.Yas/2
< 20
>
```

Diziler

```
> var arr = new Array()
< undefined
> arr.push('Merhaba')
< 1
> arr
< ▶ ['Merhaba']
> var baskaBirArray = []
< undefined
> baskaBirArray.push(5)
< 1
> baskaBirArray
< ▶ [5]
```

Döngüler

```
> for(i=0;i<50;i++)
      console.log(i)
  0
  1
  2
  3
  4
  5
  6
  7
  8
  9
```

```
> sayi
< 22
> while(sayi<100)
  {
      console.log(sayi);
      sayi++}
  22
  23
  24
  25
  26
  27
  28
```

Fonksiyonlar

Javascript programlamanın çekirdeğini oluştururlar. Modülerlik sağlalar (Yazılımların bölümlere ayrılarak incelenmesini kodlanmasını kolaylaştırırlar)

```
function Greetings(adim){
    console.log('Merhaba '+adim);
}
Greetings('Fatih')
Merhaba Fatih
```

Nesneler

```
var person1 ={
    Ad:'Tarık',
    Soyad:'Bey',
    SelamVer:function(){
        return this.Ad+" "+this.Soyad
    }
```

```
}

person1.SelamVer()

'Tarık Bey'
```

NPM, WEBPACK, BABEL

Node: Sunucu taraflı javascript kodları da yazmamızı sağlayan bir teknolojidir. Node kurulumu yaparak aynı zamanda npm (node package manager) kurulumu sağlanmış olur. npm: Hazır paketleri kendi bilgisayarımıza indirmenin en hızlı yollarından birisidir.

```
npm -v
```

Webpack: Modern javascript kütüphaneleri için üretilen bir module bundler'dır (modül paketleyicisidir) diyebiliriz. Webpack bir projede çalıştırıldığında, projenin ihtiyaç duyabileceği her modül tipini alan bir dependency graph(bağımlılık grafiği) oluşturur ve bu grafiğin işlenmesi sonucu çıktı olarak bir uygulama paketi üretir.

Node.js modüllerinin aksine, webpack modüllerinde modüller arası bağımlılıklar farklı şekillerde ifade edilebilir. Örneğin:

- ES2015'teki `import` ifadesi,
- CommonJS'teki `require()` ifadesi,
- AMD'deki `define` ve `require` ifadeleri
- CSS/SASS/LESS dosyalarındaki `@import`ifadesi,
- CSS'te görsel yükleme için oluşturulan oluşturulan `url(...)` fonksiyonu ve HTML'deki `<img src=...>` ifadesi ile modül bağımlılığı sağlanabilir.

Babel Nedir?

Babel, ECMAScript 2015+ kodunu mevcut ve eski tarayıcılarda veya ortamlarda geriye dönük olarak uyumlu bir JavaScript sürümüne dönüştürmek için kullanılan bir araçtır.

Yeni özellikler standartlaştırıldığında, tarayıcılar yavaş yavaş yeni özellikleri benimsemeye başlar. Bu yüzden Babel kullanmak bizler için bir nevi zorunluluk.

Babel için yaptığımız konfigürasyon ise, hangi JavaScript özelliklerini destekleyeceği ve hangi browser'lar için çıktı üreteceğimizden ibaret.

Babel Ne Yapar?

Babel bir **JavaScript** derleyicisidir. Google, Facebook, *Hepsiburada*, Netflix ve diğer büyük yüzlerce şirkette kullanılıyor. Hatta büyük şirketleri es geçip, bütün JavaScript ile çalışan modern uygulamaların, arkasında Babel olduğunu söylersek iddialı olmaz sanırım.

Babel sadece ES6 kullanmamızı sağlamakla kalmıyor, aynı zamanda **polyfill** desteği ile ES7, JSX vb. başka standartlarıda kullanmamıza imkan tanıyor.

Bunlar yeterli değilse, Babel, **TC39** *(JavaScript'in tasarım ve geliştirmesine liderlik eden teknik komite/grup)* ile yakın bir uyum içerisinde çalışmakta ve **TC39** topluluğu tarafından dile dahil olup olmayacağı belirlenmemiş özellikleri, JavaScript dilinde kullanmamıza imkan sağlıyor. Harika değil mi?

Şimdi babel'i hızlıca görebileceğimiz bir örnek yapalım.

```
<div id="output"></div>

<!-- Load Babel -->

<!-- v6 <script src="https://unpkg.com/babel-standalone@6/
babel.min.js"></script> -->

<script src="https://unpkg.com/@babel/standalone/babel.min.js"></
script>

<!-- Your custom script here -->

<script type="text/babel">

const getMessage = () => "Hello World";

document.getElementById('output').innerHTML = getMessage();

</script>
```

Burada görüldüğü babel compalier'ı cdn'den çağırdık ve kullandık. İlgili kısmı yorum satırına alıp deneyin. Kodların çalışmadığını hiçbirşeyin ekrana yazılmadını göreceksiniz.

Fonksiyonel Programlama Nedir?

Fonksiyonlara büyük önem veren bu yaklaşımda, fonksiyonlar bir değişkene atanabilir, fonksiyonlar anonim olarak hemen kullanılabilir, nesnelerden ziyade verileri temsil edebilir. Aşağıda yer alan kodda bu görünmektedir.

```
var log = function(message) {
    console.log(message)
};
    log("In JavaScript functions are variables")
```

```
const obj = {

message: "They can be added to objects like variables",
    log(message)
    {
       console.log(message);
    }
}
   obj.log(obj.message)
```

Immutability

Mutate: değişebilirlik iken immatable değişememezliği temsil eder. Functional programlamada data immutable'dır, değişmemelidir. Tabi bu değişme işlemi nesnelerin birbirine bağlı olmasından kaynaklı olarak gerçekleşmektedir. Aşağıdaki örneği inceleyelim:

```
let color_lawn = { title: "lawn",
color: "#00FF00",

rating: 0 }
```

Burada color_lawn nesnesinin rating değerini değiştiren bir function yazarsak.

```
function rateColor(color, rating)
{
    color.rating = rating
```

```
    return color
}


console.log(rateColor(color_lawn, 5).rating) //5

console.log(color_lawn.rating) //5
```

Fonksiyon kendisine gönderilen color nesnesini ve rating değerini işleme alıp, color nesnesinin rating değerini rating olarak ayarlıyor. Buraya kadar herşey bilindiği gibi. Ardından rateColor function çağrılıyor ve kendisine color_lawn nesnesi veriliyor ve 5 ile işlem yapılması isteniyor. Bu durumda kendisine gönderilen color_lawn nesnesinin rating değerini 5 yapıyor ve geriye döndürüyor. Bu satırda da herşey yolunda. Ancak bir sonraki satırda görüyoruz ki function kendisine gönderilen color_lawn nesnesinin değerini değiştirmiş. Bu genellikle istenen bir durum değildir. Çünkü function geriye bir nesne döndürmekte bu function dışarıdaki nesneleri etkilemesi istenmemektedir.

Bir fonksiyonun bir nesnenin değerini manipüle etmemesi için kopya çıkarmak en basit yöntemlerden biridir. Aşağıdaki fonksiyonu tekrar inceleyelim.

```
var rateColor = function(color, rating) {
    return Object.assign({}, color, {rating:rating})
}
console.log(rateColor(color_lawn, 5).rating) // 5

console.log(color_lawn.rating) // 4
```

Görüldüğü gibi burada Object.assign fonksiyonundan faydalandık. Böylece boş bir nesneye object assign işlemi yapılarak aslında yepyeni bir nesne yaratılıp geriye döndürülüyor. Böylece kendisine gönderilen nesne manipüle olmamış oluyor.

ES6 da bu şekilde yazılırken ES7 de yeni bir yazım şekli mevuttur ve biz daha çok bu yazım şeklini kullanıyor olacağız.

```
const rateColor = (color, rating ) =>

({

    …color,

    rating

})
```

Burada …color nesnesinin mevcut herseyini yanında, rating değeri de değiştiriliyor anlamı taşır.

Objelerdeki immutable olayını listelerde de görmemiz mümkün.

```
let list=[{title:'Junior Software Developer'}, {title:'Mid Level
Software Developer'}, {title:'Senir Software Developer'}]
```

addTitle adında bir fonksiyonumuz bu Colors adındaki listeye yeni bir eleman ekler.

```
var addTitle = function(title, titles) {

    titles.push({ title: title })
    return titles;

}

console.log(addTitle("Developer Trainer", list).length) //4

console.log(list.length) //4
```

Görüldüğü gibi kendisine parametre olarak gönderdiğimiz list title listesi manipüle oldu.

