

BURSA TEKNİK ÜNİVERSİTESİ

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



BLM0470 NodeJS İle Web Programlama

2023-2024 Bahar Dönemi

14.Hafta Raporu

Berkan SERBES – 22360859353

İçindekiler

1.API Testleri.....	3
1.1 Postman	3
1.1.1 Postman Kurulumu.....	3
2.Görev Yönetim Uygulaması Geliştirme.....	4
2.1 Projenin Oluşturulması.....	5
2.2 NPM Bağımlılıklarının Yüklenmesi.....	5
2.3 Yazılım Geliştirme Ortamları	6
2.3.1 Production (Üretim) Ortamı	7
2.3.2 Development (Geliştirme) Ortamı	7
2.3.3 Test Ortamı	7
2.4 Scriptlerin Oluşturulması	7
2.5 Dosya Yapısının Oluşturulması	8
2.6 Model Klasörünün Oluşturulması	10
2.7 Controller Klasörünün Oluşturulması	11
2.8 Routes Klasörünün Oluşturulması	17
2.9 Ana Dosyanın Kodlanması.....	18
2.10 Uygulamanın Postman Üzerinden Test Edilmesi	18
3. Kaynakça	23

1.API Testleri

API testleri, yazılımın Application Programming Interface (API) katmanını doğrulamak için yapılan testlerdir. Bu testler, API'lerin işlevselliğini, güvenilirliğini ve performansını kontrol eder. API testleri, genellikle doğrudan API'lerin son noktalarına (endpoints) istekler göndererek yapılır ve bu isteklerin doğru yanıtlar verip vermediğini kontrol eder.

API testleri yapılırken çeşitli araçlar kullanılır. Kullanılan araçlar arasında Postman, kolay bir kullanıcı arayüzü sağlayarak API istekleri oluşturmayı ve yanıtları analiz etmeyi sağlar. Insomnia, Postman'e benzer şekilde API testlerini yapmayı kolaylaştıran bir başka popüler araçtır. SoapUI, daha kapsamlı özellikler sunarak SOAP ve RESTful API'leri test etmek için kullanılır. Ayrıca, Newman, Postman koleksiyonlarının komut satırı üzerinden çalıştırılmasını sağlayan bir araçtır ve CI/CD süreçlerine entegre edilebilir. Bu araçlar, API'lerin doğru çalışmasını ve beklenen sonuçları vermesini sağlamak için kritik öneme sahiptir.

Bizim yapacağımız uygulamada kullanacağımız araç ise Postman olacaktır.

1.1 Postman

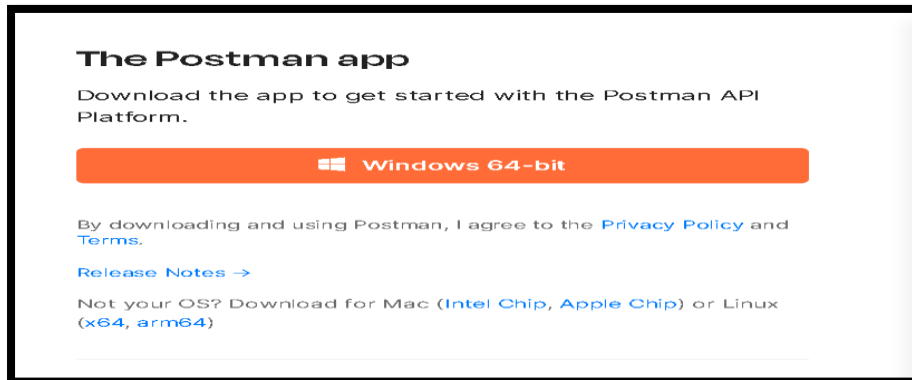
Postman, API testleri için yaygın olarak kullanılan bir araçtır. Kullanıcı dostu arayüzü sayesinde, API istekleri oluşturmayı, gönderilen istekleri ve alınan yanıtları görmeyi kolaylaştırır. Postman, GET, POST, PUT, DELETE gibi çeşitli HTTP isteklerini destekler ve kullanıcıların istek gövdeleri, başlıklar ve parametreler gibi detayları kolayca yapılandırmasına olanak tanır. Ayrıca, kullanıcılar API yanıtlarını doğrulamak için testler yazabilir ve bu testleri otomatik olarak çalıştırabilirler.

Postman'ın koleksiyonlar özelliği, bir dizi API isteğini gruplayarak düzenlemeyi ve paylaşmayı sağlar, bu da ekipler arasında işbirliğini kolaylaştırır. Koleksiyonlar, Postman'ın Newman adı verilen komut satırı aracıyla çalıştırılabilir, böylece API testleri CI/CD süreçlerine entegre edilebilir. Ayrıca, Postman, API belgelerini otomatik olarak oluşturma yeteneği ile de geliştiricilere büyük kolaylık sağlar. Bu özellikler, Postman'ı hem manuel hem de otomatik API testleri için güçlü bir araç haline getirir.

