Kadir ÇAPKIN 21360859023

NodeJS ile Web programlama Raporu Hafta – 7-8

Weather app – 3

---

Bu haftaki dersimize Object Property Shorthand ve Object Destructuring adlı konularla başladık. Öncelikle WeatherApp klasörümüz içerisinde object-test.js adlı bir dosya oluşturduk. Object Property Shorthand kavramı şu şekildedir bir nesne oluştururken nesne içerisindeki değişkenleri nesne dışarısındaki değişkenlerden doğrudan kullanacaksak doğrudan kullanmak yerine kısa bir yol ile kullanabiliyoruz.

```
const userName = "Kadir";
const userAge = 25;

const user = {
    //userName: userName, Eğer property ve değişken adı aynıysa dogrudan
kullanılabiliriz. (shorthand)
    userName,
    age: userAge,
    location: 'Bursa'
}

console.log(user);
```

Burada normalde nesne oluştururken kullandığımız userName : userName yöntemi ile dışarıda bulunan nesneyi doğrudan (userName,) kullanacağımız yöntem aynıdır. Bu yöntemi kullanarak daha kolay bir şekilde tanımlayabiliyoruz. Çıktıyı incelersek bu yöntemin kolaylık sağlamak dışında bir farkı olmadığını söyleyebiliriz.

```
PS C:\Users\kadir\Desktop\Programlama\NodeJS\Okul\WeatherApp> node object-test.js
{ userName: 'Kadir', age: 25, location: 'Bursa' }
```

Bu konudan sonra Object Destructuring konusuna gelecek olursak: Bu konunun genel amacı oluşturulan bir nesnedeki değişkenler nesne dışındaki nesnelere dağıtmak diyebiliriz. Product adında bir nesne oluşturduğumuzu varsayalım ardından bu nesnesin içerisindeki property'leri başka değişkenlere tek bir satırda dağıtmak isteyelim. Normalde yapacağımız yöntem bir değişken oluşturup bu değişkene product.property1 şeklinde atama yapmamız gerekirdi.

```
const product = {
    label: "Laptop",
    price: 300,
    stock: 20,
    salePrice: undefined,
    testString: "TestString",
    testNumber: 13,
}
```

```
/*Standart
const label = product.label;
const stock = product.stock;*/
```

Bu şekilde kolay bir şekilde atama yapabiliriz ama görüldüğü gibi eğer birden fazla değişken atamak istersek alt alta daha fazla satır yazmamız gerekecek ve bu da kod okunurluğunu azaltacaktır. Bunun yerine Object Destructuring kullanırsak

```
const product = {
    label: "Laptop",
    price: 300,
    stock: 20,
    salePrice: undefined,
    testString: "TestString",
    testNumber: 13,
}

/*Standart
const label = product.label;
const stock = product.stock;*/

const{label,stock,rating} = product; // Object destructuring (Mevcut degiskeni
propertylerle eslestirir ve olusturur.)
console.log("label:",label);
console.log("stock:",stock);
console.log("rating:",rating); // product nesnesi icerisinde rating bilgisi
olmadigi icin eslesmedi.
```

Bu kodun çıktısına bakacak olursak hiçbir fark olmadığını görürüz.

```
PS C:\Users\kadir\Desktop\Programlama\NodeJS\Okul\WeatherApp> node object-test.js
label: Laptop
stock: 20
rating: undefined
```

Burada dikkat etmemiz gereken şey nesne değişkenin ismiyle oluşturacağımız yeni değişkenin isminin aynı olmasıdır. Şu an rating değişkenin bir değer almamasının sebebi product nesnesi içerisinde aynı isimde rating property bulunmamasıdır. Eğer biz aynı isimle bir değişken yerine farklı isimde bir değişken oluşturmak istersek ise yine referans olarak kullanmak istediğimiz nesne değişkeninin ismini şu şekilde vermemiz gerekir.

```
const{label,stock,rating,testString:test} = product; // Object destructuring
(Mevcut degiskeni propertylerle eslestirir ve olusturur.)
```

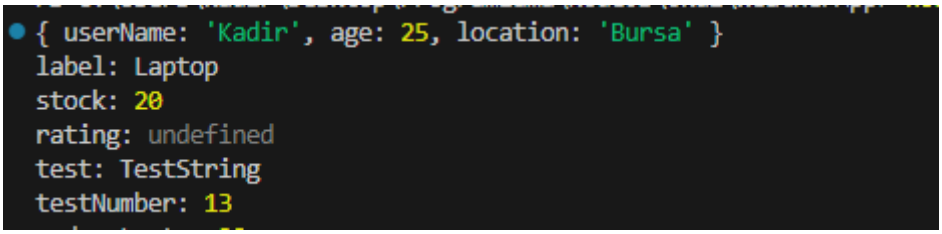
Şu an yaptığı şey söylediğim gibi product nesnesi içerisinde testString isimli değişkeni bulmak, farklı olarak test değişkeni içerisine atmaktır. Çalıştırmak istersek çıktımız:

```
{ userName: 'Kadir', age: 25, location: 'Bursa' }
label: Laptop
stock: 20
rating: undefined
test: TestString
```

Object destructuring yaparken eğer atama yaparsak sonuç değişmez ve yine nesne içerisinde bulunan değere atama yapmış olur.

```
const product = {
    label: "Laptop",
    price: 300,
    stock: 20,
    salePrice: undefined,
    testString: "TestString",
    testNumber: 13,
}

/*Standart
const label = product.label;
const stock = product.stock;*/

const{label,stock,rating,testString:test,testNumber=5} = product; // Object
destructuring (Mevcut degiskeni propertylerle eslestirir ve olusturur.)
console.log("label:",label);
console.log("stock:",stock);
console.log("rating:",rating); // product nesnesi icerisinde rating bilgisi
olmadigi icin eslesmedi.
console.log("test:",test); // Mevcut eslesmeyi yaptı ve degisken ismi
degistirdik.
console.log("testNumber:",testNumber); // Atama yapsakta nesnenin icindekini
aldi.
```

Bu kodun çıktısına bakacak olursak nesne içerisindeki değerlerin nesne dışındaki değişkenlere tek bir satırda aktarıldığını görebiliyoruz.