```
<script type="text/babel">

    let list=[{title:'Junior Software Developer'}, {title:'Mid Level
Software Developer'}, {title:'Senir Software Developer'}]


    const addTitle = (title, array) => array.concat({title});

    console.log(addTitle("Developer Trainer",list)) //4

    console.log(list.length); //3

</script>
```

Burada listeyi kopyalayan şey ise array.concat metodu olmuştur.

ES7 ile birlikte de bu fonksiyon kısaca

```
const addTitle = (title, list) => [...list, {title}]
```

Arrow function'ına dönüştü. (...list içindeki herseye al, sonuna title'ı ekle ve geri dönder şeklinde)

Data Transformations

Veriler üzerinde işlemlerimizi yaparken (arama, çıkarma, her veriyi dönme gibi) çok sık kullanacağımız metotları bu başlık altında topluyorum. Daha sonrasında da bir çok işlemde bunları kullanacağız.

```
const iller = ["İstanbul","Ankara","Adana","Adıyaman","İzmir","Bursa","Diyarbakır"];
```

```
var basHarfIller = iller.filter(il => il[0]==="İ")
console.log(basHarfIller);
```

filter metodu bir listeden bazı verileri filtrelemek için kullanılan bir metottur. Geriye her zaman bir liste döner.

```
  const removeIl = (remove, list)=>
        list.filter(il=>il !== remove);
    console.log(removeIl("Ankara",iller)); //['İstanbul', 'Adana', 'Adıyaman', 'İzmir',
'Bursa', 'Diyarbakır']
```

Map fonksiyonu da bütün elemanları tek tek gezip manipüle işlemlerinde kullanılır.

```
 const turkiyeEklenmis = iller.map(il=>`${il} Türkiye`);
  console.log(turkiyeEklenmis); //['İstanbul Türkiye', 'Ankara Türkiye', 'Adana Türkiye',
'Adıyaman Türkiye', 'İzmir Türkiye', 'Bursa Türkiye', 'Diyarbakır Türkiye']
```

Reduce fonksiyonu ise aldığı ikili parametrelerden birisi kendimizin belirlediği, diğeri ise listenin tüm elemanlarını kasteden bir parametre olarak kullanılabilir. Böylece elemanların her biri gezilirken belirlediğimiz değer de içerde kullanılabilir. Reduce ikinci parametre olarak initialValue değerini alır (Max değerinin başlangıç değeri). Geriye döndürülen değerin maxAge'e eşitlendiğine dikkat edin. Bu hali ile iki farklı işleve sahip metot olarak işimizi görmektedir.

```
const ages = [21,18,42,40,64,63,34];

    const maxAge = ages.reduce((max,age) =>{
        console.log(`${age} > ${max} = ${age > max}`);
        if (age > max){
            return age;
        }else{
            return max;
        }
    },0)
    console.log('maxAge',maxAge);


    // 21 > 0 = true
    // 18 > 21 = false
    // 42 > 21 = true
    // 40 > 42 = false
    // 64 > 42 = true
    // 63 > 64 = false
    // 34 > 64 = false
    // maxAge 64
```

Higher-Order Functions

Bu fonksiyonlar, fonksiyonel programlamanın önemli bileşenleridir. Bir higher-order fonksiyon başka fonksiyonları manipüle eder. Bir higher-order fonksiyon parametre olarak fonksiyon alır, geriye döner ya da ikisi de.

Array.map, Array.filter ve Array.reduce gibi fonksiyonlar higher-order fonksiyonlara örnektir. Aşağıda da küçük bir örneğini görmemiz mümkündür.

```
const kosulaGoreCagir = (kosul, fnDogru, fnYanlis) =>
    (kosul) ? fnDogru() : fnYanlis()

const gosterMerhaba =()=> console.log("Merhaba!!!");
const gosterYetkisiz =()=> console.log("Yetkiniz yok");

kosulaGoreCagir(true, gosterMerhaba, gosterYetkisiz) //Merhaba
kosulaGoreCagir(false, gosterMerhaba, gosterYetkisiz) //Yetkiniz yok.
```

Recursion

Kendini çağıran fonksiyonlar ihtiyaç halinde kullanılabilir. Kısa bir kullanım örneği aşağıdaki gibidir.

```
const countdown = (value, fn) =>{
            fn(value)
        return (value>0)? countdown(value-1,fn) : value;
    }
    countdown(10, value => console.log(value));
    //10
    //9
    //8
    //7
    //6
    //5
    //4
    //3
    //2
    //1
    //0
```

Dikkat: Tarayıcı Call Stack limitlerini aştığınızda burada tarayıcı size izin vermeyebilir. Yaptığınız bu işlemi tehdit olarak algılayabilir.

PURE REACT

React'i bir javascript kütüphanesi olarak düşünürsek başka bir teknolojiye müdahale etmeden (örneğin JSX gibi detaylarını ilerde göreceğiz) de HTML DOM elemanlarını manipülasyonunda kullanabiliyoruz. Burada kısaca bu müdahaleden biraz bahsedip, asıl popüler olan bu işin en iyi kullanım şekli olan (Best Practice) JSX ile kullanımına bakacağız.

React pure için cdn'den gerekli kütüphaneleri çağırıyorum. https://reactjs.org/docs/cdn-links.html adresinden cnd'lere erişebilirsiniz (google react cdn araması). Şimdiye kadar javascript fonksiyonlarını compile etmesi için babel ile çalışmıştık artık react'in fonksiyonlarına ve diğer yapılarına erişmemiz için cdn den direkt react i çağırıyoruz.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Test</title>
</head>

<body>
    <div id="react-container">
        <!-- Target container  -->
    </div>
    <script crossorigin src="https://unpkg.com/react@17/umd/react.development.js"></script>
    <script crossorigin src="https://unpkg.com/react-dom@17/umd/react-
dom.development.js"></script>

    <script>
        //Pure React and Javascript Kodları
    </script>
</body>

</html>
```

Virtual DOM Nedir ?

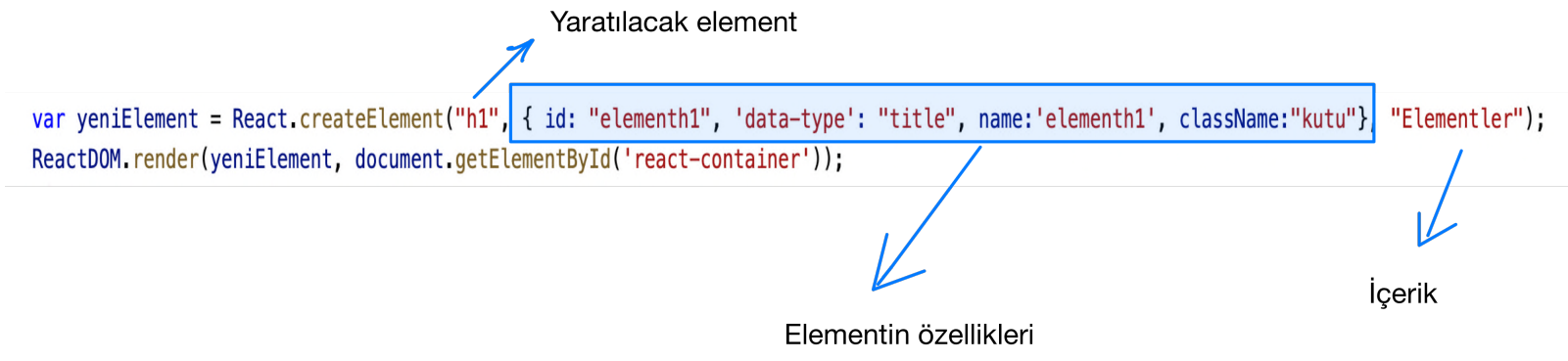
Bilindiği üzere HTML DOM elementleri ile çalışır ve html yüklendiğinde bu DOM elementleri de yüklenir ve birlikte render edilir.

Javascript'de de bu DOM elemanlarını manipüle işlemleri için bir API bulunmaktaydı. Hatırlarsınız ki document.createElement bir element oluşturur, oluşturulan bir element de document.AppendChild olarak da bir yerlere iliştirilirdi. DOM elementlerini manipüle etmekte de işe yarayan bu API çok hantal olarak iş yapmaktaydı.