1.1.1 Postman Kurulumu

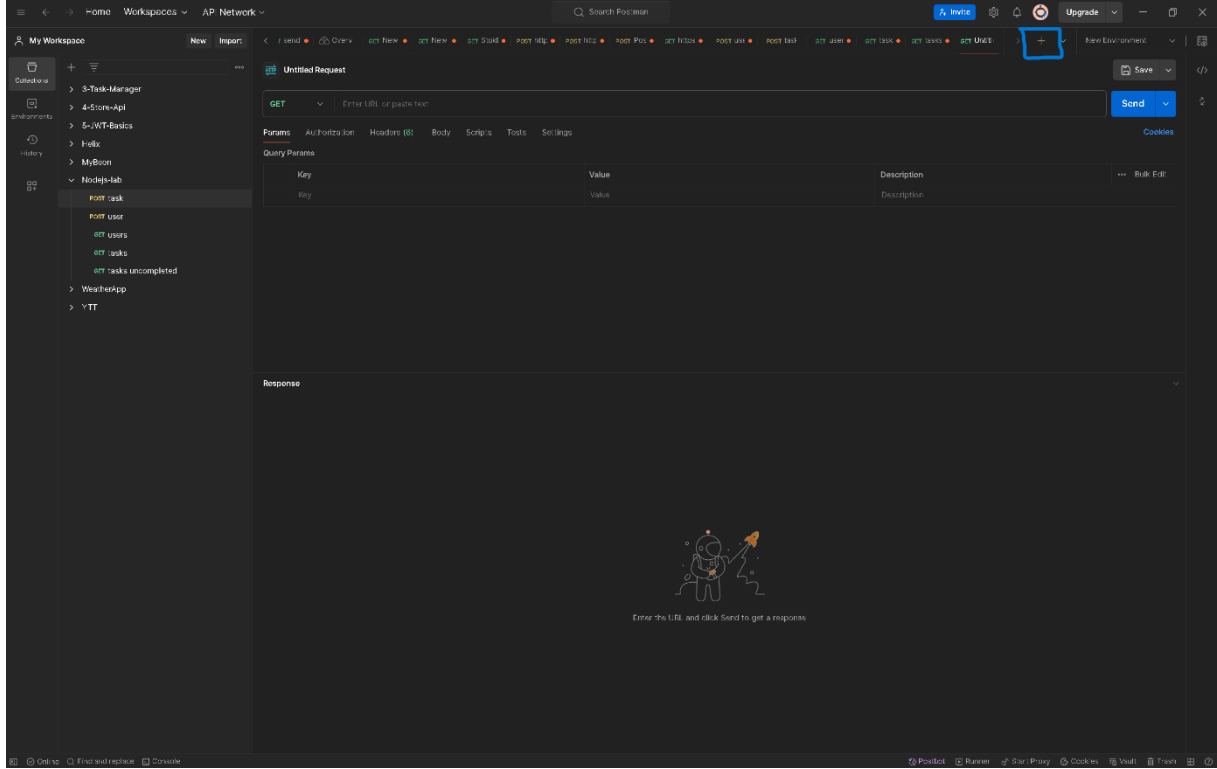
Postmanı bilgisayarınıza kurmak için izlemeniz gereken adımlar şu şekildedir:

- 1) Postman web sitesine [bağlantıya](#) tıklayarak gidin.
- 2) Aşağıda görselde yer alan Windows 64-bit butonuna tıklayın. Bu sizin sistem özelliklerinize ve işletim sisteminize göre farklı olabilir.



- 3) İndirdiğiniz “.exe” dosyasını çift tıklayarak çalıştırın.
- 4) Kurulum sihirbazını takip edin. Postman, gerekli dosyaları yükleyecek ve kurulumu tamamlayacaktır.

Postmanı başarılı bir şekilde kurduktan sonra uygulamayı açtığımızda şu şekilde bir arayüz bizi karşılayacaktır. Ben daha önceden yüklediğim için ekstradan çeşitli koleksiyonlar ve istek sekmeleri bulunmaktadır.



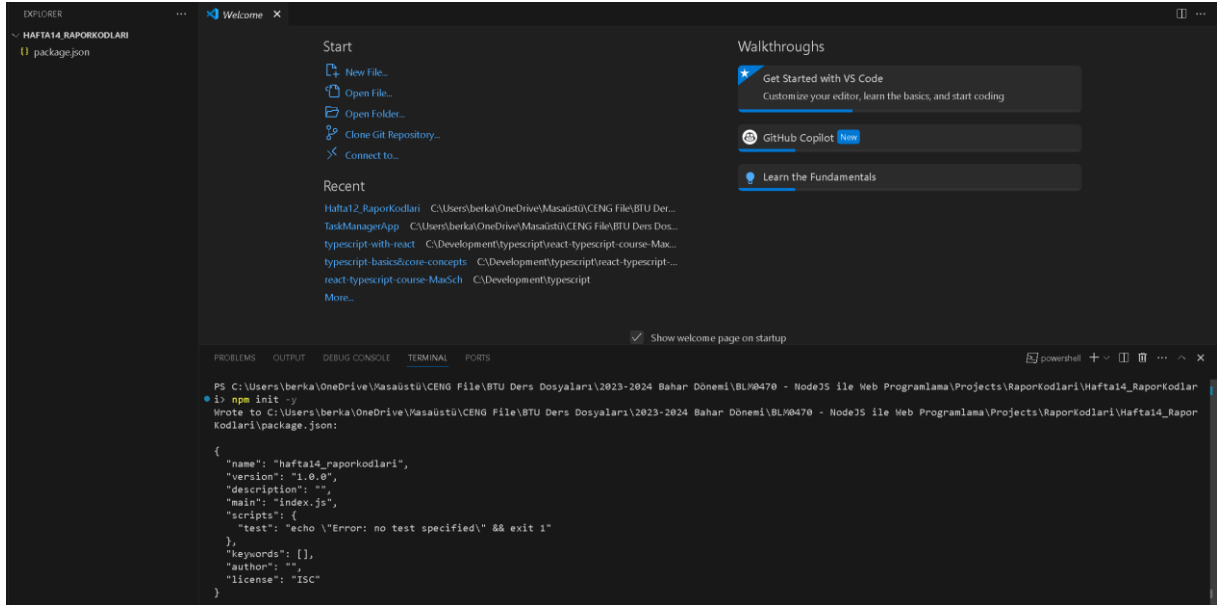
Yukarıdaki görselde yer alan mavi kısımla işaretlenen bölgede ‘+’ butonu bulunmaktadır. Bu buton bize yeni bir istek sekmesi açacaktır.

