

# ***BURSA TEKNİK ÜNİVERSİTESİ***

**Mühendislik ve Doğa Bilimleri Fakültesi – Bilgisayar  
Mühendisliği Bölümü**



**BLM0470 NodeJS İle Web Programlama**

**2023-2024 Bahar Dönemi**

## **9.Hafta Raporu**

**Berkan SERBES – 22360859353**

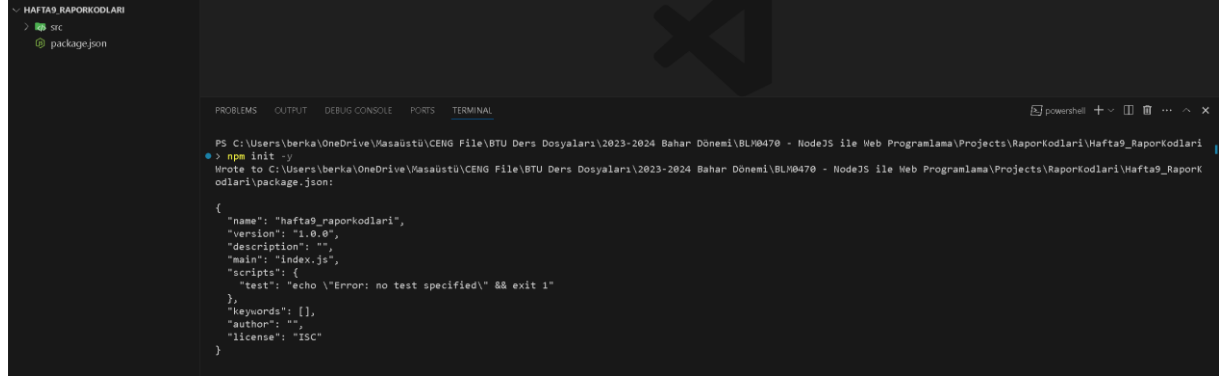
## İçindekiler

1.Dinamik Sayfalarla Hava Durumu Uygulaması Geliştirme.....	3
1.1    Uygulamanın Oluşturulması ve Gerekli Paketlerin Yüklenmesi .....	3
1.2    Klasör Yapılarının Oluşturulması .....	4
1.2.1    Views Klasörünün Oluşturulması .....	5
1.2.2 Partials Klasörünün Oluşturulması.....	7
1.3    Custom Error Sayfası .....	14
1.4    Query String .....	17
1.4.1 Query String Kullanım Örneği .....	17
1.4.2 Dış Servis Fonksiyonlarını Oluşturma .....	18
1.4.3 /weather Endpointinden Veri Çekme İşlemi.....	23
2. Kaynakça .....	27

# 1.Dinamik Sayfalarla Hava Durumu Uygulaması Geliştirme

## 1.1 Uygulamanın Oluşturulması ve Gerekli Paketlerin Yüklenmesi

Projemizin olduğu ana dizine gidip komut satırı yardımıyla **npm init -y** komutunu kullanarak **package.json** dosyamızı oluşturalım.



```
PS C:\Users\berka\OneDrive\Masaüstü\CENG File\BTU Ders Dosyaları\2023-2024 Bahar Dönemi\BLM0470 - NodeJS ile Web Programlama\Projects\RaporKodlari\Hafta9_RaporKodlari> npm init -y
Wrote to C:\Users\berka\OneDrive\Masaüstü\CENG File\BTU Ders Dosyaları\2023-2024 Bahar Dönemi\BLM0470 - NodeJS ile Web Programlama\Projects\RaporKodlari\Hafta9_RaporKodlari\package.json:

{
  "name": "hafta9_raporkodlari",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\"Error: no test specified\\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Sonrasında ise uygulama boyunca kullanacağımız paketleri **npm install** komutu aracılığıyla yüklememiz gerekmektedir. Uygulamamızda kullanacağımız paketler **express**, **hbs**, **dotenv** ve **postman-request** paketleridir.

Bu paketlerin her biri farklı amaçlar için kullanılır.

**Express**, hızlı ve esnek bir web uygulama çerçevesidir ve HTTP sunucularını yönetir, yönlendirme işlemlerini kolaylaştırır.

**hbs**, Handlebars.js için bir view engine'dir ve dinamik veri ile HTML şablonlarını birleştirmemizi sağlar.

**dotenv**, uygulama yapılandırma bilgilerini .env dosyalarından okuyarak erişimi kolaylaştırır, bu sayede hassas bilgileri kaynak koduna eklemek yerine güvenli bir şekilde saklayabiliriz.

**postman-request**, HTTP isteklerini yapmak, yanıtları işlemek ve API'lerle etkileşimde bulunmak için kullanılan bir HTTP istemcisidir ve genellikle API testlerini otomatikleştirmek için kullanılır.

Bu paketleri **npm i dotenv express hbs postman-request** komutuyla yükledikten sonra package.json dosyamız aşağıdaki görseldeki gibi olacaktır.

```
package.json > ...
1  {
2    "name": "hafta9_raporkodlari",
3    "version": "1.0.0",
4    "description": "",
5    "main": "index.js",
6    "scripts": {
7      "test": "echo \"Error: no test specified\" && exit 1"
8    },
9    "keywords": [],
10   "author": "",
11   "license": "ISC",
12   "dependencies": {
13     "dotenv": "^16.4.5",
14     "express": "^4.19.2",
15     "hbs": "^4.2.0",
16     "postman-request": "^2.88.1-postman.33"
17   }
18 }
```

PROBLEMS OUTPUT DEBUG CONSOLE PORTS TERMINAL

```
PS C:\Users\berka\OneDrive\Masaüstü\CENG File\BTU Ders Dosyaları\2023-2024 Bahar Dönemi
> npm i dotenv express hbs postman-request
npm WARN deprecated har-validator@5.1.5: this library is no longer supported

added 125 packages, and audited 126 packages in 1m

16 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities
```

## 1.2 Klasör Yapılarının Oluşturulması

Projemizin kök dizinine giderek **src** adında yeni bir klasör oluşturalım. Bu klasör içerisine, projenin başlatılacağı ana dosya olarak kullanılacak olan index.js adında bir dosya oluşturalım. Bu dosya, projenin temel yapılandırması ve işlevselliği için gereken başlangıç noktasını sağlayacak.

```
src > JS index.js > ...
1  const hbs = require("hbs");
2  const express = require("express");
3  const app = express();
4
5  app.set("view engine", "hbs");
6
7  const PORT = 3000;
8
9  app.listen(PORT, () => {
10    console.log(`Server is running on port ${PORT}`);
11  });
12
```

Yukarıda görseldeki yer alan kod bloğu, bir Node.js uygulamasında web sunucusu oluşturmak için kullanılan temel bir yapıyı içerir. İlk olarak, hbs ve express adlı paketler **require** anahtar kelimesiyle projeye eklenir. hbs, Handlebars.js için bir view engine olarak kullanılırken, express web uygulama çerçevesini oluşturmak için kullanılır. Ardından, express fonksiyonu app adında bir nesne oluşturur.

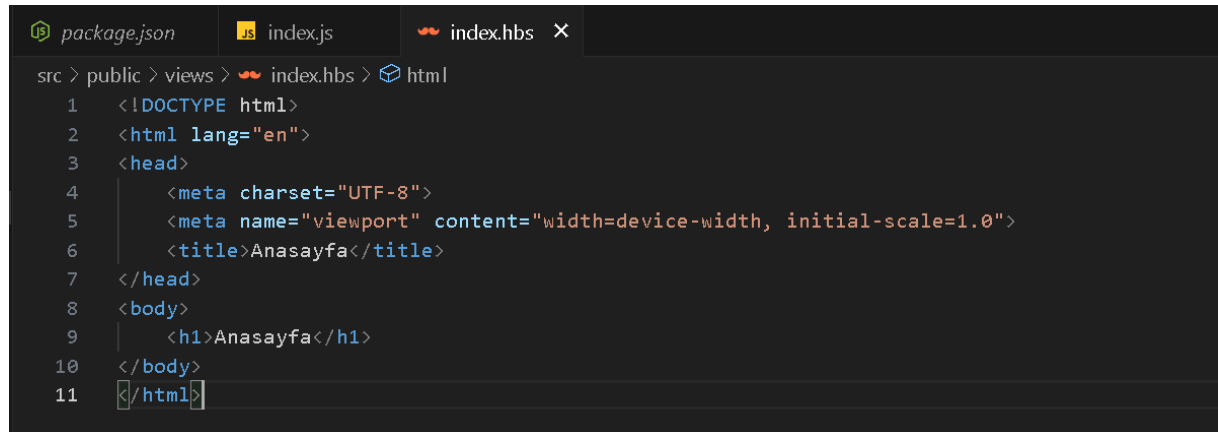
**app.set("view engine", "hbs")** satırı, Express uygulamasında view engine olarak Handlebars'ı kullanacağını belirtir. Bu, uygulamanın HTML sayfalarını dinamik olarak oluşturmak için Handlebars'ı kullanacağını gösterir.

**const PORT = 3000** ifadesi, uygulamanın hangi portta çalışacağını belirtir. Burada, 3000 numaralı port kullanılıyor, ancak isteğe bağlı olarak başka bir port da seçilebilir.

Son olarak, **app.listen(PORT, () => { console.log(Server is running on port \${PORT}) })** satırı, uygulamanın belirtilen portta çalıştırılmasını sağlar. Sunucu başlatıldığında, konsola **"Server is running on port 3000"** gibi bir mesaj yazdırılır. Bu, uygulamanın başarıyla çalıştığını ve belirtilen portta istekleri dinlediğini gösterir.

### 1.2.1 Views Klasörünün Oluşturulması

Uygulamamızın yapılandırması için, src klasörünün içinde public klasörü altında views adında bir alt klasör oluşturalım. Bu views klasörü, uygulamanın kullanıcı arayüzü için HTML dosyalarını içerecek. Ardından, views klasörü içine index.hbs adında bir dosya oluşturuyoruz. index.hbs dosyası, uygulamanın ana sayfasını temsil edecek.