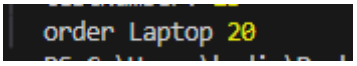



Object destructuring bir fonksiyon içerisinde de kullanabiliriz.

```
// Fonksiyonda kullanımı: istenilen parametreyi yollama
const transaction = (type,{label,stock}) => {
    //const {label} = myProduct;
    console.log(type,label,stock);
}

transaction('order',product);
```

Bu fonksiyonda olan şey iki parametre bir String ve nesne yollanıyor, fakat bu nesne içerisinden Object Destructuring yapılarak, fonksiyon {label,stock} değişkenleri içerisinde product nesnesinin(label,stock) değişkenleriyle eşleşiyor. Sonuç olarak üç farklı değişken aktarılmış oluyor diyebiliriz.

Çıktısına bakacak olursak değişkenlere değerlerin doğru bir şekilde aktarıldığını görebiliriz:




Şimdi bu fonksiyona atama örneğini daha önce yaptığımız geocode.js içerisinde kullanarak düzeltirsek

```
 request({ url: geocodeURL, json: true }, (error, response) => {
   if (error) {
     callback("Geocoding servisine bağlanamadı", undefined)
   } else if (response.body.features.length == 0) {
     callback("Belirttiğiniz konum bulunamadı", undefined)
   } else {
     const boylam = response.body.features[0].center[0] // boylam bilgisini verir
     const enlem = response.body.features[0].center[1] // enlem bilgisini verir
     const konum = response.body.features[0].place_name


     callback(undefined, {
       boylam: boylam,
       enlem: enlem,
       konum: konum
     })
```

Burada bulunan response değişkeni üzerinden ulaşmak yerine object destructuring ile doğrudan response.body nesnesi içerisine ulaşıp değişken olarak şu şekilde kullanabiliriz.

```
const geocode = (address, callback) => {
  // function body goes here
  const geocodeURL =
    "https://api.mapbox.com/geocoding/v5/mapbox.places/" + encodeURI(address)
+
".json?access_token=pk.eyJ1Ijoia2FkaXIzMjMzIiwiYSI6ImNsdTM5cno3NjA4NnAybW5kZTN
3dG9nbWMifQ._hU69OTmxDX6TQIiTd41-w"

  request({ url: geocodeURL, json: true }, (error, {body}) => {
    if (error) {
```

```
      callback("Geocoding servisine bağlanamadı", undefined)
    } else if (body.features.length == 0) {
      callback("Belirttiğiniz konum bulunamadı", undefined)
    } else {
      const boylam = body.features[0].center[0] // boylam bilgisini verir
      const enlem = body.features[0].center[1] // enlem bilgisini verir
      const konum = body.features[0].place_name

      callback(undefined, {
        boylam: boylam,
        enlem: enlem,
        konum: konum
      })
    }
  })
}
```

Bu yöntem kod okunurluğu arttırır.

Aynı şekilde app.js içerisinde de bu değişikliği yapabiliriz.

```
geocode(address,(errGeoAPI,data)=>{

  if(errGeoAPI){

    console.log("Hata bulundu:",errGeoAPI);

    return

  }


  weatherapi(data.enlem,data.boylam,(errWeatherAPI,data)=>{

    if(errWeatherAPI){

      console.log("Hata bulundu:",errWeatherAPI);

      return

    }

    console.log(data);
```

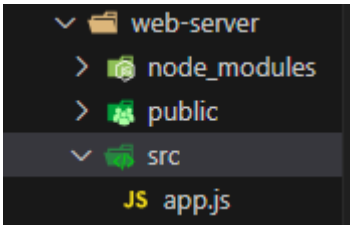
Görüldüğü gibi geocode fonksiyonun ilk parametresi bir değişken ikinci parametresi ise bir callback fonksiyon , bu callback fonksiyon ikinci parametresi olan data ise yukarıda görebileceğiniz gibi boylam,enlem ve konum bilgilerini tutan bir nesne şimdi bunun değişkenlerine tek tek data.değişkenAdi şeklinde yazmak yerine doğrudan {} parantezi açıp içerisine nesne değişkeni isimlerini yazarız ve object destructuring yaparak doğrudan kullanabiliriz.

```
geocode(address,(errGeoAPI,{enlem,boylam} = {})=>{ // Dogrudan nesne yerine istenilen property'leri yolladık. Eğer parametre hatalı girilirse err kosuluna girdiginde kod hata verir. Cünkü atama islemini yapmaya calisiyor. Bunun icin {enlem,boylam} = {} bu hale getirdik.
    if(errGeoAPI){
        console.log("Hata bulundu:",errGeoAPI);
        return
    }

    weatherapi(enlem,boylam,(errWeatherAPI,data)=>{ // datanın tamamamı yerine destructuring kullanabiliriz
        if(errWeatherAPI){
            console.log("Hata bulundu:",errWeatherAPI);
            return
        }
        console.log(data);
    })

})
```