2.Görev Yönetim Uygulaması Geliştirme

Bu kısımda daha önceki raporda yaptığımız hard-coding biçiminde kayıt ekleme işlemlerini bir API oluşturarak yapacağız. Bu API sayesinde kullanıcılar ve görevler ile ilgili işlemleri daha dinamik ve esnek bir şekilde gerçekleştirebileceğiz. Kullanıcılar artık API üzerinden yeni görevler ekleyebilecek, mevcut görevleri güncelleyebilecek ve tamamlanmış görevleri listeleyebilecekler. Bu, uygulamanın kullanıcı etkileşimini artıracak ve veritabanı işlemlerini daha güvenilir hale getirecektir.

2.1 Projenin Oluřturulması

İlk olarak **npm init -y** komutunu kullanarak Node.js projemizi oluřturalım.



2.2 NPM Bağımlılıklarının Yüklenmesi

Bu uygulamada kullanacağımız NPM paketleri şunlardır:

express, mongoose, dotenv, validator ve nodemon.

Express: Node.js için minimal ve esnek bir web uygulama çatısıdır. HTTP isteklerini işleme, yönlendirme ve middleware kullanımı gibi temel web sunucu işlevlerini kolayca gerçekleştirmenizi sağlar. Hem küçük hem de büyük ölçekli web uygulamaları için idealdir.

Mongoose: Node.js ortamında MongoDB ile çalışmayı kolaylaştıran **bir ODM (Object Data Modeling)** kütüphanesidir. MongoDB belgelerini şemalarla modellemenize, doğrulamanıza ve iş mantığını tanımlamanıza olanak tanır. Mongoose, veritabanı işlemlerini basitleştirir ve MongoDB ile çalışırken TypeScript benzeri tip güvenliği sağlar.

Dotenv: Node.js projelerinde çevresel değişkenlerin yönetimini kolaylaştıran bir pakettir. Bu paket, **.env** dosyasında tanımlanan çevresel değişkenleri okuyarak **process.env** içine yükler. Bu sayede, hassas verileri ve konfigürasyon ayarlarını kod tabanından ayrı tutarak güvenli bir şekilde yönetebilirsiniz.

Validator: Kullanıcı girdilerini doğrulamak ve temizlemek için kullanılan bir kütüphanedir. E-posta adresleri, URL'ler, sayılar ve diğer veri türleri için doğrulama ve temizleme fonksiyonları sağlar. Bu kütüphane, kullanıcı girişlerini güvenli ve geçerli tutmak için yaygın olarak kullanılır.

Nodemon: Node.js uygulamalarını geliştirirken sıkça kullanılan bir araçtır. Bu araç, kodunuzda yapılan değişiklikleri otomatik olarak algılar ve uygulamanızı yeniden başlatır. Bu sayede, geliştirme sürecinde manuel yeniden başlatma işlemi yapmanıza gerek kalmaz ve zamandan tasarruf edersiniz.

Bu paketler, uygulamamızın işlevselliğini artıracak ve geliştirme sürecinizi daha verimli hale getirecektir.

npm install komutu yardımıyla ilgili npm paketlerini yükleyelim. Nodemon paketini ayrı olarak farklı bir parametre ile yükleyeceğiz.

```
PS C:\Users\berka\OneDrive\Masaüstü\CENG File\BTU Ders Dosyaları\2023-2024 Bahar Dönemi\BLM0470 - NodeJS ile Web Programlama\Projects\RaporKodlari\Hafta14_RaporKodlar
i> npm install express mongoose dotenv validator

added 86 packages, and audited 87 packages in 16s

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

found 0 vulnerabilities
```

```
{ package.json X
package.json > ...
1  {
2    "name": "hafta14_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     "mongoose": "^8.4.0",
16     "validator": "^13.12.0"
17   },
18   "devDependencies": {
19     "nodemon": "^3.1.1"
20   }
21 }
22

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\berka\OneDrive\Masaüstü\CENG File\BTU Ders Dosyaları\2023-2024 Bahar Dönemi\BLM0470 - NodeJS ile Web Programlama\Projects\Ra
i> npm install nodemon --save-dev

added 29 packages, and audited 116 packages in 3s

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

found 0 vulnerabilities
```

Yukarıdaki görselde Nodemon paketini **--save-dev** -alternatif olarak **-D** parametresi de kullanılabilir- parametresiyle **devDependencies (geliştirme bağımlılıkları)** olarak yükledik çünkü bu paket yalnızca geliştirme aşamasında kullanılır ve üretim ortamında (production) gerekli değildir. Üretim ortamında, uygulamanın otomatik olarak yeniden başlatılması gerekmeyeceğinden, Nodemon'un yüklü olması gereksizdir.

2.3 Yazılım Geliştirme Ortamları

Yazılım geliştirme sürecinde, yazılım uygulamalarının farklı aşamalarda çalıştırıldığı ve test edildiği çeşitli ortamlar bulunmaktadır. Bu ortamlar, uygulamanın geliştirilmesi, test edilmesi ve nihai olarak kullanıcıya sunulması için farklı gereksinimleri karşılamak üzere yapılandırılmıştır. En yaygın ortamlar şunlardır: **Production (Üretim)**, **Development (Geliştirme)** ve **Test (Deneme)**. Her biri belirli amaçlar ve işlevler için kullanılır.