```
package.json index.js index.hbs X
src > public > views > index.hbs > html
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Anasayfa</title>
7 </head>
8 <body>
9   <h1>Anasayfa</h1>
10 </body>
11 </html>
```

Bir sonraki adım, oluşturduğumuz bu sayfayı kullanıcı **"/** kök route'a istek attığında render etmek olacaktır.

```

1  const hbs = require("hbs");
2  const path = require("path");
3  const express = require("express");
4  const app = express();
5
6  const PUBLIC_DIRECTORY = path.join(__dirname, "./public");
7  const VIEWS_DIRECTORY = path.join(PUBLIC_DIRECTORY, "./views");
8
9  app.use(express.static(PUBLIC_DIRECTORY));
10
11 app.set("view engine", "hbs");
12 app.set("views", VIEWS_DIRECTORY);
13
14 app.get("/", (req, res) => {
15   res.render("index");
16 });
17
18 const PORT = 3000;
19
20 app.listen(PORT, () => {
21   console.log(`Server is running on port ${PORT}`);
22 });
23

```

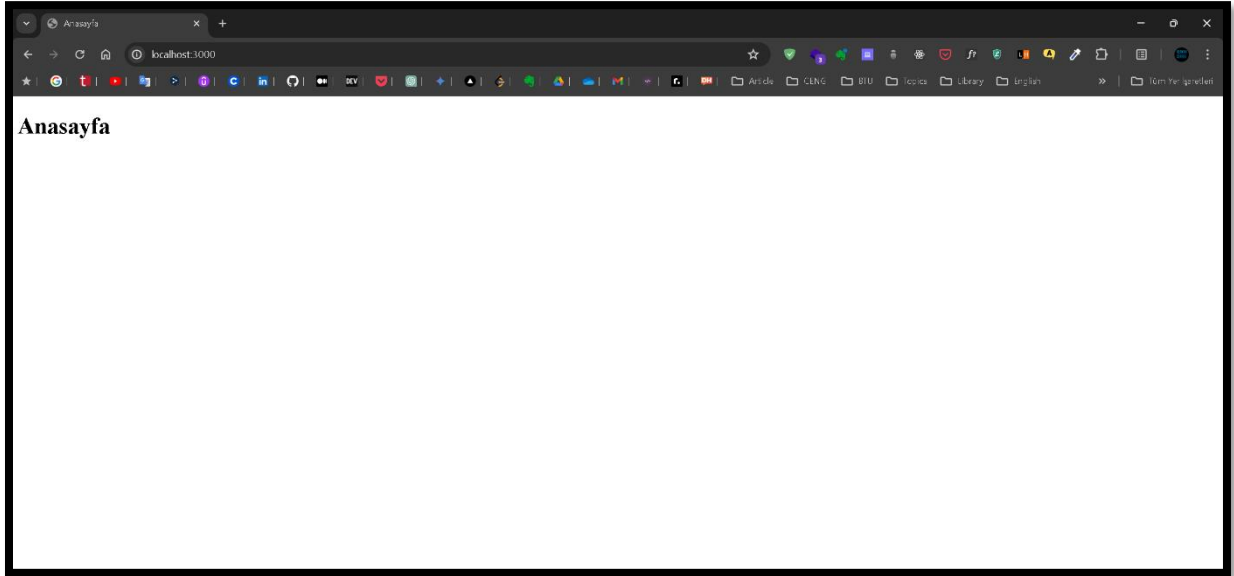
Yukarıdaki kod parçasında, ilk olarak, **path** modülü kullanılarak **PUBLIC\_DIRECTORY** ve **VIEWS\_DIRECTORY** değişkenleri oluşturulur. **PUBLIC\_DIRECTORY**, sunucu tarafından sunulacak olan statik dosyaların bulunduğu dizini temsil ederken, **VIEWS\_DIRECTORY** ise Handlebars şablon dosyalarının bulunduğu dizini gösterir.

Ardından, **app.use(express.static(PUBLIC\_DIRECTORY))** satırıyla Express uygulamasına, **PUBLIC\_DIRECTORY** içindeki statik dosyaların sunulması için bir middleware eklenir. Bu sayede, tarayıcı istekleri doğrudan bu dizindeki dosyalara yönlendirilir ve sunulur.

**app.set("view engine", "hbs")** ve **app.set("views", VIEWS\_DIRECTORY)** satırları, Express uygulamasının view engine olarak Handlebars'ı ve view dosyalarının bulunduğu dizini belirlemesini sağlar.

Son olarak, **app.get("/", (req, res) => { res.render("index") })** satırıyla, kök dizine gelen GET istekleri için bir yönlendirme tanımlanır. Bu istekler karşısında, **index.hbs** dosyası render edilerek kullanıcıya sunulur.

Uygulamamız çalıştırıp kök url'ye gittiğimizde aşağıdaki çıktıyı almaktayız.



## 1.2.2 Partials Klasörünün Oluşturulması

Partials, web uygulamalarında kullanılan tekrar eden yapıların veya bileşenlerin ayrı dosyalarda saklanmasını sağlayan bir kavramdır. Bu yapılar genellikle site genelinde birden fazla yerde kullanılan, ancak her seferinde aynı HTML kodunu tekrar yazmaktansa bir kez tanımlanıp kullanılabilen yapıları ifade eder. Örneğin, bir site genelindeki navigasyon menüsü, footer veya yan panel gibi bileşenler partialslara örnek olarak verilebilir. Partialslar, kod tekrarını önler ve bakımı kolaylaştırır çünkü bir bileşenin tüm sayfalara aynı şekilde uygulanması gerektiğinde, sadece tek bir dosyada güncelleme yapmak yeterlidir. Ayrıca, sayfa yapısını daha modüler hale getirir, böylece bir bileşenin içeriği veya tasarımı değiştiğinde sadece o bileşeni içeren partial dosyası güncellenir, diğer sayfalar etkilenmez.

Uygulamamızda 2 farklı partials bileşeni kullanacağız. Bu bileşenler **header** ve **footer** olacaktır. Header, genellikle uygulamanın üst kısmında bulunan ve genel navigasyon bağlantıları, logo ve kullanıcı giriş/çıkış işlevleri gibi bileşenleri içerir. Footer ise uygulamanın alt kısmında yer alır ve genellikle site haritası, iletişim bilgileri, sosyal medya bağlantıları gibi bilgileri içerir. İlk olarak views klasörünün altına **partials** adında bir klasör açalım. Bu klasörün içerisine **header.hbs** ve **footer.hbs** adında iki dosya oluşturalım.

Views klasöründe kullanılan HTML sayfalarının daha etkileyici bir görünüm kazanması için, bu sayfaların stil dosyalarını daha organize bir şekilde yönetmek amacıyla **public** klasörü altında bir **css** isimli bir klasör oluşturalım. Bu yeni **css** klasörü altında, her bir HTML sayfasının özel stillerini içerecek olan CSS dosyalarını oluşturacağız. Böylelikle, her HTML dosyası için ayrı bir CSS dosyası oluşturarak kodu daha modüler ve yönetilebilir hale getiriyoruz. Bu adım, web uygulamamızın görünümünü geliştirmek ve bakımını kolaylaştırmak için önemlidir.

**header.hbs** dosyamızın html içeriği ve css içeriği aşağıdaki görsellerdeki gibi olacaktır.