Alttaki şekilde yollamamızın sebebi eğer geocode api'sinden bir veri dönmezse bu veriyi boş bir şekilde weatherapi'ye parametre olarak yollamaktır.

```
{enlem,boylam} = {}
```

Şimdi Nodejs web tarafında yapabileceğimiz işlemleri inceleyeceğiz. Bunun için öncelikle web-server adlı bir klasör oluşturalım. Ardından bu klasör içerisinde bir alt klasör olan src klasörünü oluşturalım ve içerisinde app.js adlı dosyayı oluşturalım. Görüntü şu şekilde olacak:




Şimdi bu açtığımız web-server klasörü içerisine gidip 2 modül yüklememiz gerekiyor birincisi Express ikincisi ise hbs modülleridir. Bunları şu şekilde yazıp yükleyebiliriz.

```
PS C:\Users\kadir\Desktop\Programlama\NodeJS\Okul\WeatherApp> cd web-server
PS C:\Users\kadir\Desktop\Programlama\NodeJS\Okul\WeatherApp\web-server> npm i express
```

Express , Node.js tabanlı bir web framework'tür. Bu framework genellikle web uygulamaları ve API'ler için kullanılır. http metodlarıyla birlikte yönlendirmeler yapılabilir.

Hbs modülü ise template engine'dir. Html içeriğimize dinamik verileri kullanmak için kullanabiliriz.

app.js dosyamızın içerisinde gerekli modülleri yükledikten sonra require ile modülü dahil etmemiz ardından bir değişken içerisine Express uygulamamızı başlatmamız gerekir.

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

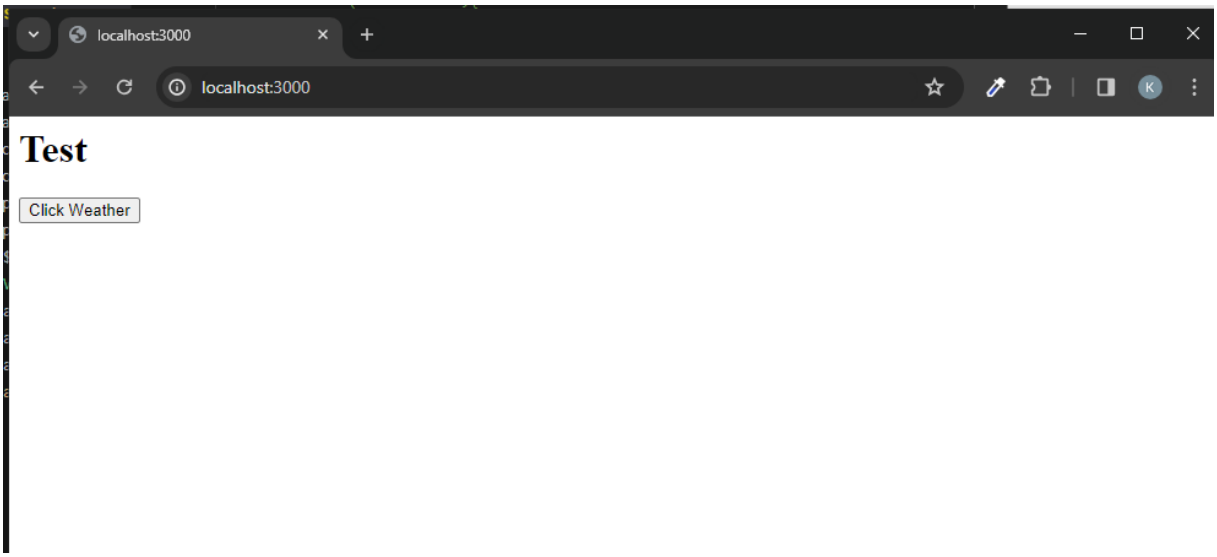
Şimdi bu app değişkeni üzerinden bize belirli metodlar gelir bu metodlar (get,post,put,delete) belirli routing işlemleri için kullanılır. Öncelikle get metoduna bakacak olursak ilk parametre olarak bir url uzantısı alır ikinci parametre olarak ise request ve response parametrelerini alan bir callback fonksiyon alır. Get metodu ile doğrudan kullanıcının göreceği html içeriklerini yollayabiliriz. Bunun yapmadan önce

Local host üzerinde çalıştığımız için app.listen(portNo,callback) kullanmamız gerekir.

Gireceğimiz port numarası 3000 olacak. Normalde web sunucuları default olarak 80 portu üzerinden bağlanır.

```
app.get('/',(req,res)=>{
    res.send(`<div>
            <h1>Test</h1>
            <a href="/weather"><button>Click Weather</button><a/>
            </div>`);
})
app.listen(3000,()=>{
    console.log("Listening on port 3000");
})
```

Şimdi app.js klasörünü çalıştırıp tarayıcıda localhost:3000 url'ine gidersek karşımıza çıkacak çıktı şudur:



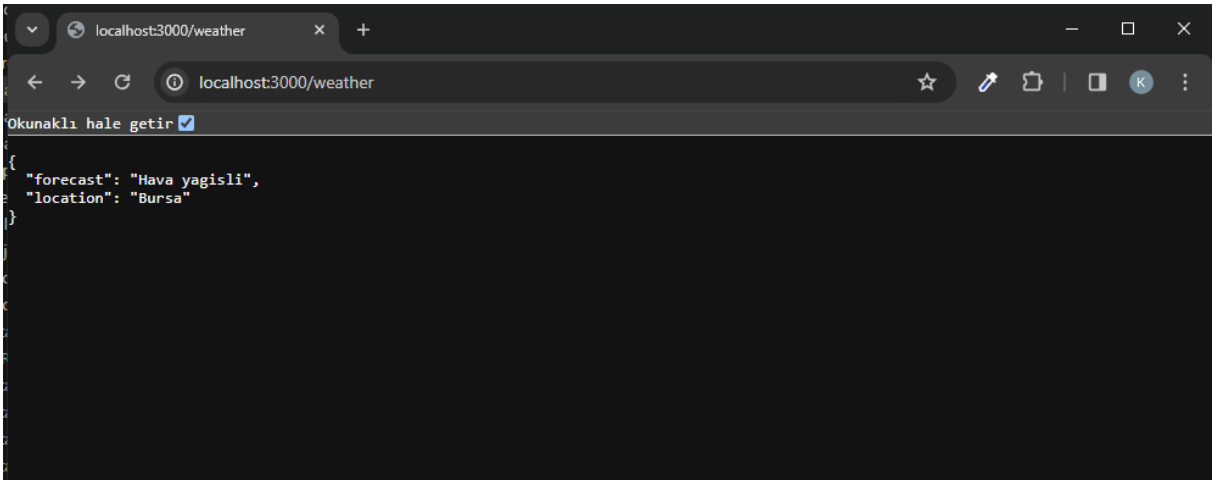

App.get metodu sayesinde doğrudan bir html içeriği yollamış olduk. Şimdi farklı url'lerinde get metodunu yazalım:

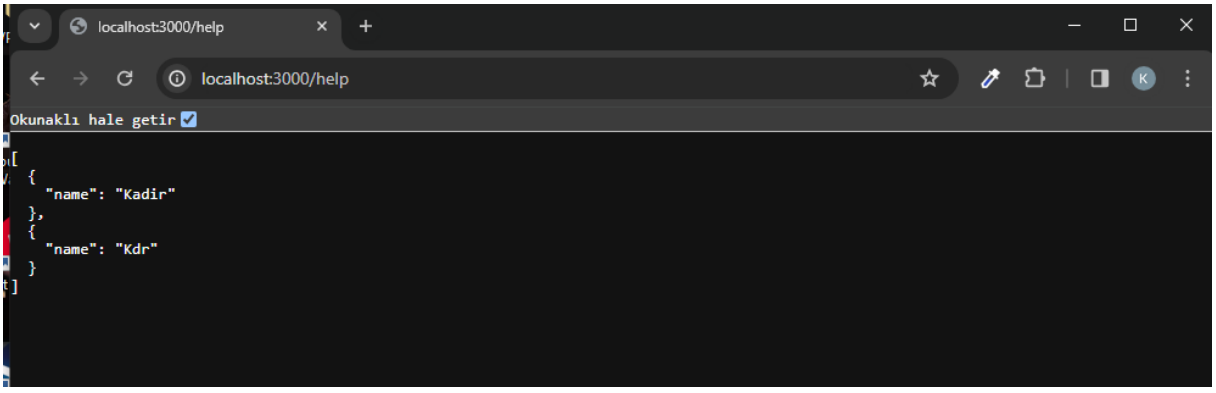
```
app.get('/weather',(req,res)=>{
    res.send({
        forecast: "Hava yagisli",
        location: "Bursa"
    });
});
```

Burada localhost:3000/weather yazdığımızda karşımıza bir nesne gönderilmiş olacak.

```
app.get('/help',(req,res)=>{
    res.send([{name:"Kadir"},
                {name:"Kdr"}]); //Json formatında yolladik
});
```

Aynı şekilde localhost:3000/help url girdiğimiz karşımıza bu sefer JSON formatında bir ekran gelecek. Çıktılarına bakacak olursak:



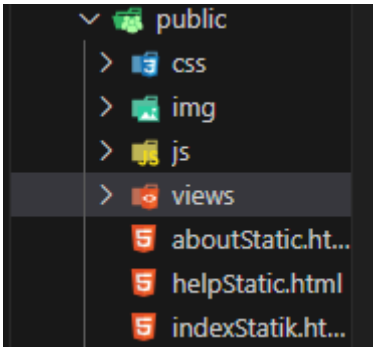




Basit bir şekilde içerik yollamayı gördük şimdi doğrudan bir html dosyası göndermek istersek yapacağımız şey şudur: oluşturduğumuz app değişkeni üzerinden yüklediğimiz hbs modülünü app.set('view engine','hbs'); diyerek dosyamıza dahil ediyoruz. Şimdi bizim statik dosyalarımız olabilir. (css,html,js,images) Bu dosyaları doğrudan