2.3.1 Production (Üretim) Ortamı

Production ortamı, yazılımın nihai kullanıcıya sunulduğu, canlı (live) ortamdır. Bu ortam, yazılımın gerçek kullanıcılar tarafından kullanıldığı yerdir ve dolayısıyla en yüksek güvenlik, performans ve güvenilirlik gereksinimlerine sahiptir. Production ortamında yapılan herhangi bir değişiklik, doğrudan kullanıcı deneyimini etkiler. Bu nedenle, değişiklikler dikkatli bir şekilde yönetilmeli ve kapsamlı bir şekilde test edilmelidir.

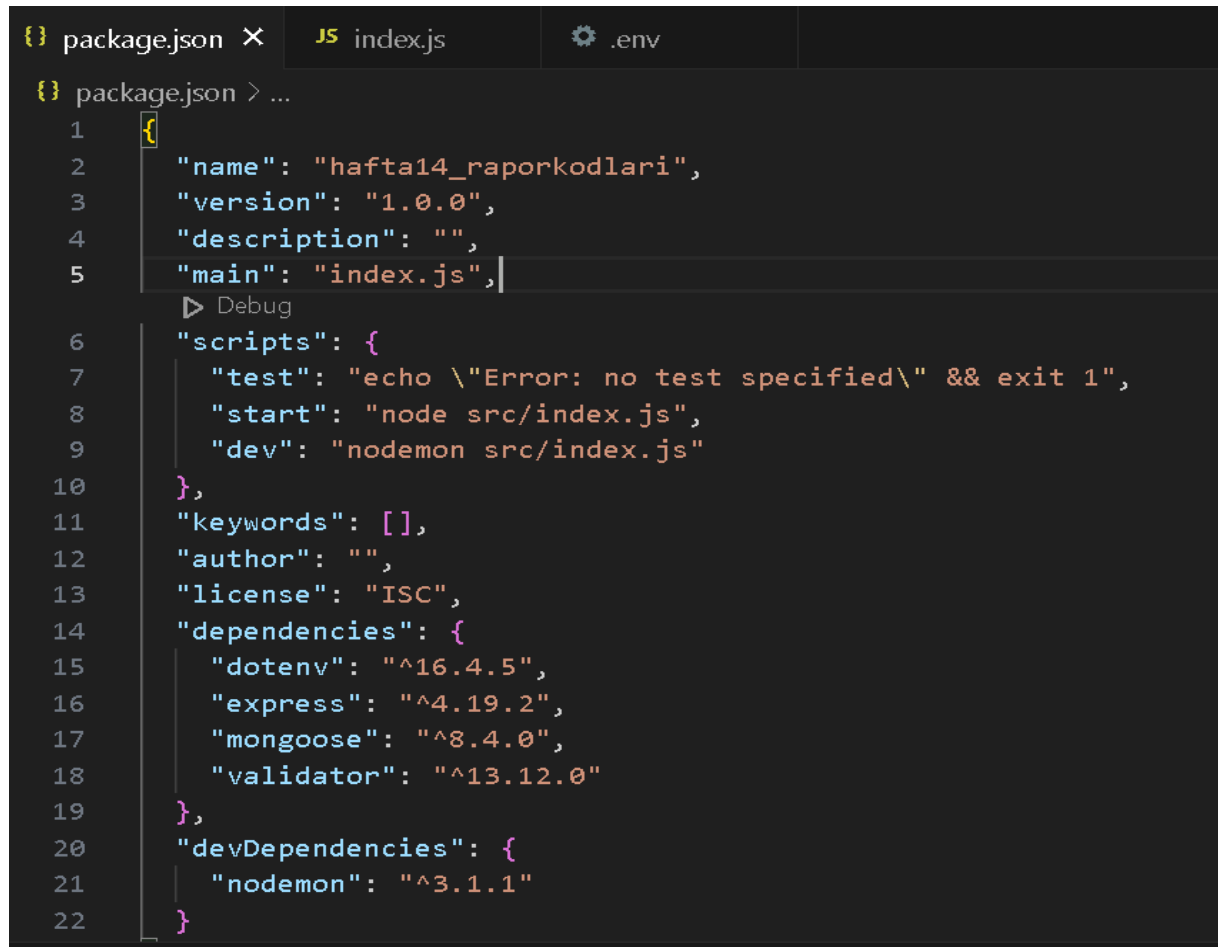
2.3.2 Development (Geliştirme) Ortamı

Development ortamı, yazılım geliştiricilerin yeni özellikler eklediği, hataları düzelttiği ve genel olarak uygulamayı geliştirdiği ortamdır. Bu ortam, geliştiricilerin uygulamanın kaynak kodunu değiştirmesine ve yeni kod yazmasına olanak tanır. Development ortamında, yazılımın çalışması için gerekli tüm araçlar, kütüphaneler ve veritabanları mevcuttur.

2.3.3 Test Ortamı

Test ortamı, yazılımın farklı yönlerinin (fonksiyonellik, performans, güvenlik vb.) test edildiği ortamdır. Bu ortam, geliştirme ve üretim ortamları arasındaki bir köprü görevi görür ve yazılımın üretime geçmeden önceki son testlerinin yapıldığı yerdir. Test ortamı, uygulamanın kullanıcıya sunulmadan önce tüm potansiyel sorunların tespit edilmesini ve çözülmesini sağlar.