```
src > public > views > partials > 🍷 header.hbs > 📦 html
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4  |   <meta charset="UTF-8">
5  |   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6  |   <link rel="stylesheet" href="../../css/header.css">
7  </head>
8  <body>
9  |   <div class="header-container">
10 |       <div>
11 |           <h1 class="app-name">
12 |               Weather App
13 |           </h1>
14 |       </div>
15 |       <div class="link-container">
16 |           <a href="/">Anasayfa</a>
17 |           <a href="/about">Hakkımızda</a>
18 |       </div>
19 |       <div>
20 |           
21 |       </div>
22 |   </div>
23 </body>
24 </html>
```



```
src > public > css > header.css > .header-container
1  body {
2    background-color: #f5f5f5;
3    margin: 0;
4    padding: 0;
5    box-sizing: border-box;
6  }
7  .header-container {
8    padding: 0px 30px;
9    background-color: black;
10   height: 10vh;
11   display: flex;
12   justify-content: space-between;
13   align-items: center;
14 }
15 .link-container {
16   display: flex;
17   gap: 5rem;
18   justify-content: center;
19   align-items: center;
20   height: 100%;
21   > a {
22     color: white;
23     font-size: 1.5rem;
24     text-decoration: none;
25     &:hover {
26       color: aqua;
27       text-decoration: underline;
28       transition: ease-in-out 0.3s all;
29     }
30   }
31 }
32 h1 {
33   color: brown;
34   text-align: center;
35 }
36
37 .app-name {
38   color: antiquewhite;
39 }
40
41 .logo {
42   width: 50px;
43   height: 50px;
44 }
45
```

Bu dosyalarımızı oluşturduktan sonra hbs motoruna bu partials klasörümüzü tanıtmamız gerekmektedir. index.js dosyamızı aşağıdaki gibi düzenleyelim.

```

src > JS index.js > ...
1  const hbs = require("hbs");
2  const path = require("path");
3  const express = require("express");
4  const app = express();
5
6  const PUBLIC_DIRECTORY = path.join(__dirname, "../public");
7  const VIEWS_DIRECTORY = path.join(PUBLIC_DIRECTORY, "../views");
8  const PARTIALS_DIRECTORY = path.join(VIEWS_DIRECTORY, "../partials");
9
10 app.use(express.static(PUBLIC_DIRECTORY));
11
12 app.set("view engine", "hbs");
13 app.set("views", VIEWS_DIRECTORY);
14
15 hbs.registerPartials(PARTIALS_DIRECTORY);
16
17 app.get("/", (req, res) => {
18   | res.render("index");
19   | });
20
21 const PORT = 3000;
22
23 app.listen(PORT, () => {
24   | console.log(`Server is running on port ${PORT}`);
25   | });

```

Yukarıdaki kodda, partials klasörünün dizin yolunu **PARTIALS\_DIRECTORY** adında bir değişkene atayarak, `hbs.registerPartials(PARTIALS_DIRECTORY)` ifadesiyle bu dizini partials görünümüleri için kullanılabilir hale getiriyoruz. Bu işlem, Handlebars'ın kısmi (partials) dosyaları bulmak ve kullanmak için belirli bir dizini tanımasını sağlar.

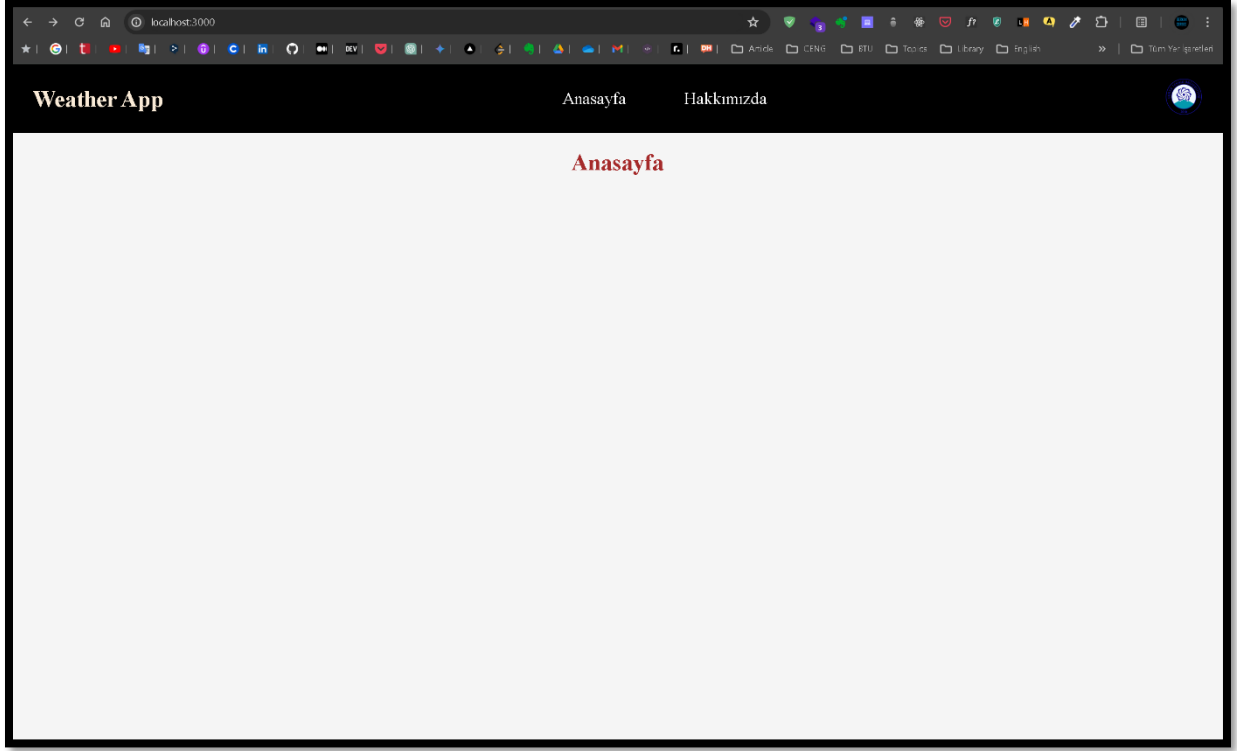
Şimdiki adımımız, oluşturduğumuz bu header partial view'ını `index.hbs` dosyamızda kullanmak olacaktır. Aşağıdaki şekilde gösterildiği gibi kullanacağız `{{> 'partials-name'}}` biçiminde.

```

1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4  |   <meta charset="UTF-8">
5  |   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6  |   <title>Anasayfa</title>
7  </head>
8  <body>
9  |   {{> header}}
10 |   <h1>Anasayfa</h1>
11 </body>
12 </html>

```

Uygulamamızı çalıştırdığımızda aşağıdaki çıktıyı almaktayız.



Header partials görünümümüzü bu şekilde oluşturduktan sonra sıra footer partials görünümünü oluşturmakta. Aşağıdaki görsellerdeki gibi kodumuzu yazalım.