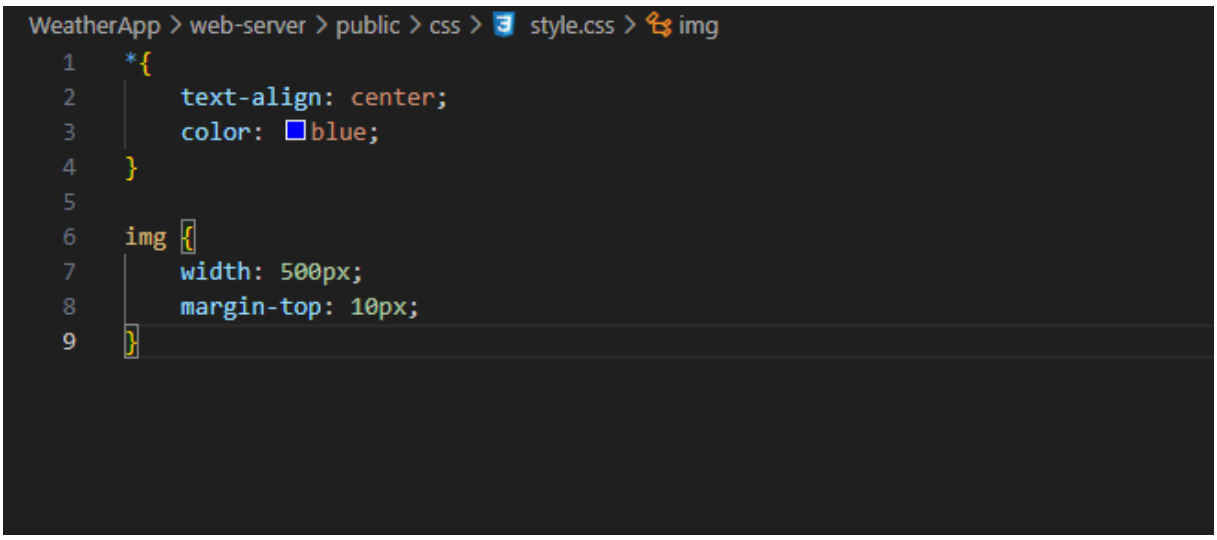
kullanamıyoruz. Bu dosyalara erişebilmek adına yapmamız gerekir iki satırlık bir kod var. Bu iki satır koddan önce path modülünü klasöre dahil ediyoruz ve şunları yazıyoruz.

```
const express = require('express');
const path = require('path');

const app = express();
app.set('view engine','hbs');

const publicDirectoryPath = path.join(__dirname,'../public'); // Statik web
sayfasını kullanmak
app.use(express.static(publicDirectoryPath)); // statik dosyalar
//app.com: app.com,app.com/help, app.com/about
```

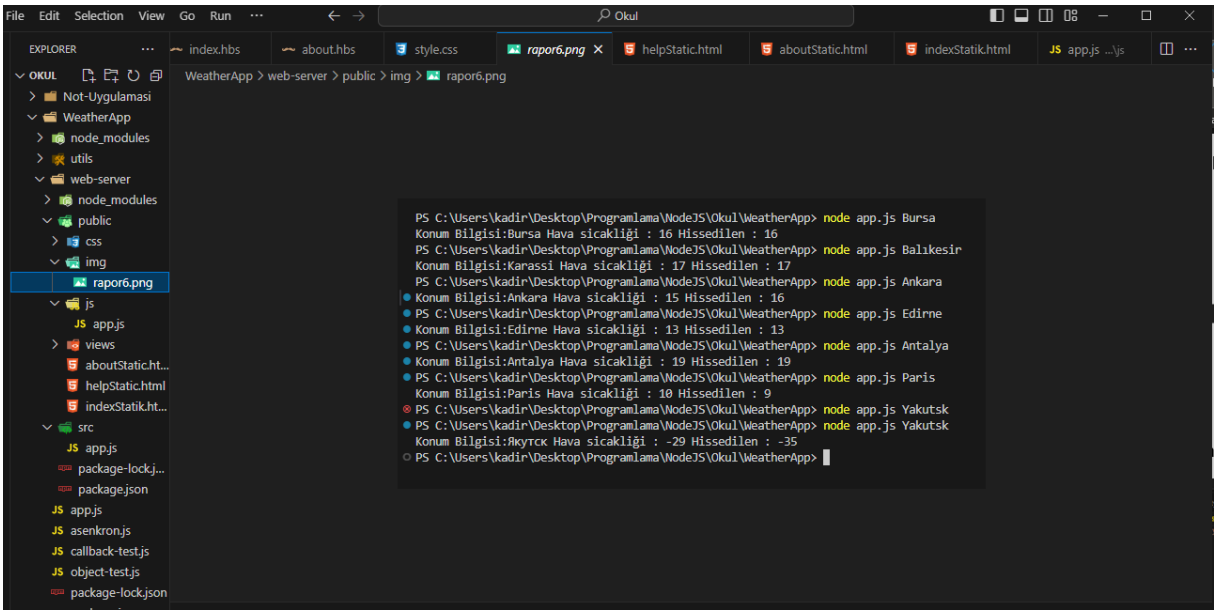
Burada __dirname core modül olan path modülüyle birlikte gelir ve mevcut dosya konumuna eriştirir. Yani burada yapılan şey app.js dosyasının bulunduğu konumu belirt , ardından  public klasörüne gir. Şu an public klasörün path bilgisini biliyoruz. Şimdi bu klasörün statik dosyaları belirttiğini göstermek için app.use(Express.static()) metodunu kullanıyoruz. Bunu yaptıktan sonra bu klasörün altında css,img,views,js klasörlerini oluşturup aynı zamanda test amaçlı statik html içeriklerini oluşturalım. Görüntümüz şu şekilde olacak:



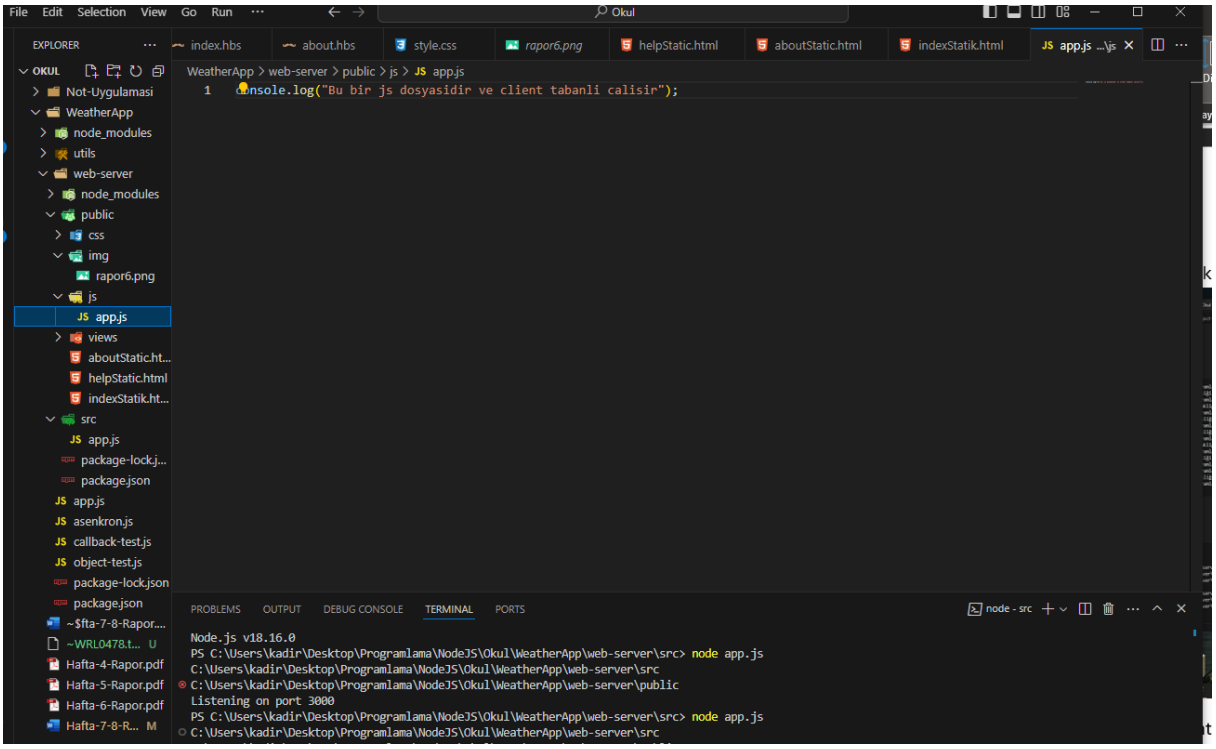

Css dosyamızın içeriği şu şekilde olsun:



İmg klasörümüzün altına şu şekilde bir fotoğraf koyalım:




Js klasörümüz altına da js uzantılı bir dosya ve içerisine de basit bir kod yazalım:

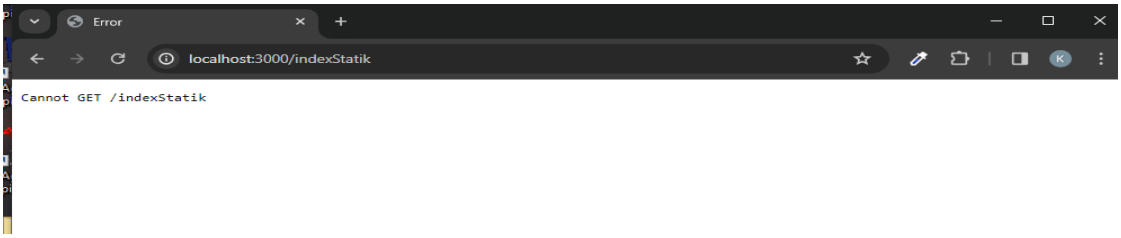



Örneğimiz içerisindeki indexStatik.html içeriği ise şu şekildedir. Yukarıda oluşturduğumuz css ve js içeriklerini html dosyamızın içeriğine head etiketi üzerinde dahil ediyoruz.

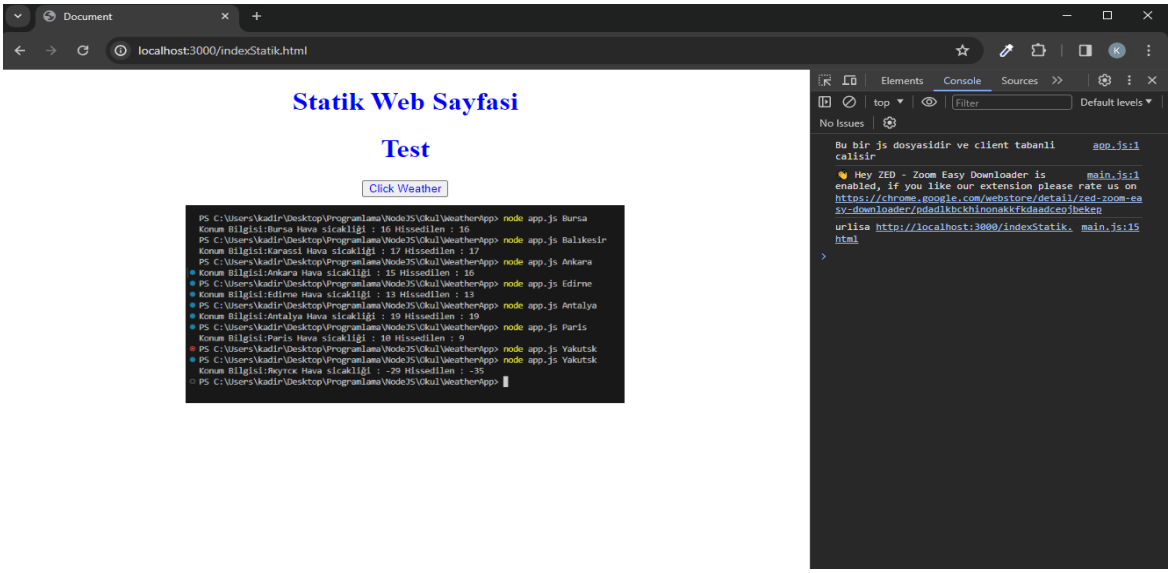
```
    <link rel="stylesheet" href="./css/style.css">
    <script src="./js/app.js"></script>
```

```
WeatherApp > web-server > public > indexStatik.html > html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <link rel="stylesheet" href="./css/style.css">
</head>
<body>
    <h1> Statik Web Sayfasi</h1>
    <div>
        <h1>Test</h1>
        <a href="/weather"><button>Click Weather</button><a/>
    </div>

    <img src="./img/rapor6.png" alt="">
</body>
</html>
```