2.4 Scriptlerin Oluşturulması



```
{ package.json X JS index.js .env
package.json > ...
1 {
2   "name": "hafta14_raporkodlari",
3   "version": "1.0.0",
4   "description": "",
5   "main": "index.js",
6   "scripts": {
7     "test": "echo \"Error: no test specified\" && exit 1",
8     "start": "node src/index.js",
9     "dev": "nodemon src/index.js"
10  },
11  "keywords": [],
12  "author": "",
13  "license": "ISC",
14  "dependencies": {
15    "dotenv": "^16.4.5",
16    "express": "^4.19.2",
17    "mongoose": "^8.4.0",
18    "validator": "^13.12.0"
19  },
20  "devDependencies": {
21    "nodemon": "^3.1.1"
22  }
```

Bu package.json dosyasındaki **"scripts"** bölümü, projenin farklı senaryoları için çalıştırılabilir komutların tanımlandığı bir bölümdür. Bu senaryolar genellikle geliştirme, test veya başlatma gibi farklı amaçlar için kullanılır. İşte bu örnekteki "scripts" bölümünde tanımlı olan komutlar:

1. **"test"**: Bu komut, test senaryosunu çalıştırmak için kullanılır. Ancak, bu örnekte varsayılan olarak bir test senaryosu tanımlanmamıştır. Dolayısıyla, "Error: no test specified" mesajıyla birlikte hata kodu 1 döndürülür.

2. **"start"**: Bu komut, uygulamanın normal çalışma modunda başlatılmasını sağlar.

3. **"dev"**: Bu komut, geliştirme sırasında uygulamanın başlatılmasını sağlar. "nodemon src/index.js" komutu, Nodemon aracılığıyla uygulamanın başlatılmasını ve kod değişikliklerinin algılanarak otomatik olarak yeniden başlatılmasını sağlar. Bu şekilde, geliştirme süreci daha verimli hale gelir ve kod değişikliklerinin hemen etkisini görebilirsiniz.

Bu script'ler, **npm run** komutuyla birlikte kullanılabilir. Örneğin, **npm run start** komutuyla uygulamanızı başlatabilir veya **npm run dev** komutuyla geliştirme sırasında otomatik yeniden başlatma özelliğini kullanabilirsiniz. Bu, geliştirme sürecini kolaylaştıran ve hızlandıran yaygın bir uygulamadır.

2.5 Dosya Yapısının Oluşturulması

İlk adım olarak, projenin ana dizininde, kodumuzda görünmesini istemediğimiz bilgileri saklamak için bir **.env** dosyası oluşturmamız gerekmektedir. Bu dosyada **PORT** ve **MONGODB_URL** gibi değişkenler bulunmaktadır. PORT değişkeni, uygulamanın hangi port numarasında çalışacağını belirtirken, MONGODB_URL değişkeni, MongoDB Atlas bağlantı URL'sini içerir. Bu bilgiler, uygulamanın doğru şekilde çalışabilmesi için gereklidir.

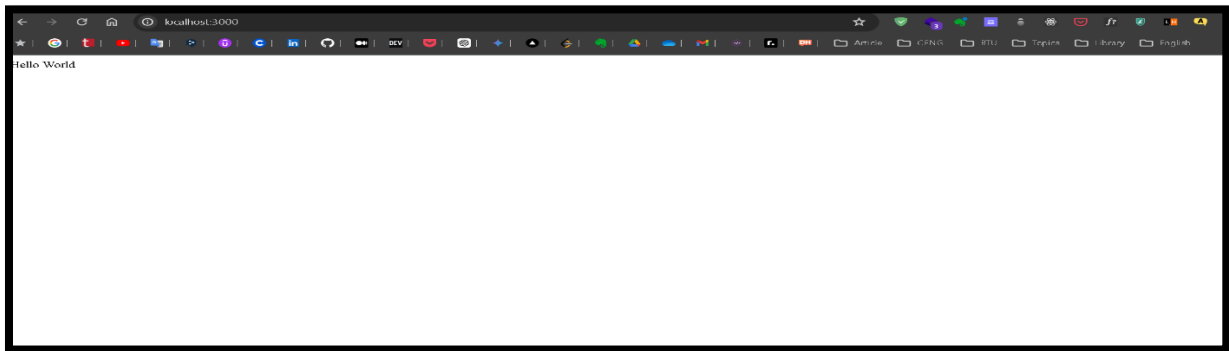
```
.env
1  PORT=3000
2  CONNECTION_URL="mongodb+srv://berkanserbes:berkanserbes@my-cluster.mongodb.net/?retryWrites=true&w=majority&appName=task-m:
3
4
5
6
7
```

.env dosyasını oluşturduktan sonra, projenin ana dizinine **src** adında bir klasör ekleyelim. Bu klasör, projemizin kaynak kodlarını içerecek ve uygulamamızın ana çalışma dosyasını barındıracaktır. Bu dosyaya **index.js** adını vereceğiz. Bu dosya, uygulamamızın başlangıç noktası olacak ve diğer dosyaları ve modülleri buradan çağıracaktır.

index.js dosyasının içeriği şimdilik aşağıdaki görseldeki gibidir.


```
src > JS index.js > ...
1  require("dotenv").config();
2  const express = require('express');
3  const app = express();
4
5  const PORT = process.env.PORT || 3000;
6
7  app.get('/', (req, res) => {
8    res.send('Hello World');
9  });
10
11 app.listen(PORT, () => {
12   console.log(`Server is listening on port ${PORT}`)
13 })
```

npm run dev komutunu çalıştırıp tarayıcımızdan **localhost:3000** adresine gittiğimizde aşağıdaki çıktıyı alacağız.