React DOM elementleri de bunlara benzer ancak yapısal olarak tamamen farklıdırlar. Bir React öğesi, gerçek DOM öğesinin nasıl görünmesi gerektiğinin bir açıklamasıdır. Diğer bir ifadeyle, React elementleri browser elementlerinin "nasıl yaratılması gerektiği" talimatlarıdır.

h1 elementi oluşturmak için React ile aşağıdaki kod kullanılır.

```
var yeniElement = React.createElement("h1", { id: "elementh1", 'data-type': "title",
name:'elementh1', className:"kutu"}, "Elementler");
ReactDOM.render(yeniElement, document.getElementById('react-container'));
```



Daha komplike ve içeriği olan elementler için şu kullanılabilir:

```
var yeniElement = React.createElement(
            "ul",
            null,
            React.createElement('li',null,'Elma'),
            React.createElement('li',null,'Armut'),
            React.createElement('li',null,'Şeftali'),
            React.createElement('li',null,'Kayısı'),
            React.createElement('li',null,'Üzüm'),
            React.createElement('li',null,'Muz'),
            React.createElement('li',null,'Kivi'),
        )
        ReactDOM.render(yeniElement, document.getElementById('react-container'));
```

Görüleceği gibi içerikte de yeni elementler yaratılabilir.

```
var sehirler = ["İstanbul","İzmir","Ankara","Adana"];

        var yeniElement = React.createElement(
            "ul",
            null,
            sehirler.map((sehir,index)=>
            React.createElement("li",{key:index},sehir))
        )
        ReactDOM.render(yeniElement, document.getElementById('react-container'));
```

Burada dikkat etmemiz gereken şeylerden birisi, bu şekilde tekrar eden elementlerde hızı artırmak için bir indexleme yapmamız gerekiyor. Yapmazsanız zaten bu konuda hata kısmında yazmamız gerektiği söyleniyor. Yukarıdaki örnekte de her bir üretilen li elementi için key anahtarına elementin index değeri atanıyor. İndex değerinin benzersiz olması burada önem taşıyor.

Tüm bunlar React'in virtual DOM elementleri olarak geçiyor. Örneğin örneklerde bir elemente class ataması yaparken virtual dom da bunu kullanamıyoruz, bunun için className kullanmak zorundayız (Hata olarak karşımıza çıkar aksi durumda)

COMPONENTS

Genellikle bizler yazdığımız bazı şeyleri tekrar kullanılabilir olması için çalışırız. Bu istekler zamanlar yazılım standartlarına da eklenmiştir. React'te de bunu her zaman isteriz. İşte React'te yeniden kullanılabilirliği(resuable) arttırmak için componentler kullanılır.

ES6 ile hayatımıza giren createClass kavramı, react tarafında da kullanılmaya başlandı. Modülerliği ve yeniden kullanılabilirliği artırmak adına yapılan bu girişimin örneklerine bakalım. Tabi bu kodlar ES6 tarafında çalışacaktır. Bizler şu an ES7 ile çalışıyoruz.

```
const alisverisListesi = React.createClass({

    displayName:'AlışVerişListesi',
    render(){
        return React.createElement('ul',{'className':'alisverisListesi'},
        React.createElement('li',null,'1 kg Soğan'),
        React.createElement('li',null,'1 kg Patates'),
        React.createElement('li',null,'2 kg Domates'),
        React.createElement('li',null,'1 kg Patlıcan'),
        React.createElement('li',null,'0.5 kg Sarımsak'),
        React.createElement('li',null,'1 adet Yeşil Soğan'),
        React.createElement('li',null,'0.5 kg Salatalık')
        )
    }
});

const list = React.createElement(alisverisListesi,null,null);
ReactDOM.render(
    list,
    document.getElementById('react-container')
)
```

Bu kodların ES7'deki karşılığı ise şu şekildedir:

```
class AlisverisListesi extends React.Component{
       render(){
           return React.createElement('ul',{className:'aliverisListesi'},
           React.createElement('li',null,'1 kg Patates'),
           React.createElement('li',null,'1 kg Patates'),
           React.createElement('li',null,'2 kg Domates'),
           React.createElement('li',null,'1 kg Patlıcan'),
           React.createElement('li',null,'0.5 kg Sarımsak'),
           React.createElement('li',null,'1 adet Yeşil Soğan'),
           React.createElement('li',null,'0.5 kg Salatalık'))
       }
   }
   var yeniElement = React.createElement(AlisverisListesi,null,null);
   ReactDOM.render(yeniElement,document.getElementById('react-container'));
```

Stateless functional components

İlerde ayrıntılarını da göreceğimiz stateless functional componentler kendi içlerinde state bulundurmazlar. Bir sınıf gibi çalışmadıkları gibi (yukarıdaki örnekte olduğu gibi) render işlemleri de yoktur. Aşağıdaki örneği inceleyelim:

```
    const { render } = ReactDOM;
        const AlisverisListesi = ({ list }) => React.createElement('ul', null,
list.map((item, i) =>
            React.createElement('li', { key: i }, item)
        )
        )
        const AlisVerisFactory = React.createFactory(AlisverisListesi)
        const list = [
            "1 kg Patates",
            "1 km Domates",
            "1 kg Soğan",
            "0.5 kg Sarımsak",
        ]
        render(
            AlisVerisFactory({ list }),
            document.getElementById('react-container')
        )
```

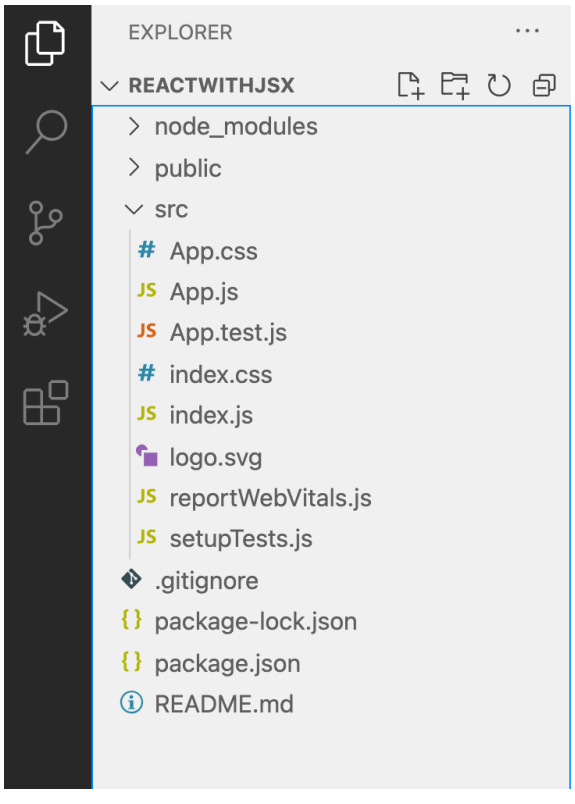
State yapısını ilerde göreceğiz. Burada kodların bir önceki örneğe göre farklı olduğunu görebiliyoruz.

REACT VE JSX

Buraya kadarki kısımlarda hep işlerin eski yollarını gördük. Genellikle işlerimizi daha modüler ve daha hızlı yapmak isteriz. Bu yüzden React tarafında harika bir özellik mevcut. JSX. Bu bildiğimiz html e çok benziyor ama html olmadığı asla akıllardan çıkarılmamalı.

Artık boilerplate templatimizi de kullanma zamanımız geldi. Dosya yapımızı burada da inceleyeceğiz. Hemen bir klasöre konuşlanıp terminalden şu kodumuzu yazıyoruz:

```
npx create-react-app .
```



Kurulum işlemleri bittikten sonra (ilk kurulum için bazı paketlerin indirilmesi zaman alacaktır)

Kurulum bitiminde node_modules klasörü paketlerimizin bulunduğu klasörü, src kaynaklarımızın bulunduğu klasörü, public ise temel web kaynaklarımızın (resimler, css dosyaları gibi) bulunduğu klasörü temsil etmektedir.

Şimdi ise daha önce yaptığımız alışverişlistesi örneğini burada da hızlıca yapalım.