```
src > public > views > partials > 🍷 footer.hbs > 📦 html
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <meta name="viewport" content="width=device-width, initial-scale=1.0">
6    <link rel="stylesheet" href="../../css/footer.css" />
7  </head>
8  <body>
9    <div class="footer-container">
10     <p>{{developer}} tarafından geliştirilmiştir.</p>
11   </div>
12 </body>
13 </html>
```

```
src > public > css > footer.css > ...
1  body {
2      background-color: #f5f5f5;
3      margin: 0;
4      padding: 0;
5      box-sizing: border-box;
6  }
7  .footer-container {
8      position: absolute;
9      bottom: 0;
10     color: white;
11     background-color: black;
12     width: 100%;
13     display: flex;
14     justify-content: center;
15 }
16
```

Şimdide oluşturduğumuz footer partial'ını index.js dosyamızda kullanalım. Dikkat ettiyseniz footer.hbs dosyasındaki içerikte p etiketleri arasına **developer** değişkeni render edilmiş. Bu değişkeni render edebilmek için bu partial görünümünü kullanan ilgili view sayfasına bu değişkeni göndermemiz gerekmektedir.

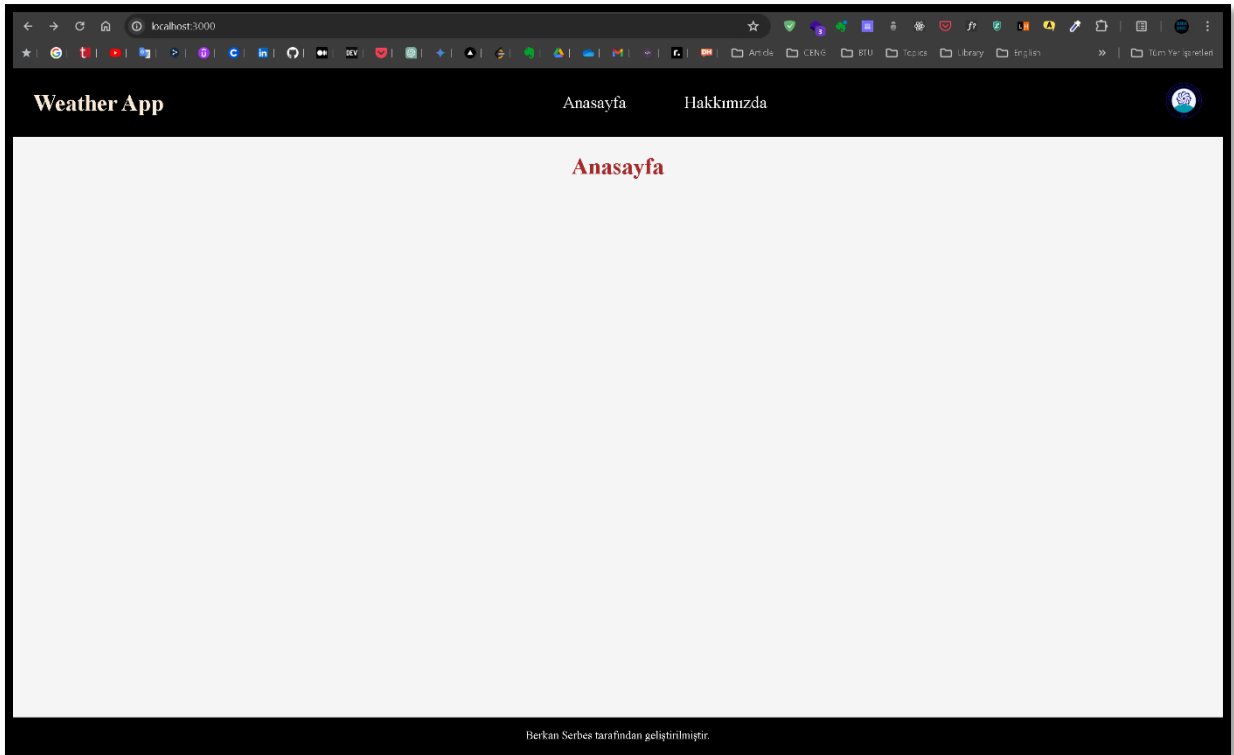
Aşağıdaki görselde 18. satırda bu değişkeni nasıl göndereceğimiz gösterilmiştir.

```
1  const hbs = require("hbs");
2  const path = require("path");
3  const express = require("express");
4  const app = express();
5
6  const PUBLIC_DIRECTORY = path.join(__dirname, "./public");
7  const VIEWS_DIRECTORY = path.join(PUBLIC_DIRECTORY, "./views");
8  const PARTIALS_DIRECTORY = path.join(VIEWS_DIRECTORY, "./partials");
9
10 app.use(express.static(PUBLIC_DIRECTORY));
11
12 app.set("view engine", "hbs");
13 app.set("views", VIEWS_DIRECTORY);
14
15 hbs.registerPartials(PARTIALS_DIRECTORY);
16
17 app.get("/", (req, res) => {
18     res.render("index", { developer: "Berkan Serbes" });
19 });
20
21 const PORT = 3000;
22
23 app.listen(PORT, () => {
24     console.log(`Server is running on port ${PORT}`);
25 })
```

index.hbs dosyamızın içeriği ise aşağıdaki gibi güncellenmesi gerekmektedir.

```
src > public > views > index.hbs > html
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <meta name="viewport" content="width=device-width, initial-scale=1.0">
6    <title>Anasayfa</title>
7  </head>
8  <body>
9    {{header}}
10   <h1>Anasayfa</h1>
11   {{footer}}
12 </body>
13 </html>
```

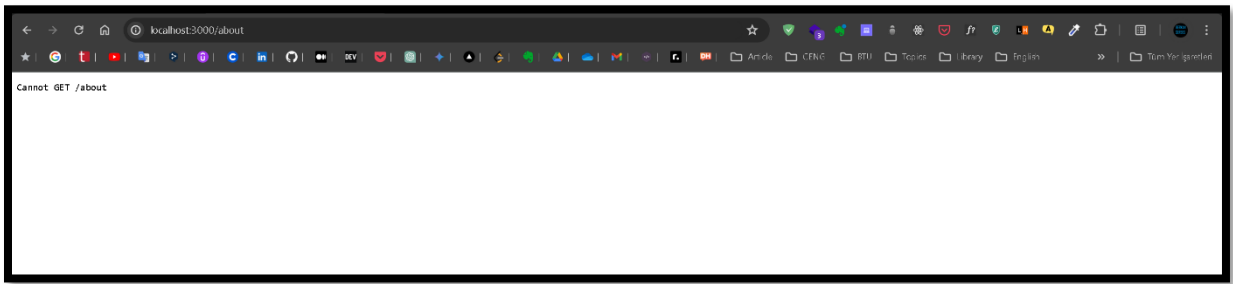
Kodlarımızı bu şekilde düzenledikten sonra index.hbs dosyamızın görüntüsü tarayıcıda aşağıdaki gibi olacaktır.



## 1.3 Custom Error Sayfası

Custom error sayfası, kullanıcıların var olmayan veya geçersiz bir URL'ye istek yaptıklarında karşılaştıkları özel bir sayfadır. Bu sayfa genellikle "404 Not Found" hatasıyla ilişkilendirilir ve kullanıcıya "Sayfa Bulunamadı" veya benzeri bir mesaj sunar. Amacı, kullanıcıların karşılaştığı hataları daha anlaşılır hale getirmek ve böylece kullanıcı deneyimini geliştirmektir. Ayrıca, custom error sayfası, web uygulamasının marka kimliğine uygun olarak tasarlanabilir ve hata durumunda kullanıcıya yönlendirici veya yardımcı bilgiler sağlayarak kullanıcının yönlendirilmesini kolaylaştırır. Bu sayede, kullanıcılar hata durumunda dahi uygulamayla etkileşimlerini sürdürebilir ve olumlu bir deneyim yaşayabilirler.

Şuan ki uygulamamızda herhangi bir custom error sayfası bulunmamaktadır. Kullanıcı tanımlanmayan bir url'e istek attığında aşağıdaki görüntü karşımıza gelecektir.



Custom error sayfası oluşturmak için ilk önce views klasörünün altına **404.hbs** adında bir dosya oluşturalım. Bu dosyanın html ve css içeriği aşağıdaki gibi olacaktır.