Şimdi ise veritabanına bağlanmak için bir fonksiyon yazmamız gerekmekte. Bu fonksiyon, projenin başlangıcında çağrılacak ve veritabanıyla iletişim kuracak. İlk olarak, **“.env”** dosyasından bağlantı URL’imizi alıyoruz. Bağlantıyı gerçekleştirmek için **connect** adlı bir fonksiyon oluşturuyoruz. Bu fonksiyon, **mongoose.connect** metodu kullanılarak MongoDB veritabanına bağlanıyor. Bağlantı başarılı olduğunda **"Connected to the database"** mesajını konsola yazdırıyor. Bağlantı sırasında herhangi bir hata oluşursa, hata mesajını konsola yazdırıyoruz. Son olarak, oluşturduğumuz fonksiyonu projenin dışına ihraç ediyoruz, böylece diğer dosyalar da bu fonksiyonu kullanabilir.

Fonksiyonumuzun doğru çalışıp çalışmadığını test etmek için bu fonksiyonu çağırıp terminal üzerinden bu dosyayı çalıştırıyoruz.

Aşağıdaki görselde görüldüğü üzere veritabanına başarılı bir şekilde bağlandık. Eğer başarılı bir şekilde bağlandıysak fonksiyon çağrımını siliyoruz çünkü bu fonksiyonu ana dosyamızda (index.js) çağıracağız.

```
src > db > JS config.js > CONNECTION_STRING
1 require("dotenv").config();
2 const mongoose = require("mongoose");
3
4 const CONNECTION_STRING = process.env.CONNECTION_URL;
5
6 const connect = async () => {
7   try{
8     mongoose.connect(CONNECTION_STRING).then(() => {
9       console.log('Connected to the database');
10    }).catch(err => {
11      console.log(err);
12    });
13   }
14   catch(err) {
15     console.log(err);
16   }
17 }
18
19 connect();
20
21 module.exports = {connect};
```

```
PS C:\Users\berka\OneDrive\Masaüstü\CENG File\BTU Ders Dosyaları\2023-2024 Bahar Dönemi\BLM0470 - NodeJS ile Web Programlama\Projects\RaporKodlari\Hafta14_RaporKodlar
i> node .\src\db\config.js
Connected to the database
```

2.6 Model Klasörünün Oluşturulması

Projemizde, kullanıcı ve görevlerle ilgili bilgileri içerecek olan iki farklı model bulunacaktır. Kullanıcı modeli, kullanıcıya ait detayları, görev modeli ise görevlerin özelliklerini içerecektir.

User modelimizin içeriği aşağıdaki görseldeki gibi olacaktır.

```
src > models > JS Users.js > UserSchema > age
1 const mongoose = require("mongoose");
2 const { Schema } = mongoose;
3 const validator = require("validator");
4
5 const UserSchema = new Schema({
6   name: {
7     type: String,
8     required: true,
9     trim: true
10  },
11  age: {
12    type: Number,
13    required: false,
14    default: 0,
15    validate(value) {
16      if(value < 0) {
17        throw new Error("Age must be a positive number");
18      }
19    }
20  },
21  email: {
22    type: String,
23    required: true,
24    trim: true,
25    lowercase: true,
26    validate(value) {
27      if(!validator.isEmail(value)) {
28        throw new Error("Email is invalid");
29      }
30    }
31  },
32  telephone: {
33    type: String,
34    required: false,
35    validate(value) {
36      if(!validator.isMobilePhone(value)) {
37        throw new Error("Phone number is invalid");
38      }
39    }
40  },
41 });
42
43 module.exports = mongoose.model("User", UserSchema)
```

Task modelimizin içeriği aşağıdaki görseldeki gibi olacaktır.

```
src > models > JS Task.js > ...
1  const mongoose = require("mongoose");
2  const { Schema } = mongoose;
3
4  const TaskSchema = new Schema({
5    description: {
6      type: String,
7      required: true,
8      trim: true
9    },
10   completed: {
11     type: Boolean,
12     required: false,
13     default: false
14   }
15 });
16
17 module.exports = mongoose.model("Task", TaskSchema);
```

2.7 Controller Klasörünün Oluşturulması

Uygulamamızın ana iş mantığını yönetmek için bir controller klasörü oluşturacağız. Bu klasörün içerisinde oluşturacağımız dosyalar veritabanıyla iletişime geçerek gelen istekleri işleyecek.

İki farklı controller dosyası oluşturacağız: **taskController.js** ve **userController.js**. Task controller, görevlerle ilgili işlemleri yönetecek ve bu işlemler arasında görev ekleme, listeleme, güncelleme ve silme gibi işlemler yer alacak. User controller ise kullanıcılarla ilgili işlemleri yönetecek, kullanıcı ekleme, kullanıcı bilgilerini güncelleme ve kullanıcıları listeleme gibi işlevler sağlayacak.

İlk olarak **userController.js** adlı dosyamızı oluşturmakla başlayalım.

```

1  const User = require("../models/User");
2
3  const getUsers = (req, res) => {
4    try {
5      User.find()
6        .then((result) => {
7          if (!result) {
8            return res.status(404).send();
9          }
10         return res.status(200).send(result);
11       })
12        .catch((error) => {
13          console.log(error);
14        });
15    } catch (err) {
16      console.log(err);
17      return res.status(500).json({ error: err.message });
18    }
19  };
20

```

Yukarıdaki kod, User modelini kullanarak **tüm kullanıcıları** veritabanından getiren bir işlevi tanımlar. **User.find()** yöntemiyle kullanıcılar bulunur ve sonuç varsa 200 durum koduyla, yoksa 404 durum koduyla istemciye gönderilir. Hata durumunda ise hata mesajları konsola yazdırılır ve 500 durum koduyla istemciye geri bildirim yapılır.