Src klasörüne component'lerimizi yazacağımız bir klasör koyuyoruz.

```
const AlisverisListesi = (props) =>(
    <ul>
        <li>1kg Patates</li>
        <li>1kg Domates</li>
        <li>1k Soğan</li>
    </ul>
)
export default AlisverisListesi;
```

Hızlıca kendime bir functional komponent oluşturuyorum. En altta yazdığım export default bu komponentin bir paket olarak dışarı varsayılan olarak açılmasını sağlıyorum. Burada birden fazla komponent yaratıp sadece birinin bu isimde default olarak dışarı açılmasını istiyorum (backend yazılımlarda kullandığımız public ifadesi gibi). Npm start ile uygulamayı çalıştırıyoruz. Tüm uygulamanın başladığı yer App.js komponentidir. Kendi oluşturduğum AlisverisListesi komponentimi de buraya yazıyorum.

```
import AlisverisListesi from './components/AlisverisListesi';

function App() {
  return (
    <div className="App">
      <AlisverisListesi/>
    </div>
  );
}
export default App;
```

App.js in son hali yukarıdaki gibi olacaktır. Yukarıda import AlisverisListesi ifadesi ise ilgili komponenti içeri almamızı sağlıyor. Npx kurulumundan gelen app.css stil dosyasını kaldırdık.

```
const AlisverisListesi = (props) => {
    var list = ["1kg Patates", "1kg Domates"]
    return (
        <ul>
            {list.map((l, i) =>
                (<li key={i}>{l}</li>))
            }
        </ul>
    )
}
export default AlisverisListesi;
```

Alışveriş listesini bir listeden de çekebiliyoruz. Böylece burada süslü parantezler içerisinde değişkenler tanımlayıp bunları return ifadesi içerisinde kullanabiliyoruz. Return ifadesi içerisinde JSX kodları yer alır. JSX html'in React tarafındaki karşılıklarıdır. Tabi bu komponenti functional komponent olarak yapmak zorunda değiliz. Bir class component oluşturmak isteseydik kodlar aşağıdaki gibi olurdu:

```
import React, { Component } from 'react'

export default class AlisverisListesi extends Component {
    list = ["1kg Patates", "1kg Domates", "1kg Soğan"];
    render() {
        return (
            <ul>
                {
                    this.list.map((l, i) => (<li key={i}>{l}</li>))
                }
            </ul>
        )
    }
}
```

Bir class komponent'in de kullanımı yukarıdaki gibidir. react paketinden React, Component ifadelerini içerde miras yoluyla kendi sınıfımıza (komponent) kazandırmak için çağırıyoruz. Böylece kendi sınıfımıza bir component özelliği kazandırmış oluyoruz.

STATE & PROPS

Bir state bir komponent'in her yerinde kullanılabilecek bir değişken gibi düşünülebilir. Direkt olarak değer ataması yapılamaz ancak class componentlerde setState, functional componentlerde ise useState ifadeleri ile değerleri değiştirilebilir. Bir çok kullanıcı girişlerinde yoğun miktarda kullanılır.

```
export default class AlisverisListesi extends Component {
    state = {
        list:[]
    }
    componentWillMount() {
        this.setState({
            list: ["1kg Patates", "1kg Domates"]
        })
    }
    render() {
        return (
            <ul>
                {
                    this.state.list.map((l, i) => (<li key={i}>{l}</li>))
                }
            </ul>
        )
    }
}
```

componentWillMount ifadesine daha sonra life scycle bölümünde değineceğiz. Görüldüğü gibi bir state tanımlanıyor ve bu bir object. Bu objenin list adındaki dizisi (başlangıçta boş), componentWillMount içinde değeri değiştiriliyor. setState ifadesi ile değeri değiştirilen list, JSX içinde yukarıdaki gibi kullanılır.

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

const AlisverisListesi = (props)=>{
    const [list, setList] = useState([]);
    useEffect(() => {
        setList(["1kg Patates", "1kg Domates"]);
    }, [])
    return (
        <ul>
            {list.map((l,i)=>(
                <li key={i}>{l}</li>
            ))}
        </ul>
    )
}
export default AlisverisListesi;
```

Bir functional component yapısında da yukarıdaki gibi useState'ler state'leri manipüle etmek için kullanılır (useEffect yapılarından bahsedeceğiz). Görüldüğü üzere bir functional componentte bu işler biraz daha kolay olabiliyor. Bu yapıları çok sık şekilde kullanacağız.

Bir props ise; komponentlerin hiyerarşik ilişkisinden kaynaklı durumlarda yukarıdan aşağıya doğru veri aktarmak için sıklıkla kullanılır. Üst komponentte tanımlanan bir değer alt komponente aktarılmak istendiğinde readonly props yapıları kullanılır. Üst komponentten gelen bu yapılar manipüle edilemezler.

```
import { useState, useEffect } from "react";
import Liste from "./Liste";

const AlisverisListesi = (props)=>{
    const [list, setList] = useState([]);
    useEffect(() => {
        setList(["1kg Patates", "1kg Domates", "1kg Kabak"]);
    }, [])
    return (
        <Liste list={list}/>
    )
}
export default AlisverisListesi;
```

Liste komponenti:

```
const Liste = (props) => {
    return (
        <ul>
            {props.list.map((l, i) => (
                <li key={i}>{l}</li>
            ))}
        </ul>
    )
}
export default Liste;
```

Görüldüğü üzere AlisverisListesi komponenti ile Liste komponenti arasında bir parent-child ilişkisi bulunmakta, AlisverisListesi komponenti Liste komponentini içinde barındırmaktadır. Bu durumda üst componentte (AlisverisListesi) tanımlanmış bir listeyi alt componente yukarıdaki gibi Liste isimli componente list={list} olarak geçmekteyiz. Bu sanki bir metoda parametre geçmeye benzer. Alt komponentten ise props.props_name söz dizimi ile buna erişilebiliyor.

Props Doğrulama

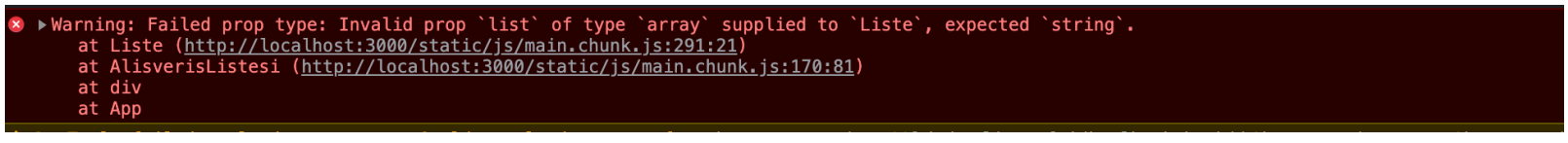
Genel olarak gelen propslar any tipinde görünür. Yani herhangi birşey olabilir. Ancak yazılımda belirsizlikler her zaman bizi zor durumlara sürükler. Bu yüzden tavsiye edilen her zaman doğrulamanın yapılmasıdır. Gelen bir prop'sun doğrulanması propType'larla yapılır. Şimdi propType tiplerine hızlıca bakalım.

Type	Validator
Arrays	`React.PropTypes.array`
Boolean	`React.PropTypes.bool`
Functions	`React.PropTypes.func`
Numbers	`React.PropTypes.number`
Objects	`React.PropTypes.object`
Strings	`React.PropTypes.string`

List componentimizi aşağıdaki gibi düzenleyelim.

```
import PropTypes from 'prop-types';
const Liste = (props) => {
    return (
        <ul>
            {props.list.map((l, i) => (
                <li key={i}>{l}</li>
            ))}
        </ul>
    )
}
Liste.propTypes = {
    list:PropTypes.string
}
export default Liste;
```

Props içinde gelen list'in tipinin string olduğunu belirtiyoruz. Ancak kendisi bir array. Bu yüzden console'da bir hata ile karşılaşıyoruz.



Console'da da söylediği gibi string bekleniyordu ancak bir array geldiği söyleniyor. Bu tür doğrulamalar her zaman işlerimizi daha düzenli yapmamızı sağlar. Bir nevi test işlevi de üstlenir (Beklenen değer ile gelen değerin karşılaştırılmasından dolayı).

```
import { useState, useEffect } from "react";
import Liste from "./Liste";

const AlisverisListesi = (props)=>{
    const [list, setList] = useState([]);
    useEffect(() => {
        setList(["1kg Patates", "1kg Domates", "1kg Kabak"]);
    }, [])
    return (
        <Liste list={list} title={15}/>
    )
}
export default AlisverisListesi;
```

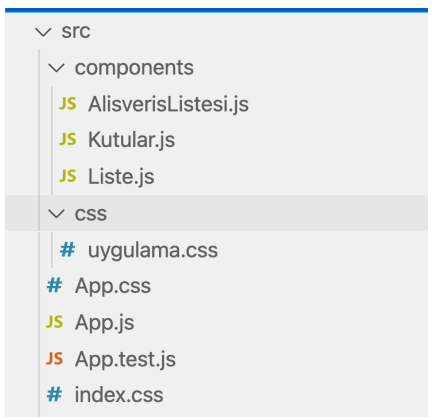
```
import PropTypes from 'prop-types';
const Liste = (props) => {
    return (
        <>
            {props.title} - {props.count}
            <ul>
                {props.list.map((l, i) => (
                    <li key={i}>{l}</li>
                ))}
            </ul>
        </>

    )
}
Liste.propTypes = {
    count:PropTypes.number.isRequired,
    list:PropTypes.array.isRequired,
    title: (props, propName) =>
        (typeof props[propName] !== 'string') ?
            new Error('title string olmalı') :
            (props[propName].length > 20) ?
                new Error('title 20 karakterden fazla olamaz') : null
}
export default Liste;
```

İki komponentte yaptığımız birkaç değişiklikle title adında yeni bir prop eklemiş olduk. Title a string beklerken bir Number geçtik. Kendi özel propTypes'ımızı da üstteki şekilde ayarlayabiliyoruz. Ayrıca tip belirttikten sonra da isRequired anahtar kelimesi ile de o prop'un olması gerektiğini belirtebiliyoruz. Böylece propertilerdeki doğrulama işlemleri bu şekilde yapılır.

Stil Dosyalarının Eklenmesi

Her web uygulaması gibi react'in de stil dosyaları ile güzelleştirilmesi gerekir. Bunun için istenilen sayfada import ifadesi ile bir stil dosyası (css) çağrılabilir.

Stil dosyamızı src altında css klasörüne ekliyoruz. İçinde ise şu basit css kodları yer almakta.