Şimdi app.js dosyamızı çalıştırdıktan sonra eğer mevcut dosyanın ismini url yazarsak karşımıza şu ekran gelir:




Şu an statik sayfanın içeriğinin yüklenmemesinin sebebi şudur: Bu bir statik html dosyası olduğu için ve şu an indexStatik için bir router(app.get metodunu) kullanmadığımız için bize bu hatayı veriyor. Eğer indexStatik dosyamızın çalışmasını istiyorsak router yazmamız gerekir. Bu statik dosya farklı şekilde şöyle çalıştırabiliriz. indexStatik yerine url'de indexStatik.html yazarsak bu sefer doğru şekilde çalışır ve bu ekran karşımıza gelir:



Şimdi dinamik sayfaları kullanalım. Dinamik sayfaları kullanmak adına hbs view engine app.js içerisine set etmiştik. Şu an doğrudan kullanmaya çalışırsak bize hata verir çünkü hbs view engine default olarak bir yolu bulmaya çalışır. Bu yolu kendi projemize göre set etmemiz gerekirse yapacağımız işlem yine çok basittir. App.js dosyamıza gelip şu iki satır kodu yazmaktır:

```
const viewsPath = path.join(__dirname,'../public/views');
app.set('views',viewsPath);
```

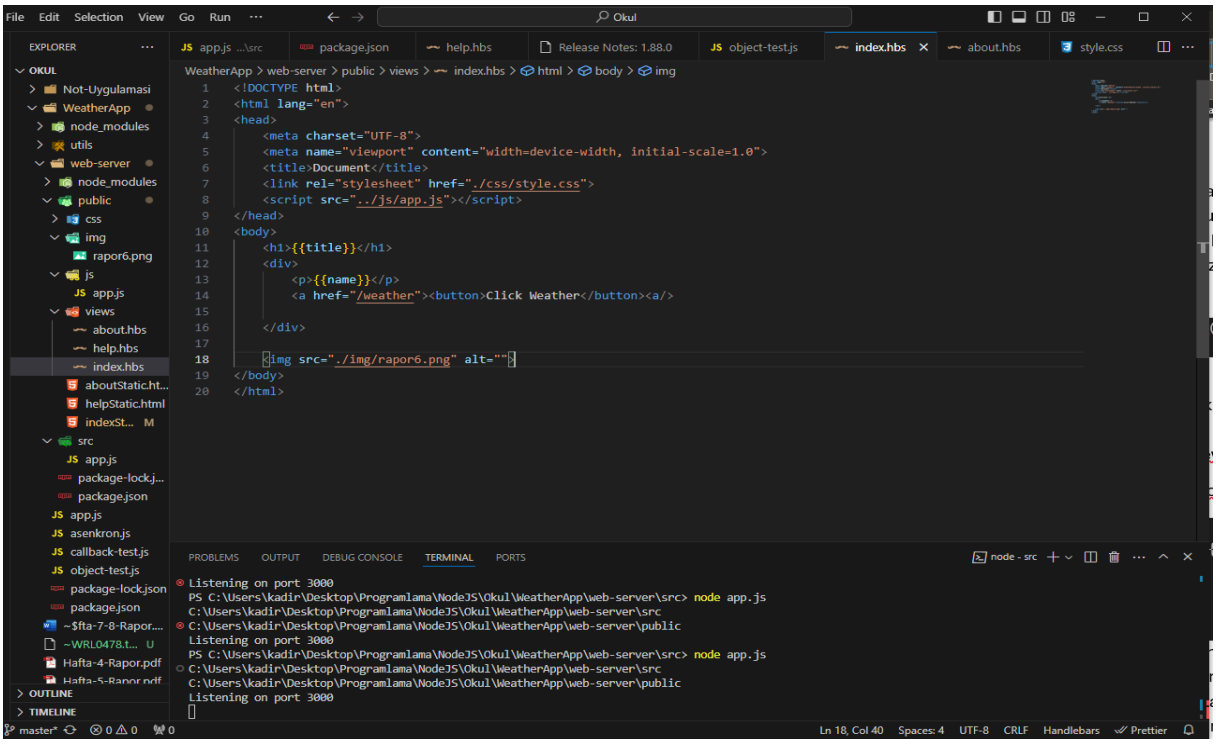
Bu kodun da amacı üstte statik dosyaların path belirtmemizle aynıdır. viewPath değişkeni mevcut konumu alır, ardından public klasörüne girip views klasörüne konumlanır. App.set('views',viewsPath) ile default olan views yolunu değiştirmiş olduk.