Eğer yalnızca kullanıcının id'sine göre spesifik bir kullanıcıyı veritabanından getirmek istiyorsak, aşağıda yer alan fonksiyonu kullanabiliriz.

```

1  const getUser = (req, res) => {
2    try {
3      const { id } = req.params;
4
5      User.findById(id)
6        .then((user) => {
7          if (!user) {
8            return res.status(404).send();
9          }
10         return res.status(200).send(user);
11       })
12        .catch((err) => {
13          return res.status(404).json({ error: err });
14        });
15    } catch (err) {
16      console.log(err);
17      return res.status(500).json({ error: err.message });
18    }
19  };
20

```

Burada, **req.params** içinden kullanıcının ID'sini alır ve **User.findById(id)** metodu ile veritabanından bu ID'ye sahip kullanıcıyı arar. Kullanıcı bulunamazsa 404 durum kodu ile istemciye boş bir yanıt gönderilir. Kullanıcı başarıyla bulunursa, 200 durum kodu ile kullanıcı verileri istemciye iletilir. Herhangi bir hata meydana gelirse, ilgili hata mesajı ve durum kodu ile istemciye döndürülür.

Yeni bir kullanıcı oluşturmak için aşağıdaki fonksiyonu kullanabiliriz.

```
1  const createUser = (req, res) => {
2    try {
3      const user = new User(req.body);
4      user
5        .save()
6        .then(() => {
7          return res.status(201).send(user);
8        })
9        .catch((err) => {
10         return res.status(404).json({ error: err.message });
11       });
12    } catch (err) {
13      console.log(err);
14      return res.status(500).json({ error: err.message });
15    }
16  };
17
```

Bu fonksiyon, HTTP isteği gövdesinden (req.body) gelen verilerle yeni bir kullanıcı oluşturur ve bu kullanıcıyı veritabanına kaydeder. Bu işlem, User modeline göre yeni bir kullanıcı oluşturulması ve save() metodu ile veritabanına kaydedilmesiyle gerçekleşir. Kayıt işlemi başarılı olursa, istemciye 201 durum kodu ile birlikte oluşturulan kullanıcı verileri gönderilir. Eğer bir hata oluşursa, 404 durum kodu ve hata mesajı ile birlikte hata bilgisi döner.

Var olan bir kullanıcıyı veritabanından silmek için aşağıdaki fonksiyonu kullanabiliriz.

```
const removeUser = (req, res) => {
  try {
    const { id } = req.params;

    User.findByIdAndDelete(id)
      .then(() => {
        if (!result) {
          return res.status(404).send();
        }
        return res.status(200).json({ message: "User deleted successfully" });
      })
      .catch((err) => {
        return res.status(404).json({ error: err });
      });
  } catch (err) {
    console.log(err);
    return res.status(500).json({ error: err.message });
  }
};
```

Burada, req.params içinden kullanıcının ID'sini alır ve User.findByIdAndDelete(id) metodu ile veritabanından bu ID'ye sahip kullanıcıyı arar. Kullanıcı bulunamazsa 404 durum kodu ile istemciye boş bir yanıt gönderilir. Kullanıcı başarıyla bulunursa, 200 durum kodu ile ilgili kullanıcı başarılı bir şekilde veritabanından silinir ve mesaj olarak "User deleted successfully" döndürülür.

UserController.js dosyamızda yer alan son fonksiyon ise **updateUser** fonksiyonudur. Bu fonksiyon, bir kullanıcının bilgilerini güncellemek için kullanılır. Bu işlem, HTTP isteğiyle gelen bir kullanıcı kimliği (id) ve güncellenecek kullanıcı verileriyle gerçekleştirilir. Fonksiyon, User modeline göre belirtilen kullanıcıyı bulur ve günceller. Güncelleme işlemi başarılıysa, istemciye 200 durum kodu ile birlikte "User successfully updated" mesajı ve güncellenmiş kullanıcı bilgileri gönderilir. Eğer belirtilen kullanıcı bulunamazsa, 404 durum kodu gönderilir

```
77
78 const updateUser = (req, res) => {
79   try {
80     const { id } = req.params;
81     const user = req.body;
82
83     User.findByIdAndUpdate(id, user, { new: true })
84       .then((result) => {
85         if (!result) {
86           return res.status(404).send();
87         }
88         return res
89           .status(200)
90           .json({ message: "User successfully updated", user: result });
91       })
92       .catch((err) => {
93         return res.status(404).json({ error: err });
94       });
95   } catch (err) {
96     console.log(err);
97     return res.status(500).json({ error: err.message });
98   }
99 };
100
101 module.exports = { getUsers, getUser, createUser, removeUser, updateUser };
102
```

Diğer bir controller'ımız olan taskController.js dosyamız da benzer bir mantıkla oluşturulmuştur. Bu controller'da yer alan fonksiyonları tek tek açıklamak yerine yalnızca kodlarını göstereceğim.