```css
.kutu{
    background-color:red;
    border:1px solid blue;
    width:100px;
    height: 100px;
    float:left;
}
```

Kutular adında yeni bir komponent oluşturuyoruz. App.js içerisinde diğer komponentimizin (AlisverisListesi) yerine ya da altına bunu iliştiriyoruz.

```jsx
import "../css/uygulama.css";
const Kutular = (props)=>{
    return(
        <div className={"kutu"}>1. Kutu</div>
    )
}
export default Kutular;
```

Kodlardan da görüleceği gibi src klasörü altına css adında oluşturduğumuz klasörde yer alan bu css dosyasını rahatlıkla component içinden çağırabiliyoruz. Dikkat etmemiz gereken; public klasöründe yer alan stillere sadece index.html tarafından erişebiliyor olmamız (Belki tüm uygulama genelinde kullanmak istediğiniz stilleri burada kullanabilirsiniz). Ayrıca CDN'den çekmemiz gereken stil dosyalarını index.html içinden çağırmamız gerekmekte.

Bir başka örnek de stil dosyasının bir modülmüş gibi kullanımıdır.

Css klasörü altında mystyles.module.css dosyasına css kodlarımızı yazıyoruz.

```css
.bigblue {
    color: red;
    padding: 40px;
    font-family: Arial;
    text-align: center;
  }
  .txbAd{
    color:blue;
```

```
    font-size:24px;
    font-family:Arial, Helvetica, sans-serif;
    text-align: center;
    border:1px solid green;
  }
```

Komponentimiz ise

```
import styles from '../css/mystyles.module.css';
const StilKullanim = () => {
    return (
        <>
            <div className={`${styles.bigblue}`}>
                .bigBlue stilinin uygulanmış hali.
            </div>
            <input type="text" className={`${styles.bigblue} ${styles.txbAd}`}/>
        </>
    )
}
export default StilKullanim;
```

Görüldüğü üzere styles adını farklı yerlerde kullanabiliyoruz. Birden fazla class adı kullanmak için string literal kullandığımıza dikkat edin.

Refs

Referanslar diğer adlarıyla refs, react komponentlerin içerisinde barındırdıkları Child elementlerle interaktif bir etkileşim kurmasını sağlar. Kısaca bir elemente referansta bulunmaya yarar.

```
import {useRef} from 'react'
const ChangeText = (props) => {
    const txbAd = useRef(null);
    function clickButton(){
        txbAd.current.value="3223";
    }
    return (
        <div>
            <input type="text" ref={txbAd}/>
            <button type="button" onClick={clickButton}>Tıkla</button>
        </div>

    )
}
export default ChangeText;
```

Göründüğü gibi input'a bir ref değeri ataması yapıldı. Bu ref değeri o elementi temsil (refere) ediyor. Böylece onun herhangi bir değerini bu referanstan değiştirebilir, onun bir değerine erişebiliriz. Çok tavsiye edilen bir yöntem değildir. Bunların yerine daha çok state'ler sık bir şekilde kullanılırlar.

State'leri de içerisinde barındıran bir form örneği aşağıda yer almaktadır.

Kullanıcının bilgilerini istediğimiz bir form için bu bilgileri topladığımız state aşağıdaki şekilde tanımlanıyor.

```
import { useState } from "react";
const RegisterForm = (props) => {
    const [userInfo, setUserInfo] = useState({
        Ad: "",
        Soyad: "",
        Email: ""
    })
    function textChangedHandler(element) {
        console.log(element.target.name);
        setUserInfo({ ...userInfo, [element.target.name]: element.target.value });
    }
    return (
        <form className="container col-md-5 mt-3">
            <div className="row">
                {userInfo.Ad && (<>Girilen Bilgiler : <h5>{userInfo.Ad} {userInfo.Soyad}
{userInfo.Email}</h5></>)}
            </div>
            <div className="row">
                <label htmlFor={"txbAd"} className="col-form-label col-1">Ad</label>
                <div className="col-11">
                    <input type="text" name="Ad" className="form-control" id="txbAd"
onChange={(e) => textChangedHandler(e)} />
                </div>
            </div>
            <div className="row">
                <label htmlFor={"txbSoyad"} className="col-form-label col-1">Soyad</label>
                <div className="col-11">
                    <input type="text" name="Soyad" className="form-control" id="txbSoyad"
onChange={(e) => textChangedHandler(e)} />
                </div>
            </div>
            <div className="row">
                <label htmlFor={"txbEmail"} className="col-form-label col-1">Email</label>
                <div className="col-11">
                    <input type="email" name="Email" className="form-control" id="txbEmail"
onChange={(e) => textChangedHandler(e)} />
                </div>
            </div>
        </form>
    )
}
```

```
export default RegisterForm;
```

Kısaca açıklamak gerekirse; her bir input'un aslında aynı Change event'i bulunmakta. Bu event change olmuş elementin özelliklerini textChangedHandler isimli event'e aktarıyor. Bu aktarma işlemini element.target ile yapıyor. Daha sonra bu elementin özellikleri (name ve value değerleri) bir state'e aktarılıyor. State'in değeri her input değişikliğinde değiştiriliyor.

NAVIGATION

Bir react web uygulamasının asıl amacı single page web application (SPA) olsa da, bir çok komplike web uygulamasında da kullanılması için navigation özelliklerini içinde barındırır.

Her web teknolojisinde olduğu gibi buradaki ismi de routing olan bu yöntemler için öncellikle react-router-dom paketinin kurulması gerekecektir.

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

index.js içindeki şu güncellemeyi unutmuyoruz. Tüm elemenlerimizin BrowserRouter komponenti içinde sarmalanmış olduğunu belirtiyoruz.

```
ReactDOM.render(
  <BrowserRouter>
    <App />
  </BrowserRouter>,
  document.getElementById('root')
);
```

Daha sonra Sayfalarımızı basitçe şu şekilde komponentler halinde yazalım.

NotFound componenti olmayan url'leri bize bildirecek.

```
import React from 'react'

export default function NotFound(props) {
    return (
        <div>
            Ooops {props.location.pathname} bulunamadı.
        </div>
    )
}
```

Linklerimizi üzerinde barındıran bir Anasayfa

```
import React from 'react'
import { Link } from 'react-router-dom';

export default function AnaSayfa() {
    return (
        <div>
            <a href="/" style={{marginLeft:'20px'}}>Anasayfa</a>
```

```
            <a href="/Iletisim" style={{marginLeft:'20px'}}>İletişim</a>
            <Link to="Hakkimizda" style={{marginLeft:'20px'}}>Hakkımızda</Link>
        </div>
    )
}
```

Basit anlamda bir Hakkımızda sayfası.

```
import React from 'react'

export default function Hakkimizda() {
    return (
        <div>
            Hakkımızda
        </div>
    )
}
import React from 'react'

export default function Iletisim() {
    return (
        <div>
            İletişim
        </div>
    )
}
```

Route yapılandırmamızı yaptığımız App.js'mizi de şu şekilde düzenliyoruz.

```
import React, { Suspense } from 'react';
import { Route, Router } from 'react-router';
import NotFound from './components/NotFound';

const Anasayfa = React.lazy(() => import('./components/HomePages/AnaSayfa'))
const Hakkimizda = React.lazy(() => import('./components/HomePages/Hakkimizda'))
const Iletisim = React.lazy(() => import('./components/HomePages/Iletisim'))
function App() {
  return (
      <div className="App">
        <Suspense fallback={<div>Yükleniyor...</div>}>
          <Route extract strict path="/" component={Anasayfa} />
          <Route exact path="/Hakkimizda" component={Hakkimizda}/>
          <Route exact path="/Iletisim" component={Iletisim}/>
        </Suspense>
      </div>

  );
}

export default App;
```

Lazy ifadesi ihtiyaç halinde yüklenmesini sağlamak için kullanılır. Yazılım teknolojilerinde sık kullanılan lazy loading terminolojisi ile çalışır. Suspense ifadesi bir container görevi görür ve içerisindekiler yüklenirken fallback içindeki ifadeyi bize gösterir.

Bir örneğimiz de olmayan sayfalar için hatanın gösterilmesi olabilir.