Dinamik sayfaları kullanırken router oluşturmamız gerekir. Örnek router:

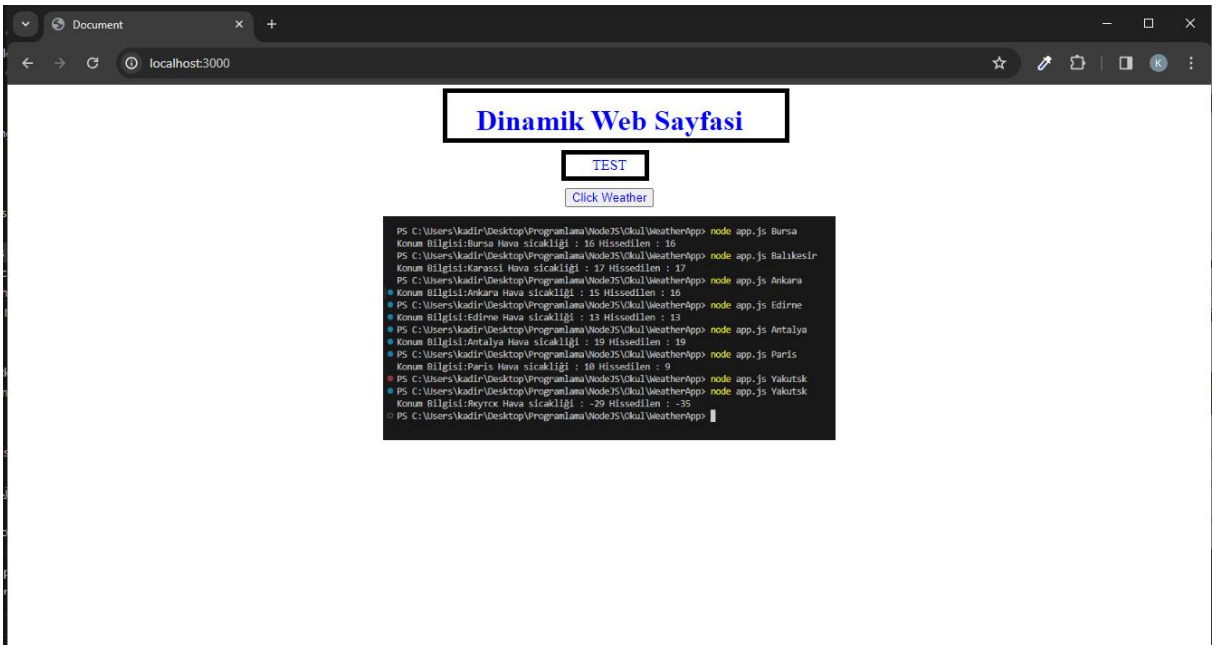
```
app.get('/',(req,res)=>{
    res.render('index.hbs',{
        title: "Dinamik Web Sayfasi",
        name:"TEST"
    })
});
```

Doğrudan dinamik html içeriğini yollamak için callback fonksiyonu içerisine res.render metodunu kullanmamız gerekir. Bu metodun ilk parametresi yukarıda belirlediğimiz view engine path'ini kullanır ve orada yazılan index.hbs dosyasının olup olmadığına bakar eğer varsa ikinci parametre olarak ise dinamik verileri nesne halinde gönderir. Bu şekilde index.hbs dosyası url'de localhost:3000 yazdığımızda görüntülenir.

Şimdi views klasörü altında index.hbs dosyamıza bakalım.



hbs içeriğine bakacak olursak doğrudan bir html içeriğini kullanır. Tek farkı şudur: nesne olarak yolladığımız değişkenleri {{yollananNesneAdi}} html etiketleri içerisine istediğimiz gibi kullanabiliyoruz. Yukarıdaki örnekte yolladığımız title değişkenin içeriği h1 etiketleri arasına alınmış , name değişkeni ise p  etiketleri içerisine alınmış app.js dosyasını çalıştırıp tarayıcıda localhost:3000 url'ine gidersek alacağımı çıktı şu şekilde olacaktır.




Görüldüğü gibi res.render üzerinden yollanan nesnedeki içerik doğrudan html dosyasının içeriğine aktarıldı. Router yardımıyla da '/' url'ine yönlendirildi ve app.listen(3000) sayesinde bu port üzerinden local olarak sunucu çalıştırıldı.

```
app.get('/help',(req,res)=>{/*
    res.send([{name:"Kadir"},
                {name:"Kdr"}]); //Json formatında yolladik*/
    res.render('help.hbs',{
        title:"Yardim Sayfasi Dinamik Baslik",
        name:"Yardim sayfasi dinamik icerik"
    })
});

app.get('/about',(req,res)=>{
    res.render('about.hbs',{
        title:"Dinamik About Sayfasi",
        name:"About sayfasi Dinamik Test"
    });
});
```

Diğer routerları ve bu routerlar içerisinde kullanılacak hbs içeriklerini de hazırlayalım

WeatherApp > web-server > public > views > help.hbs > html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <link rel="stylesheet" href="./css/style.css">
    <script src="./js/app.js"></script>
</head>
<body>
    <h1>{{title}}</h1>
    <p>{{name}}</p>

</body>
</html>
```

WeatherApp > web-server > public > views > about.hbs > html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <link rel="stylesheet" href="./css/style.css">
    <script src="./js/app.js"></script>
</head>
<body>
    <h1>{{title}}</h1>
    <p>{{name}}</p>

</body>
</html>
```

Bunları da app.js dosyamızı çalıştırıp tarayıcı üzerinde test edecek olursak karşımıza çıkacak sonuç şu şekilde olacaktır:


Tüm bu nodejs kodlarımızı yazarken sürekli çalıştırmakla uğraşmamak adına nodemon modülünü projemize dahil edebiliriz. Bu modülü npm i nodemon yazarak indirebiliriz. Modülü yükledikten sonra mevcut dosyamızı sürekli node dosyaAdi.js ile her değişiklikte tekrar çalıştırmak yerine nodemon dosyaAdi.js yazarak bir kez çalıştırıyoruz. Bunu yaptığımızda kaydettiğimiz her değişiklik algılanır ve tekrar güncel bir şeklide otomatik olarak çalıştırılır.