```
src > public > views > 404.hbs > ...
1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8" />
5      <meta name="viewport" content="width=device-width, initial-scale=1.0" />
6      <title>Error</title>
7      <link rel="stylesheet" href="../css/404.css" />
8
9    </head>
10   <body>
11     <div class="container">
12       <h1><span class="uri">{{baseUrl}}</span> <span class="arrow">=></span> 404 Not Found</h1>
13     </div>
14   </body>
15 </html>
16
```

```
src > public > css > 404.css > ...
1  body {
2    background-color: #f5f5f5;
3    margin: 0;
4    padding: 0;
5    box-sizing: border-box;
6  }
7  .container {
8    display: flex;
9    justify-content: center;
10   align-items: center;
11   height: 100vh;
12 }
13
14 .uri {
15   color: red;
16 }
17
18 .arrow {
19   color: gray;
20 }
21
```

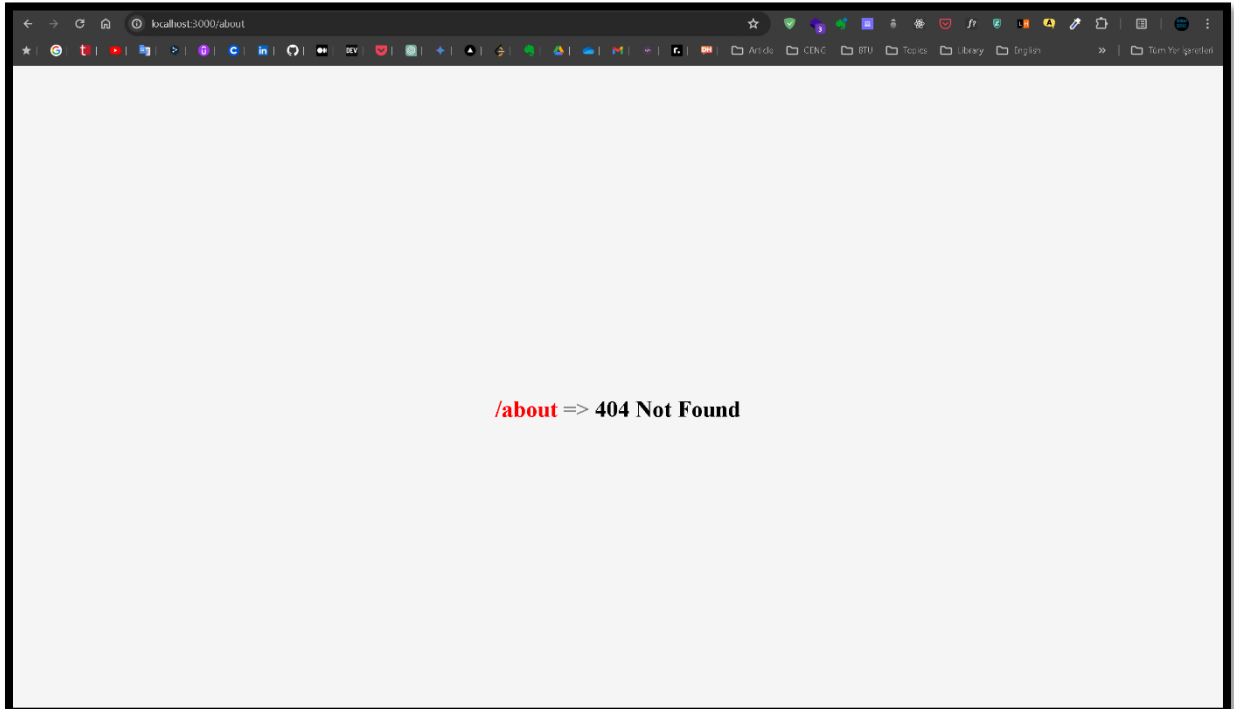
Sayfamızı bu şekilde oluşturduktan sonra yapmamız gereken şey bu dosyayı index.js dosyasında ilgili route'da tanımlamaktır.

```
1  const hbs = require("hbs");
2  const path = require("path");
3  const express = require("express");
4  const app = express();
5
6  const PUBLIC_DIRECTORY = path.join(__dirname, "./public");
7  const VIEWS_DIRECTORY = path.join(PUBLIC_DIRECTORY, "./views");
8  const PARTIALS_DIRECTORY = path.join(VIEWS_DIRECTORY, "./partials");
9
10 app.use(express.static(PUBLIC_DIRECTORY));
11
12 app.set("view engine", "hbs");
13 app.set("views", VIEWS_DIRECTORY);
14
15 hbs.registerPartials(PARTIALS_DIRECTORY);
16
17 app.get("/", (req, res) => {
18   res.render("index", { developer: "Berkan Serbes" });
19 });
20
21 app.get("*", (req, res) => {
22   res.render("404", { baseUrl: req.url });
23 });
24
25 const PORT = 3000;
26
27 app.listen(PORT, () => {
28   console.log(`Server is running on port ${PORT}`);
29 });
```

```
app.get("*", (req, res) => {  
  res.render("404", { baseUrl: req.url });  
});
```

Bu kod parçası, Express uygulamasında herhangi bir istek için route tanımlar. ‘\*’ karakteri, tüm istek URL’lerini kapsar. `res.render("404", { baseUrl: req.url })` ifadesi ise, bu durumda kullanıcılara gösterilecek olan sayfayı oluşturur. `res.render()` metodu, bir Handlebars (hbs) şablonunu render etmek için kullanılır. Burada "404" adında bir şablon dosyası render edilir ve bu dosya, kullanıcılara gösterilecek özel bir sayfayı temsil eder. İkinci parametre olarak `{ baseUrl: req.url }` objesi verilir ve bu, 404 sayfasının içinde kullanılacak olan veri veya değişkenleri temsil eder. Bu örnekte, `baseUrl` adında bir değişken tanımlanır ve istenen URL’yi (`req.url`) içerir. Bu, 404 sayfasında kullanıcılara hangi URL’ye istekte bulunduklarını göstermek için kullanılır.

“/” dışında bir URL’ye istek attığımızda aşağıdaki çıktıyı alacağız.





## 1.4 Query String

Query string, web tarayıcıları tarafından URL'ye eklenen ve sunucuya bilgi iletmek için kullanılan bir veri formatıdır. Bu format genellikle "?" karakteri ile URL'den ayrılır ve anahtar-değer çiftlerinden oluşur. Query string, web uygulamalarında kullanıcıların belirli bir işlemi gerçekleştirmek için sunucuya veri iletmek amacıyla yaygın olarak kullanılır.

Query string'de her bir anahtar-değer çifti, anahtar ve değer arasında "=" işareti ile ayrılır. Birden fazla anahtar-değer çifti kullanılacaksa, bunlar "&" işareti ile ayrılır. Örneğin, bir arama motorunda "JavaScript" kelimesi ile yapılan bir aramanın URL'si şu şekilde olabilir:

`https://www.example.com/search?q=JavaScript`

Query string, sunucuya kullanıcı tercihlerini, arama sorgularını, sayfa numaralarını veya diğer verileri iletmek için kullanılabilir. Sunucu, bu bilgileri alır ve isteğe bağlı olarak işler.

JavaScript'te query string'i işlemek için **URLSearchParams** API'si kullanılabilir. Bu API, URL'deki query string'i almak, parametreleri okumak, parametre eklemek veya kaldırmak gibi işlemleri kolaylaştırır. Veya Node.js'te yer alan request objesinin içindeki query özelliği aracılığıyla da query string'deki parametreler alınabilir. Bu özellik, Express.js gibi web çatılarında isteğin ayrıntılarını taşıyan request objesinin bir parçası olarak bulunur. query özelliği, URL'deki query string'deki anahtar-değer çiftlerini bir obje olarak içerir. Bu sayede Node.js uygulamaları, istek URL'sinden gelen parametreleri kolayca okuyabilir ve işleyebilir. Örneğin, Express.js'te `req.query` kullanılarak query string parametrelerine erişilebilir ve istenilen işlemler gerçekleştirilebilir. Bu sayede web uygulamaları, kullanıcıların isteklerini doğru şekilde işleyebilir ve sonuçlarını uygun şekilde sunabilir.

Özetle, query string, web uygulamalarında kullanıcı ile sunucu arasında bilgi iletmek için yaygın olarak kullanılan bir veri formatıdır. Anahtar-değer çiftleri arasında "=" işareti ile ayrılarak ve "&" işareti ile farklı çiftler birbirinden ayrılarak kullanılır.

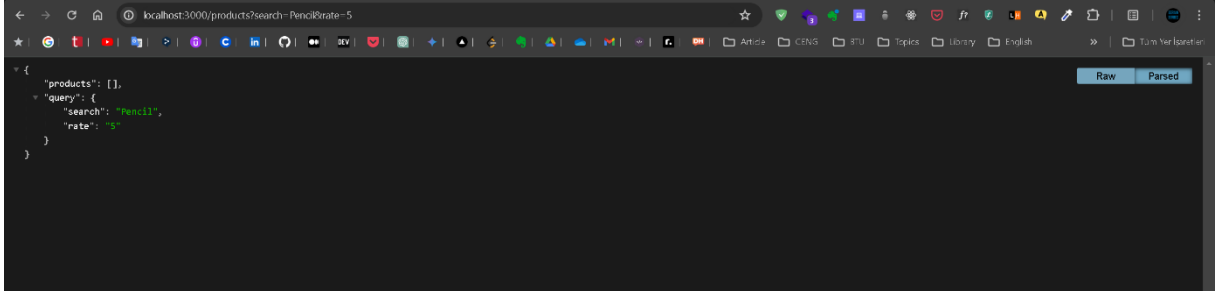
### 1.4.1 Query String Kullanım Örneği