```
import React from 'react'
import { Link } from 'react-router-dom'

export default function Navigation() {
    return (
        <div>
            <a href="/" style={{marginLeft:'20px'}}>Anasayfa</a>
            <a href="/Iletisim" style={{marginLeft:'20px'}}>İletişim</a>
            <Link to="Hakkimizda" style={{marginLeft:'20px'}}>Hakkımızda</Link>
        </div>
    )
}
```

```
import React from 'react';
import { Route, Switch } from 'react-router';
import Iletisim from './components/HomePages/Iletisim';
import Navigation from './components/Navigation';
import NotFound from './components/NotFound';
import AnaSayfa from './components/HomePages/AnaSayfa';
function App() {
  return (
    <div className="App">
      <Navigation/>
      <Switch>
        <Route exact path="/" component={AnaSayfa}/>
        <Route path="/Iletisim" component={Iletisim}/>
        <Route component={NotFound}/>
      </Switch>
    </div>

  );
}

export default App;
```

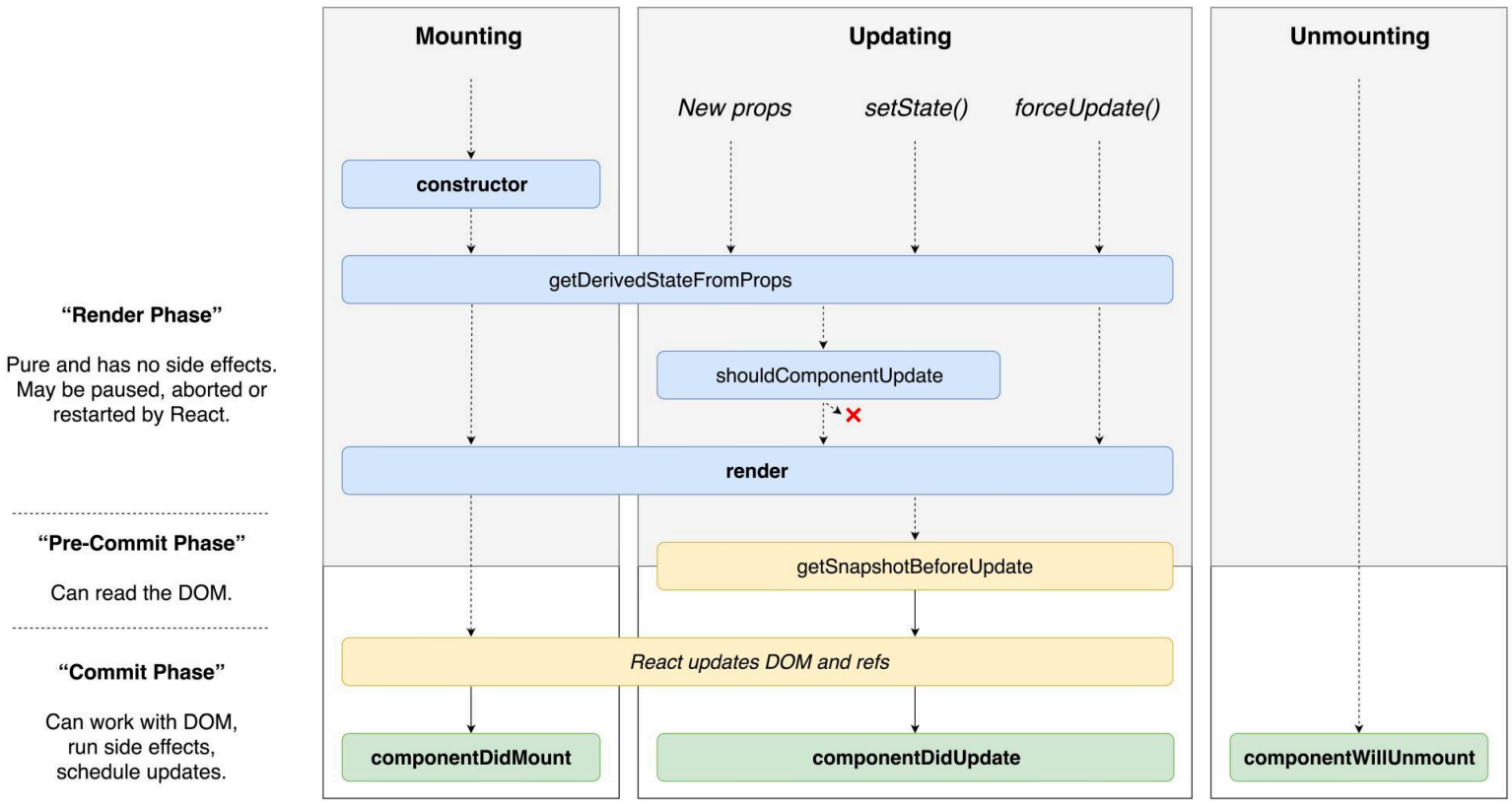
Switch içerisinde yönlendirme sayfalarımızı görüyoruz. Burada Route olarak belirttiklerimiz path'lere karşılık gelen sayfaları göstermekte. Switch'in görevi bir if else yapısı gibi çalışmak ve istekte bulunan sayfayı yukarıdan aşağıya kontrol etmekte. Birisi / Hakkimizda sayfasına gitmek isterse, öncelikle ilk satır kontrol edilecek buna uymadığı için (uymaması için exact kullandık, exact url'in tam eşleşmesini ister) bir aşağıdaki /Iletisim url'ine uyup uymadığı kontrol eder. Bu da uymadğı için de en sondakini çalıştırır (sondakinin herhangi bir path'inin olmadığına dikkat edin). Bu durumda son yazılan (path ve exacth olmadığı için) else yapısı gibi iş yapacak ve bu sayfa yok diyecektir. /Iletisim/sdfsdf gibi

bireyler de yazdığımızda url'in çalıştığını görmekteyiz. Buna engel olmak için path="/Iletisim" olan Route a exact ekleyebiliriz. (Exact = Birebir aynı, strict= sıkı).

LIFECYCLE

Bir react uygulaması index.js içinde başlar ve ardından App.js root component'i ile devam eder. Buraya kadar herşey temiz bir şekilde ilerlemekteyken, anlaşılması kolay iken sonraki aşamada, komponent kısmına geldiğimizde farklı yaşam döngüleri ile karşılaşmaktayız. Bu yüzden komponent düzeyinde yaşam döngüsü hayati önem taşımaktadır.

Bir class component içerisinde lifecycle event'ler 3 farklı bölümden oluştur. Bunlardan



ilki mounting, ikincisi updating, üçüncüsü unmounting. Yukarıdaki resimde de detaylarını gördüğümüz yaşam döngüsü metotlarından bazıları güncel versiyondan adı değiştirildi (UNSAFE isimleri getirildi).

```
import React, { Component } from 'react'

export default class MemberList extends Component {
    constructor() {
        console.log("Constructor çalıştı")
        super()
        this.state = {
            members: [],
            loading: false,
            error: null
        }
    }
    UNSAFE_componentWillMount() {
        console.log("UNSAFE_componentWillMount")
        this.setState({ loading: true });
```

```
        getFakeMembers(20).then(
            members => {
                this.setState({ members, loading: false })
            },
            error => {
                this.setState({ error, loading: false })
            }
        )
    }
    UNSAFE_componentWillUpdate() {
        console.log('UNSAFE_componentWillUpdate')
    }
    render() {
        const { members, loading, error } = this.state
        return (
            <div className="member-list">
                {(loading) ? <span>Loading Members</span> :
                    (members.results.length) ? members.results.map((user, i) => <Member
key={i} {...user} />) : <span>0 members loaded...</span>
                }
                {(error) ? <p>Error Loading Members: error</p> : ""}

            </div>
        )
    }
}

const Member = ({ email, picture, name, location }) =>
    <div className="member">
        <img src={picture.thumbnail} alt="" />
        <h1>{name.first} {name.last}</h1>
        <p><a href={"mailto:" + email}>{email}</a></p>
        <p>{location.city}, {location.state}</p>
    </div>

const getFakeMembers = count => new Promise((resolves, rejects) => {
    const api = `https://api.randomuser.me/?nat=US&results=${count}`;
    fetch(api).then(res => resolves(res.json()), error => rejects(error));
})
```

Örnekte de göreceğimiz gibi constructor önce çalışıyor ardından da diğerleri sırası ile çalışıyor (UNSAFE_componentWillMount, UNSAFE_componentWillUpdate). WillMount bir componentin mount edilmesi sırasında, WillUpdate ise komponent güncellendiğinde çalışacaktır.

Promise sınıflar, kendisine verilen iki function parametreden birincisi, then zincirinde başarılı sonucu, ikinci parametre ise başarısız olması durumunda gerçekleşir.

Functional componentlerde ise bu işlem useEffect'ler ile sağlanmakta.

```
import React,{useEffect} from 'react'

export default function MemerList() {
    useEffect(() => {
        console.log("Komponent yüklendi")
    }, [])
    return (
        <div>
            MemerList
        </div>
    )
}
```

Görüldüğü gibi useEffect bu işte kolay bir şekilde kullanılıyor. En sondaki [] ifadesi içerisinde içeriği güncellenecek ifade yazılabilir. Bu durumda o ifade güncellenir güncellenmez useEffect çalışacaktır. Bu kullanımı ile useEffectler çok kullanışlıdır. Örneğin, props'ların değişmesi durumunda bile [props] şeklinde yazılıp, propsların değişmesinde kodların çalışması sağlanacaktır.

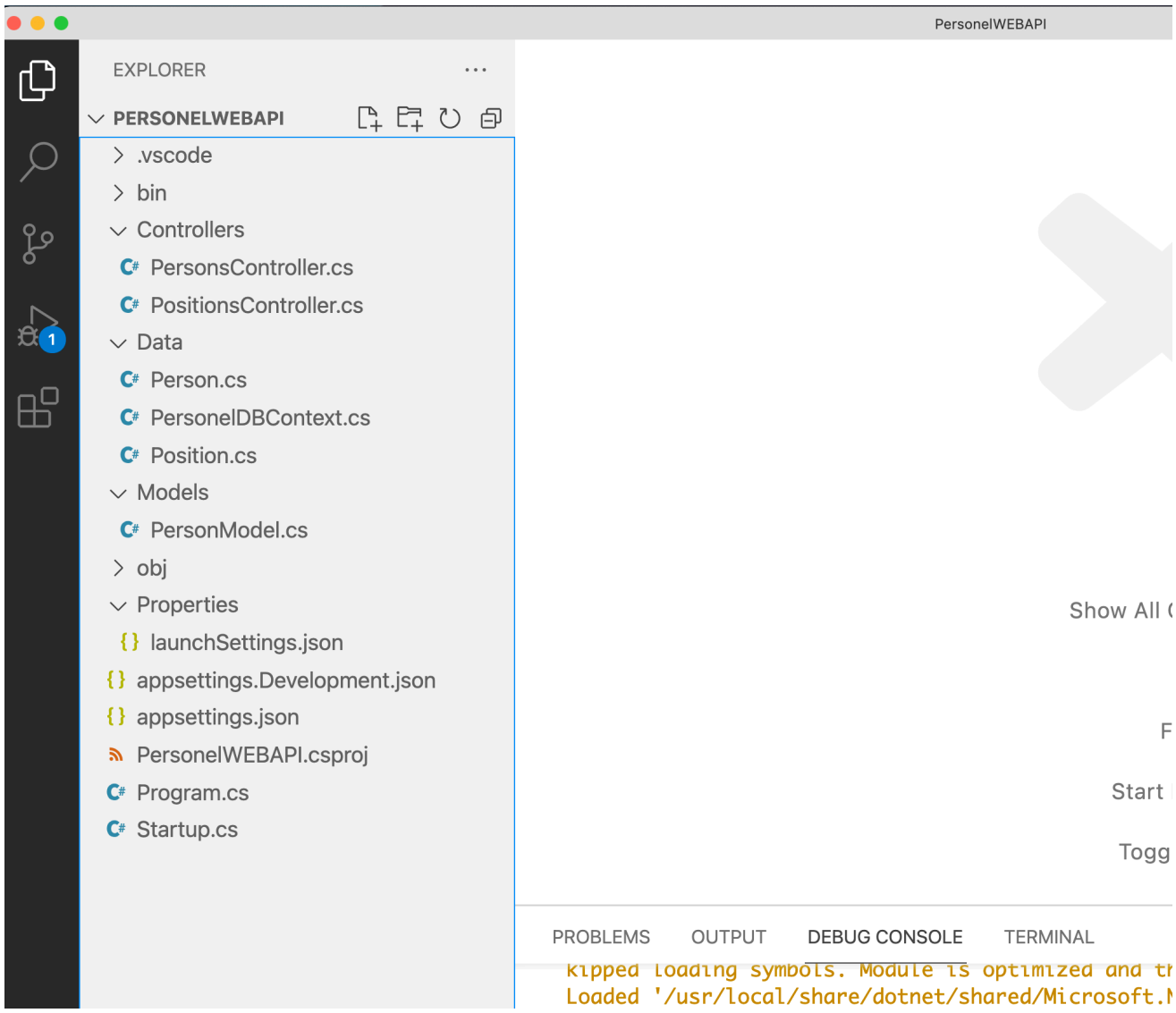
KULLANIM ÖRNEĞİ:

Bilgilerimizi tazelemek, pekiştirmek ve biraz da yeni bilgiler eklemek adına bir örnekle devam ediyoruz.

Örneğiniz bir personel takip sistemi üzerine olacak. .NET 5 ile küçük bir web API mimarisi kuracağız (InMemoryDB).

Personelleri listeleyebildiğimiz bir sayfamız (komponent) olacak ve bu liste içerisinde bir personel arayabileceğiz. Yeni personel kaydedebildiğimiz bir sayfa ve mevcut kullanıcı güncelleyebildiğimiz de bir sayfamız olacak.

Öncelikle .NET 5 Web API mimarisinden başlayalım. Bunun için dotnet 'in kurulu ve console'dan çalışıyor olması lazım. dotnet —version komutu ile versiyon kontrol edilebilir. Olmaması durumunda .NET 5 SDK indirilip yüklenmelidir. (Yükleme sonrası bilgisayar ve/veya tarayıcı yeniden başlatılmalı). dotnet new webapi komutunu çalıştırarak yeni bir webapi projesi oluşturuyoruz.



Projemizin dosya yapısı yukarıda görüldüğü gibi. Controllers, Data(Veritabanı context'imizi barındıracak) ve Models.

PersonsController sınıfımız aşağıdaki gibi.

```
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using PERSONELWEBAPI.Data;

namespace PERSONELWEBAPI.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class PersonsController : Controller
    {
        private readonly PersonelDBContext _context;
        public PersonsController(PersonelDBContext context)
        {
            _context = context;
        }
        [HttpGet]
        public IActionResult Get() => Json(_context.Persons.Take(100).ToList());
        [HttpGet("{id}")]
        public IActionResult Get(int id) => Json(_context.Persons.FirstOrDefault(p => p.Id
== id));
        [HttpPost]
        public IActionResult Post(Person person)
```

```
        {
            if (person == null)
                return NoContent();
            _context.Persons.Add(person);
            _context.SaveChanges();
            return Ok();
        }
        [HttpGet]
        [Route("SearchPerson/{searchTerm}")]
        public IActionResult SearchPerson(string searchTerm)
        {
            if (string.IsNullOrEmpty(searchTerm))
            {
                return Json(_context.Persons.Include(p => p.Position).Take(100).ToList());
            }
            var searchResult = _context.Persons.Where(p => p.FirstName.Contains(searchTerm)
|| p.LastName.Contains(searchTerm)).ToList();
            return Json(searchResult);
        }
    }
}
```

PositionsController sınıfımız aşağıdaki gibi.

```
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using PERSONELWEBAPI.Data;

namespace PERSONELWEBAPI
{
    [Route("api/[controller]")]
    [ApiController]
    public class PositionsController : Controller
    {
        private readonly PersonelDBContext _context;
        public PositionsController(PersonelDBContext context)
        {
            _context = context;
        }
        [HttpGet]
        public IActionResult Get() => Json(_context.Positions.ToList());
        [HttpPost]
        public IActionResult Post(Position position){
            if(position==null) return NoContent();

            _context.Positions.Add(position);
            _context.SaveChanges();
            return Ok();
        }
    }
}
```

Veritabanımız için gerekli olan Person ve Position tiplerimiz ve Context'imiz aşağıdaki gibi.

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace PERSONELWEBAPI.Data
{
    public class Person
    {
        [Key]
        public int Id { get; set; }
        [Required]
        public string FirstName { get; set; }
        [Required]
        public string LastName { get; set; }
        public int PositionId { get; set; }
        public Position Position { get; set; }
    }
}

using System.ComponentModel.DataAnnotations;

namespace PERSONELWEBAPI.Data
{
    public class Position
    {
        [Key]
        public int Id { get; set; }

        public string Name { get; set; }
        public decimal MaximumSalary { get; set; }
    }
}

using Microsoft.EntityFrameworkCore;

namespace PERSONELWEBAPI.Data
{
    public class PersonelDBContext : DbContext
    {
        public PersonelDBContext(DbContextOptions<PersonelDBContext> options) :
base(options)
        {
        }
        public DbSet<Person> Persons { get; set; }
        public DbSet<Position> Positions { get; set; }

    }
}
```

Uygulamamızın MiddleWare'ini de şu şekilde yapılandırıyoruz.