```
app.get("/products", (req, res) => {  
  res.status(200).json({ products: [], query: req.query });  
});
```

Bu kod parçası, Express.js uygulamasında /products yoluna gelen GET isteklerini işler. Bir istek alındığında, 200 HTTP durum koduyla birlikte JSON formatında bir yanıt döndürülür. Yanıtta, products adında boş bir dizi ve kullanıcının isteğindeki query string parametrelerinin bir kopyası olan query adında bir nesne bulunur. `req.query` ifadesi, Express.js'in request nesnesinin bir özelliğidir ve gelen isteğin query string parametrelerini içeren bir nesne döndürür. Bu sayede, istekte bulunan kullanıcının gönderdiği query string parametreleri alınarak JSON yanıtı içinde kullanıma hazır hale getirilir.

Tarayıcımız üzerinden **'products'** route'una iki adet query string parametresi ile bir istek gönderelim. Route url'sinden sonra " ? " (soru işareti) koyalım daha sonra göndermek istediğimiz parametreleri **key = value** biçiminde gönderelim eğer birden fazla değer göndereceksek bunları " & " (ampersand) ile ayıralım.

Aşağıdaki görselde yer alan istekte, search parametresi "Pencil" değerini, rate parametresi ise 5 değerini içerir. Express.js uygulaması, bu isteği alır ve req.query özelliği aracılığıyla bu query string parametrelerine erişebilir.



```
{
  "products": [],
  "query": {
    "search": "Pencil",
    "rate": "5"
  }
}
```

## 1.4.2 Dış Servis Fonksiyonlarını Oluşturma

Projemizi dinamikleştirmek için hava durumu verilerini dış bir API'dan çekelim. Bunun için bu API ile bağlantı kuracak fonksiyonları yazmamız gerekmekte. Bunun için projemizde src klasörü altında utils adında bir klasör oluşturalım. Bu klasörün içinde de forecast.js ve geocode.js adında iki tane javascript dosyası oluşturalım. Daha sonra bu API'lere istek atarken ihtiyacımız olan API erişim anahtarlarını projemizin ana dizininde .env dosyası oluşturarak bu dosyaya yazalım aşağıdaki şekildeki gibi.



```
.env
1 MAPBOX_KEY="pk.eyJ1Ijo1YmVya2Fuc2VyyVZlIiw1YSI6ImNs dHpoYjd4 dTAwMWE ybXBk NWgwOWE 2NW01 fQ"
2 WEATHER_KEY="8160d4b42bcd0921f6c9c9c"
```

Sonrasında ise MapBox servisi geocode.js adlı dosyada aşağıdaki görseldeki gibi kodlanacaktır.

```
geocode.js
src > utils > geocode.js > ...
1  require("dotenv").config();
2  const request = require("postman-request");
3
4  const geocode = (address, callback) => {
5    const requestURL = `https://api.mapbox.com/geocoding/v5/mapbox.places/${encodeURIComponent(
6      address
7    )}.json?access_token=${process.env.MAPBOX_KEY}`;
8
9    try {
10     request({ url: requestURL, json: true }, (error, response) => {
11       if (error) {
12         callback("Unable to connect geocode services", undefined);
13       }
14       if (response.body.features.length === 0) {
15         callback("Unable to find location. Try another search", undefined);
16       }
17       const longitude = response.body?.features[0].center[0]; // boylam
18       const latitude = response.body?.features[0].center[1]; // enlem
19
20       callback(undefined, {
21         longitude,
22         latitude,
23         location: response.body.features[0].place_name,
24       });
25     });
26   } catch (err) {
27     callback(err, undefined);
28   }
29 };
30
31 module.exports = geocode;
32
```

WeatherStack servisini kullanan forecast.js adlı dosyamızda bulunan forecast fonksiyonu aşağıdaki şekildeki gibi kodlanacaktır.

```
geocode.js • forecast.js
src > utils > forecast.js > ...
1  require("dotenv").config();
2  const request = require("postman-request");
3
4  const forecast = (longitude, latitude, callback) => {
5    const requestURL = `http://api.weatherstack.com/current?access_key=${process.env.WEATHER_KEY}&query=${longitude},${latitude}`;
6
7    try {
8     request({ url: requestURL, json: true }, (error, result) => {
9       if (error) {
10         callback("Unable to connect weather services", undefined);
11       } else if (result.body.error) {
12         callback("Unable to find location. Try another search", undefined);
13       } else {
14         const temperature = result.body.current.temperature;
15         const feelslike = result.body.current.feelslike;
16         callback(undefined, {
17           temperature,
18           feelslike,
19         });
20       }
21     });
22   } catch (err) {
23     callback(err, undefined);
24   }
25 };
26
27 module.exports = forecast;
28
```

Sonrasında ise bu fonksiyonları index.js dosyamıza import edelim.

```
src > JS index.js > ...
1  const hbs = require("hbs");
2  const path = require("path");
3  const express = require("express");
4  const app = express();
5
6  const geocode = require("./utils/geocode");
7  const forecast = require("./utils/forecast");
8
```