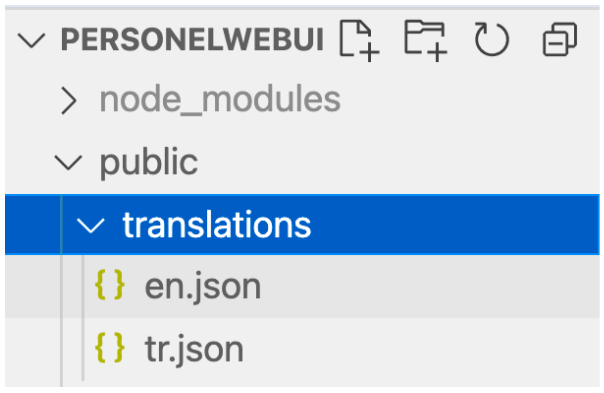
```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.OpenApi.Models;
using PERSONELWEBAPI.Data;

namespace PersonelWEBAPI
{
    public class Startup
    {
        readonly string originsConfiguration = "originsConfiguration";
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        // This method gets called by the runtime. Use this method to add services to the
container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddCors(options =>{ options.AddPolicy(name: originsConfiguration,
                builder =>
                {
                    builder.AllowAnyOrigin().AllowAnyHeader().AllowAnyMethod();
                });
            });

            services.AddControllers();
            services.AddSwaggerGen(c =>
            {
                c.SwaggerDoc("v1", new OpenApiInfo { Title = "PersonelWEBAPI", Version =
"v1" });
            });

            services.AddDbContext<PersonelDBContext>(option =>
            {
                option.UseInMemoryDatabase(databaseName: "PersonelDB");
            });
        }

        // This method gets called by the runtime. Use this method to configure the HTTP
request pipeline.
        public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
```

```
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
                app.UseSwagger();
                app.UseSwaggerUI(c => c.SwaggerEndpoint("/swagger/v1/swagger.json",
"PersonelWEBAPI v1"));
            }

            app.UseHttpsRedirection();
            app.UseCors(originsConfiguration);
            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllers();
            });
        }
    }
}
```

Şimdi ise, önyüzde geliştireceğimiz kısımları oluşturalım. npx create-react-app . ile yeni bir uygulama oluşturuyorum (Uygun bir yerde klasör açıp, vs code ile bu proje dizinine konuşlandıktan sonra).

Girişte tüm personelleri listeleyeceğiz. Ardından herhangi biri üzerinde düzenleme yapabileceğiz. Yeni bir personel oluşturabilmeliyiz. npx create-react-app . ile ui tarafını oluşturuyoruz. Ardından npm install react-router-dom —save paketini ekliyoruz. Ayrıca api istekleri için npm install axios —save paketini de kuruyoruz.

Çoklu dil paketi için ise npm i i18next i18next-xhr-backend react-i18next -S paketini kullanıyoruz. Dil dosyalarını public/translations klasöründe barındıracağız. Her dil için bir json dosyası ekleyeceğiz ve dile ait kelimelerin karşılıklarını (ya da cümle de olabilir) json olarak tutacağız.



Resimde de göründüğü üzere translations altında İngilizce için en.json, bunlara karşılık Türkçe için tr.json dosyası oluşturuldu.

İçerisinde ise her kelimeye karşılık key value çiftleri yer almakta.

{} tr.json

public > translations > {} tr.json > ...

```
{
    "yenipersonel.firstname":"Ad",
    "yenipersonel.lastname":"Soyad"
}
```

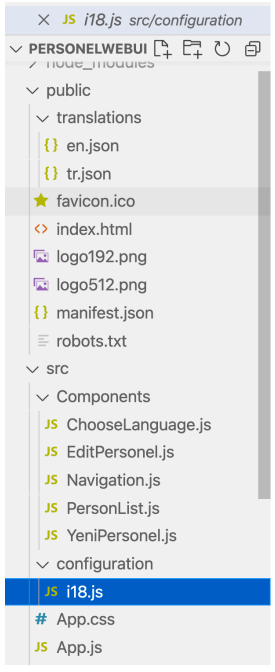
i18n yapılandırması için gerekli dosyayı configuration klasörü altında i18.js dosyası olarak oluşturuyoruz. Dosya içerisinde ise aşağıdaki resimdekiler mevcut.

JS *i18.js*

src > configuration > JS i18.js > ...

```
import i18n from 'i18next'
import Backend from 'i18next-xhr-backend'
import { initReactI18next } from 'react-i18next'

i18n
    .use(Backend)
    .use(initReactI18next)
    .init({
        lng: 'tr',
        backend: {
            loadPath: '/translations/{{lng}}.json'
        },
        (loading...) (property) interpolation: {
            escapeValue: boolean;
            formatSeparator: string;
        }
        interpolation: {
            escapeValue: false,
            formatSeparator: ','
        },

    });
    export default i18n;
```

Dosya yapımız da aşağıdaki resimdeki gibi olacak.



Dil seçimini sağlayan ChooseLanguage.js componenti aşağıdaki gibi olacaktır.

```
import React from 'react'
import { useTranslation } from 'react-i18next'

const ChooseLanguage = () => {
    const { t, i18n } = useTranslation()

    const changeLanguage = (event) => {
        i18n.changeLanguage(event.target.value)
    }

    return (
        <div className="d-flex justify-content-end">
            Change Language
            <select onChange={changeLanguage} defaultValue={"tr"}>
                <option value="en">English</option>
                <option value="tr">Turkish</option>
            </select>
        </div>

    )
}

export default ChooseLanguage
```

Kodların içeriği şişirmemesi adına kodların tamamını github'da bulunduruyorum. Link: https://github.com/matanist/personelwebui . Kodlara bu adresten erişebilirsiniz.

REDUX AND REDUX DEV TOOLS

Herşeyi modülleştirme çalışırken her şeyi komponent içinden çağırmak ne kadar mantıklıdır? Bir daha kullanacak bir kodu A komponent'i içinde bu kodları yazmak modülerim demekle ancak modüler davranmamakla aynı şeyi kasteder. Örneğin yukarıda fake kullanıcılar çeken fetch kodlarını tekrar kullanmak istersem o kodları tekrar kopyalayıp yazmam gerekecektir.

State ve props yapılarını da görünce hep birşey dikkatimizi çekmelidir. Veri aktarımı hep tek yönlü gerçekleşmekte, üst komponentten alt komponente aktarılmakta (propslar ile). Bu çok önemli hız artırırımı sağlamakla birlikte bazı durumlarda kötü etki yaratmaktadır. Örneğin, bir yerde yapılan güncelleme tüm componentleri etkilemesi gereken durumlarda bizler genellikle bunu ya cookie'ye ya da localStorage'lere yazma durumunda kalıyoruz. Ancak tarayıcının bunu desteklememesi ya da herhangi bir engelinin olması durumunda bu verileri saklamakta zorluklar çekeceğiz. Ayrıca verinin yoğun olduğu durumlarda ise tarayıcı sınırlarından dolayı bu işlemde problemler yaşanmakta, hız problemleri ile karşı karşıya

kalınmaktadır. İşte bu problemlerin çözümü için react redux yapısı kullanılmaktadır. Redux merkezi bir store oluşturarak uygulama genelinden erişilmesi gereken verileri bize sunuyor. Merkezi olarak oluşturduğunuz bir veri mağazasında (store) istediğiniz verileri saklayarak daha sonra kullanabiliyorsunuz. Redux yapısında da Action ve Reducer'lardan yararlanılır. Action'lar merkezi yerden kontrol edilen bir tek immutable objelerden oluşmaktadır. Reducer'lar ise bu objelerin işlemlere göre taşınmasını sağlar. Reducer'lar indiktör (kötü etkiyi azaltacak) görevi görürler. Kısacası Redux bize merkezi bir state kaynağı sağlar.

React redux'dan faydalanabilmek için şu paketlerin kurulması gerekir.

```
npm install react-redux
npm install redux
```