```
24 app.get("/weather", (req, res) => {
25   const { address } = req.query;
26
27   if (!address) {
28     return res.send({ error: "You must provide an address" });
29   }
30
31   geocode(address, (error, { longitude, latitude, location } = {}) => {
32     if (error) {
33       return res.status(404).json({ error });
34     }
35
36     forecast(longitude, latitude, (err, result) => {
37       if (err) {
38         return res.status(404).json({ error: err });
39       }
40
41       return res.status(200).json({ location, ...result });
42     });
43   });
44 });
45
```

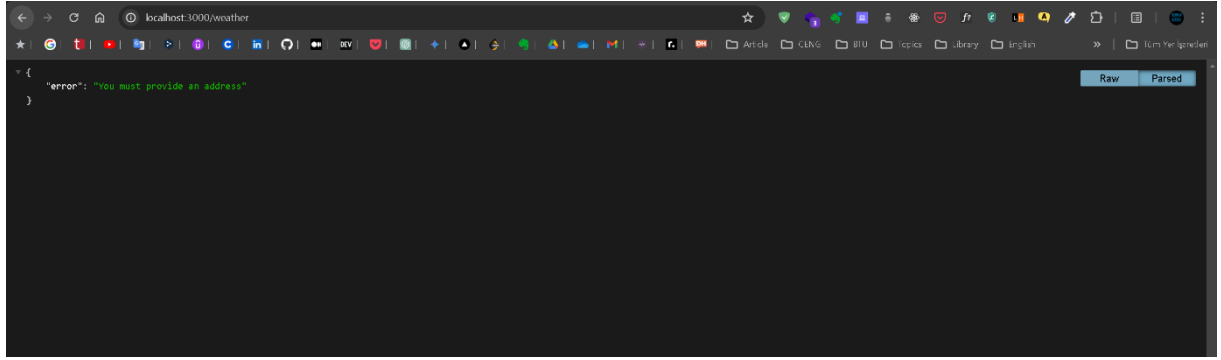
Yukarıdaki kod parçası, **/weather** yoluna gelen **GET** isteklerini işleyen bir endpoint tanımlar. İstekten gelen query string parametrelerini (**address**), **req.query** aracılığıyla alır. Ardından, bu parametrelerin varlığını kontrol eder. Eğer **address** parametresi yoksa, yetersiz istek durumunu ifade eden bir yanıt gönderir ve işlemi sonlandırır.

Eğer **address** parametresi varsa, bu adresle ilgili olarak hava durumu bilgilerini almak üzere **geocode** fonksiyonunu çağırır. **geocode** fonksiyonu, verilen adresle ilişkili enlem ve boylam koordinatlarını çözer. Bu koordinatlar, ardından **forecast** fonksiyonuna geçirilir.

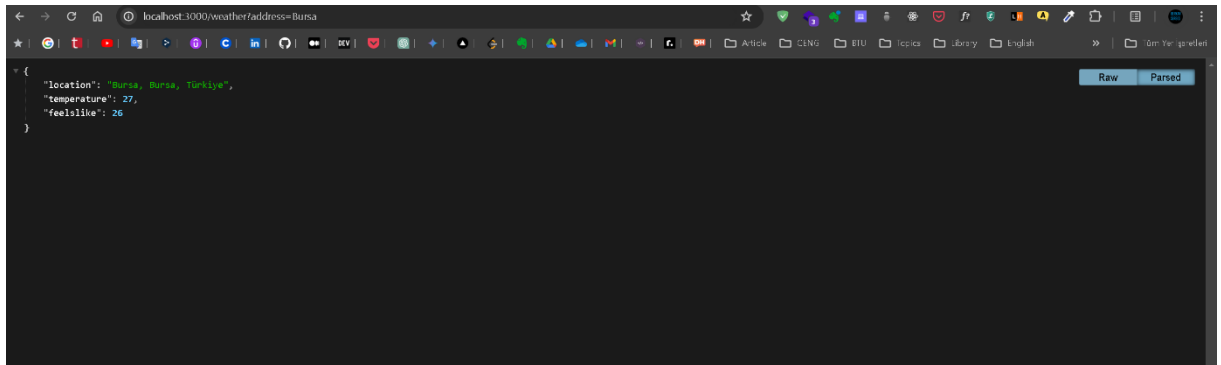
**forecast** fonksiyonu, belirtilen enlem ve boylam koordinatlarına dayanarak hava durumu bilgilerini getirir. Bu bilgiler, hava durumu servisinden alınan verilerdir ve sıcaklık, hava durumu özeti gibi bilgiler içerir. Eğer hava durumu bilgileri başarıyla alınırsa, bu bilgileri bir yanıt olarak gönderir.

Ancak, herhangi bir hata oluşması durumunda, **geocode** veya **forecast** fonksiyonlarından herhangi biri bir hata döndürür. Bu durumda, hatayı içeren bir yanıt gönderilir ve işlem sonlandırılır. Hataya, uygun bir HTTP durum kodu atanır (örneğin, 404 Not Found). Bu şekilde, istemciye doğru hata mesajıyla birlikte yanıt verilmiş olur.

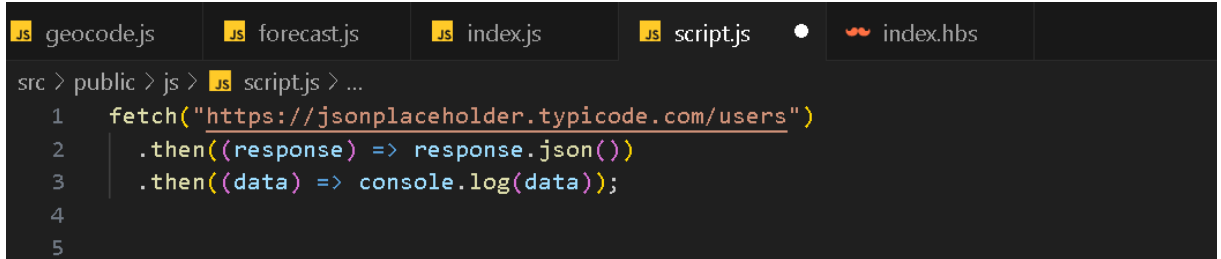
Eğer herhangi bir query parametresi göndermezsek aşağıdaki çıktıyı alacağız.



Doğru bir şekilde query string parametrelerini gönderdiğimiz takdirde aşağıdaki görseldeki yanıtı alacağız.



Şimdi, dış bir API'ye GET isteği göndererek veri almak için JavaScript'in fetch fonksiyonunu kullanacağız. Public klasörümüzün altına **js** adında bir klasör oluşturup içerisine **script.js** adında bir dosya oluşturalım ve içeriğini aşağıdaki görseldeki gibi yazalım.



Yukarıdaki kodda, öncelikle **fetch** fonksiyonu aracılığıyla hedef API'nin URL'sine istek gönderiyoruz (**https://jsonplaceholder.typicode.com/users**). Bu istek, bir **Promise** döndürür. Dönen Promise, sunucudan bir yanıt alındığında veya hata oluştuğunda çözülür.

İlk **.then** bloğu, sunucudan gelen yanıtı alır ve bu yanıtı JSON formatına dönüştürmek için **response.json()** metodunu kullanır. Bu işlem, gelen veriyi JavaScript nesnelere dönüştürür ve bir sonraki **.then** bloğuna geçirir.

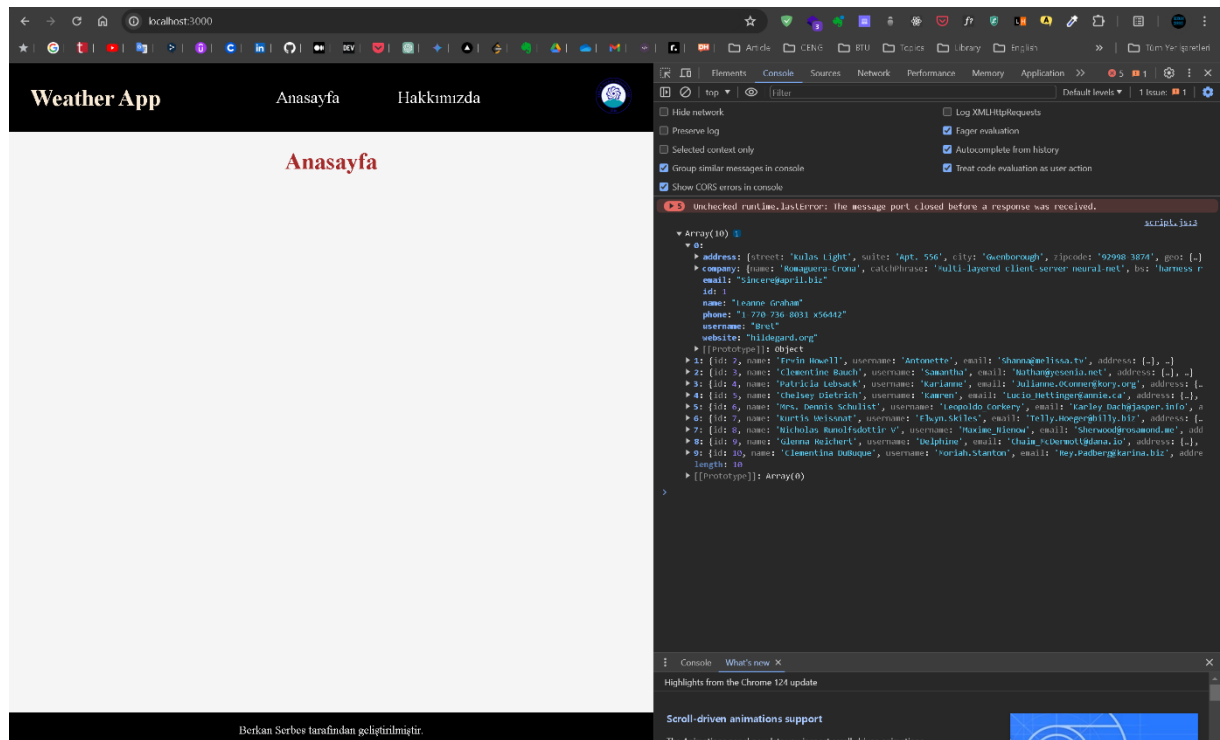
İkinci **.then** bloğu, dönüştürülmüş veriyi alır ve bu veriyi konsola yazdırmak için **console.log** fonksiyonunu kullanır. Böylelikle, dış API'den alınan veriyi konsola basarak işlemi tamamlarız.

Bu yöntem, web uygulamalarında dış API'lerle etkileşimde bulunmak için yaygın olarak kullanılır.

Sonrasında bu fonksiyonu kullanmamız için index.hbs dosyamıza bu javascript dosyasını tanımlamamız gerekmekte aşağıdaki şekilde nasıl tanımlanacağı 12.satırda gösterilmiştir.

```
src > public > views > index.hbs > html > body > script
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Anasayfa</title>
7 </head>
8 <body>
9   {{>header}}
10  <h1>Anasayfa</h1>
11  {{>footer}}
12  <script src=" ../js/script.js"></script>
13 </body>
14 </html>
```

Uygulamamızı çalıştırıp developer tools'dan konsol kısmına gidersek çektiğimiz veriyi göreceğiz.



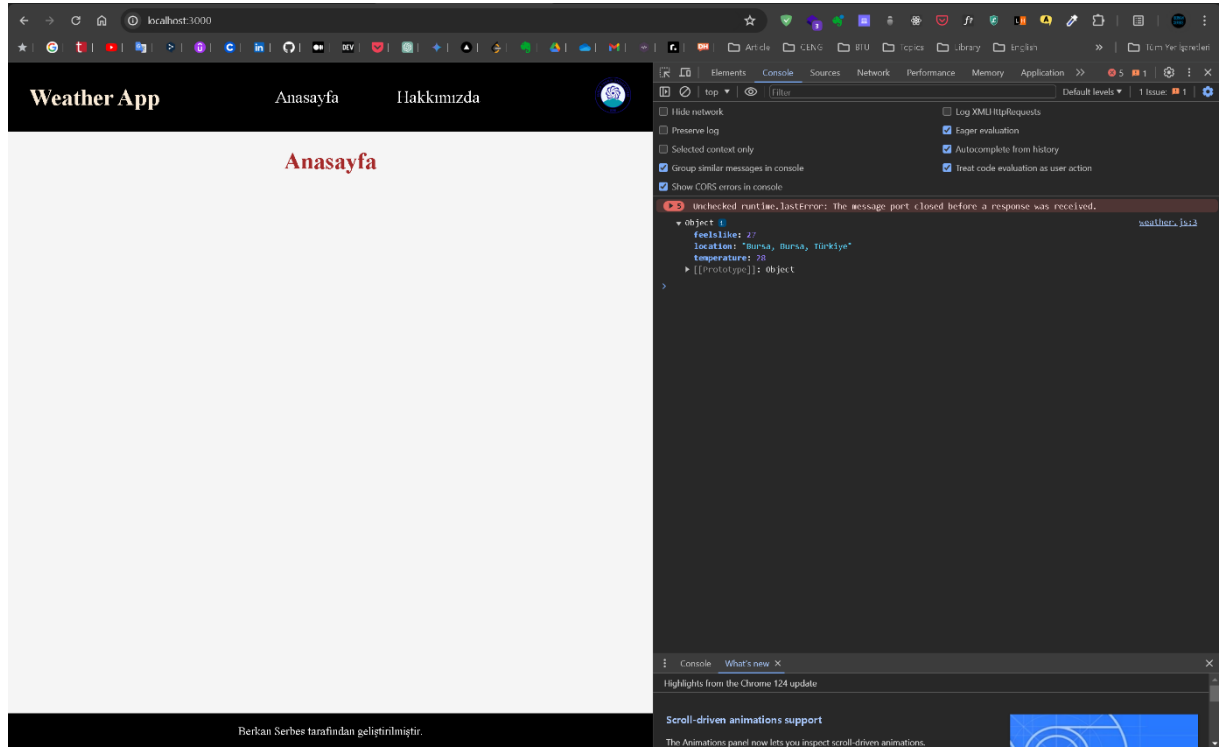
### 1.4.3 /weather Endpointinden Veri Çekme İşlemi

Bu bölümde daha önceki adımlarda Express ile oluşturduğumuz **'weather'** endpointinden veri çekeceğiz. İlk olarak bu işlemi konsol üzerinden yapacağız, ardından bu veriyi kullanıcı ile etkileşime girerek bir form üzerinden dinamik bir şekilde istek atacağız.

**weather.js** adında bir dosya oluşturalım ve içeriğini aşağıdaki görseldeki gibi yazalım. Sonrasında bu javascript dosyasını index.hbs dosyamıza script dosyası olarak ekleyelim.

```
1 fetch("http://localhost:3000/weather?address=Bursa")
2   .then((response) => response.json())
3   .then((data) => console.log(data));
4
5
```

Yukarıdaki işlemleri başarılı bir şekilde yaptığımızda aşağıdaki çıktıyı alacağız konsolda.



Şimdi de **index.hbs** sayfamızda bir form oluşturalım. Aşağıdaki şekildeki görsellerde bu sayfanın html görünümü ve css görünümü verilmiştir.

```
src > public > views > index.hbs > html > body > div.container
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <meta charset="UTF-8">
5    <meta name="viewport" content="width=device-width, initial-scale=1.0">
6    <title>Anasayfa</title>
7    <link rel="stylesheet" href=" ../css/index.css">
8  </head>
9  <body>
10   {{>header}}
11   <h1>Anasayfa</h1>
12   <div class="container">
13     <form class="form">
14       <input type="text" name="address" id="location" class="input" placeholder="Enter location">
15       <button type="submit">Gönder</button>
16     </form>
17     <div class="result-container">
18       <p id="message-1"></p>
19       <p id="message-2"></p>
20     </div>
21   </div>
22   {{>footer}}
23   <script src=" ../js/weather.js"></script>
24 </body>
25 </html>
```

```
src > public > css > index.css > button
1  .form {
2    display: flex;
3    justify-content: center;
4    gap: 10px;
5  }
6
7  .input {
8    border: 1px solid #ccc;
9    padding: 8px;
10 }
11
12 .result-container {
13   display: flex;
14   align-items: center;
15   flex-direction: column;
16 }
17
18 button {
19   padding: 8px 16px;
20   background-color: #007bff;
21   color: white;
22   border: none;
23   cursor: pointer;
24 }
25
26 button:hover {
27   cursor: pointer;
28 }
29
```



weather.js adlı dosyanın içeriği:

```
src > public > js > weather.js > ...
1  // fetch("http://localhost:3000/weather?address=Bursa")
2  // .then((response) => response.json())
3  // .then((data) => console.log(data));
4
5  const weatherForm = document.querySelector(".form");
6  const message1 = document.querySelector("#message-1");
7  const message2 = document.querySelector("#message-2");
8  const input = document.querySelector(".input");
9  const button = document.querySelector("button");
10
11  weatherForm.addEventListener("submit", (e) => {
12    e.preventDefault();
13
14    const location = input.value;
15
16    console.log(location);
17    message1.textContent = "Loading...";
18    message2.textContent = "";
19
20    fetch(`http://localhost:3000/weather?address=${location}`)
21      .then((response) => response.json())
22      .then((data) => {
23        if (data.error) {
24          message1.textContent = data.error;
25          message2.textContent = "";
26        } else {
27          message1.textContent = `Location: ${data.location}`;
28          message2.textContent = `Temperature: ${data.temperature}, Feels like: ${data.feelslike}`;
29        }
30      });
31  });
32
```

Bu JavaScript kodu, bir HTML formunu işlemek ve sunucudan hava durumu verisi almak için kullanılır.

- weatherForm, HTML'de .form sınıfına sahip bir form elementini seçer.
- message1 ve message2, HTML'de #message-1 ve #message-2 id'lerine sahip iki metin elementini seçer.
- input, HTML'de .input sınıfına sahip bir input elementini seçer.
- button, HTML'de <button> elementini seçer.

weatherForma bir **'submit'** olay dinleyicisi eklenir. Bu, form gönderildiğinde çalışacak işlevi tanımlar. İşlev, öntanımlı davranışı engellemek için **e.preventDefault()** ile başlar, böylece sayfanın yeniden yüklenmesi engellenir.

Daha sonra, kullanıcının girdiği konumu almak için **input.value** kullanılır ve bu değer location değişkenine atanır. Ardından, message1 içeriği "Yükleniyor..." olarak değiştirilir ve message2 içeriği boş stringe çevrilir.

Sonra, fetch fonksiyonu kullanılarak sunucudan hava durumu verisi alınır. Sunucu, /weather endpoint'ine kullanıcının girdiği konumu bir query string olarak iletir. Gelen yanıt response.json() ile JSON formatına dönüştürülür ve ardından ikinci bir then bloğunda işlenir.

Eğer sunucudan bir hata mesajı dönerse, data.error kontrol edilir ve eğer varsa message1 içeriği bu hatayla değiştirilir, message2 içeriği boşaltılır. Aksi takdirde, hava durumu verileri data.location, data.temperature ve data.feelslike ile message1 ve message2 içeriğine eklenir.

Uygulamayı çalıştırdığımızda form kısmına geçersiz bir veri girdiğimizde aşağıdaki çıktıyı alıyoruz.

Weather App

AnasayfaHakkımızda

### Anasayfa

Gönder

Unable to find location. Try another search

Berkan Serbes tarafından geliştirilmiştir.

Eğer geçerli bir konum girdiğimizde cevapları dinamik olarak aşağıdaki gibi alıyoruz.

Weather App

AnasayfaHakkımızda

### Anasayfa

Gönder

Location: Bursa, Bursa, Türkiye

Temperature: 29, Feels like: 28

Berkan Serbes tarafından geliştirilmiştir.

## 2. Kaynakça

<https://chat.openai.com/>