Tekli ve çoklu görev getirme fonksiyonları:

```
1 const Task = require("../models/Task");
2
3 const getTasks = (req, res) => {
4   try {
5     Task.find()
6       .then((result) => {
7         if (!result) {
8           return res.status(404).send();
9         }
10        return res.status(200).send(result);
11      })
12      .catch((error) => {
13        console.log(error);
14      });
15   } catch (err) {
16     console.log(err);
17     return res.status(500).json({ error: err.message });
18   }
19 };
20
21 const getTask = (req, res) => {
22   try {
23     const { id } = req.params;
24
25     Task.findById(id)
26       .then((task) => {
27         if (!task) {
28           return res.status(404).send();
29         }
30        return res.status(200).send(task);
31      })
32      .catch((error) => {
33        console.log(error);
34      });
35   } catch (err) {
36     console.log(err);
37     return res.status(500).json({ error: err.message });
38   }
39 };
40
```

Aşağıda yer alan fonksiyon yalnızca completed değeri false olan görevleri getirmektedir.

```
const getUncompletedTask = async (req, res) => {
  try {
    const tasks = await Task.find({ completed: false });
    res.status(200).send(tasks);
  } catch (err) {
    console.log(err);
    return res.status(500).json({ error: err.message });
  }
};
```

Görev oluşturma fonksiyonu:

```
1 const createTask = (req, res) => {
2   try {
3     const task = new Task(req.body);
4
5     task
6       .save()
7       .then(() => {
8         return res.status(201).send(task);
9       })
10      .catch((error) => console.log(error));
11   } catch (err) {
12     console.log(err);
13     return res.status(500).json({ error: err.message });
14   }
15 };
16
```

Görevi güncelleme fonksiyonu:

```
7  const updateTask = (req, res) => {
8    try {
9      const { id } = req.params;
10     const task = req.body;
11
12     Task.findByIdAndUpdate(id, task, { new: true })
13       .then((result) => {
14         if (!result) {
15           return res.status(404).send();
16         }
17         return res
18           .status(200)
19           .json({ task: result, message: "Task updated successfully" });
20       })
21       .catch((error) => console.log(error));
22   } catch (err) {
23     console.log(err);
24     return res.status(500).json({ error: err.message });
25   }
26 };
27
```

Görev silme fonksiyonu:

```
88  const deleteTask = (req, res) => {
89    try {
90      const { id } = req.params;
91
92      Task.findByIdAndDelete(id)
93        .then((result) => {
94          if (!result) {
95            return res.status(404).send();
96          }
97
98          return res.status(200).json({ message: "Task deleted successfully" });
99        })
100        .catch((error) => console.log(error));
101   } catch (err) {
102     console.log(err);
103     return res.status(500).json({ error: err.message });
104   }
105 };
106
107 module.exports = {
108   getTasks,
109   getTask,
110   updateTask,
111   createTask,
112   deleteTask,
113   getUncompletedTask,
114 };
115
```


2.8 Routes Klasörünün Oluşturulması

Projemizdeki API isteklerini düzenli ve yönetilebilir hale getirmek için bir routes (yönlendirme) klasörü oluşturmak önemlidir. Routes klasörü, farklı endpoint'leri tanımladığımız ve ilgili controller fonksiyonlarını çağırdığımız yerdir. Bu yapı, projenin büyümesiyle birlikte kodun okunabilirliğini ve bakımını kolaylaştırır.

İlk olarak, src klasörünün içine routes adında bir klasör oluşturalım. Bu klasörün içinde, kullanıcılar ve görevler için ayrı ayrı dosyalar oluşturacağız. Bu dosyalar, ilgili controller fonksiyonlarını çağırarak API isteklerini yönlendirecek.

Kullanıcılar için userRoute.js adında bir dosya oluşturalım. Dosyanın içeriği aşağıdaki gibi olacaktır.

```
src > routes > JS userRoute.js > ...
1  const { Router } = require("express");
2  const router = new Router();
3
4  const {
5    getUsers,
6    getUser,
7    createUser,
8    updateUser,
9    removeUser,
10 } = require("../controllers/userController");
11
12 router.route("/").get(getUsers).post(createUser);
13 router.route("/:id").get(getUser).put(updateUser).delete(removeUser);
14
15 module.exports = router;
16
17
```

Görevler için taskRoute.js adında bir dosya oluşturalım. Dosyanın içeriği aşağıdaki gibi olacaktır.

```
src > routes > JS taskRoute.js > ...
1  const { Router } = require("express");
2  const router = new Router();
3
4  const {
5    getTasks,
6    getTask,
7    createTask,
8    updateTask,
9    deleteTask,
10   getUncompletedTask,
11 } = require("../controllers/taskController");
12
13 router.route("/").get(getTasks).post(createTask);
14 router.route("/uncompleted").get(getUncompletedTask);
15 router.route("/:id").get(getTask).put(updateTask).delete(deleteTask);
16
17 module.exports = router;
18
19
```

2.9 Ana Dosyanın Kodlanması

Son adım olarak uygulamamızın çalışacağı ana dosyayı kodlayacağız. Aşağıdaki görselde yer alan kod parçası ana dosyamızın bitmiş halidir.

```
src > js index.js > ...
1  require("dotenv").config();
2  const express = require("express");
3  const app = express();
4  const { connect } = require("./db/config");
5  const userRoute = require("./routes/userRoute");
6  const taskRoute = require("./routes/taskRoute");
7
8  app.use(express.json());
9
10 app.use("/users", userRoute);
11 app.use("/tasks", taskRoute);
12
13 const connectDB = async () => {
14   try {
15     await connect();
16   } catch (error) {
17     console.log(error);
18   }
19 };
20
21 connectDB();
22
23 const PORT = process.env.PORT || 3000;
24
25 app.listen(PORT, () => {
26   console.log(`Server is listening on port ${PORT}`);
27 });
28
```

Uygulamanın başlangıcında, dotenv paketi kullanılarak .env dosyasındaki ortam değişkenleri yüklenir. Express framework'ü kullanılarak bir web uygulaması oluşturulur ve MongoDB bağlantısı için gerekli olan connect fonksiyonu içe aktarılır. Kullanıcı ve görev rotaları ayrı dosyalardan alınır ve uygulamaya entegre edilir. JSON formatındaki istekleri işlemek için gerekli yapılandırma yapılır ve kullanıcı ve görev rotaları sırasıyla /users ve /tasks yollarına bağlanır.

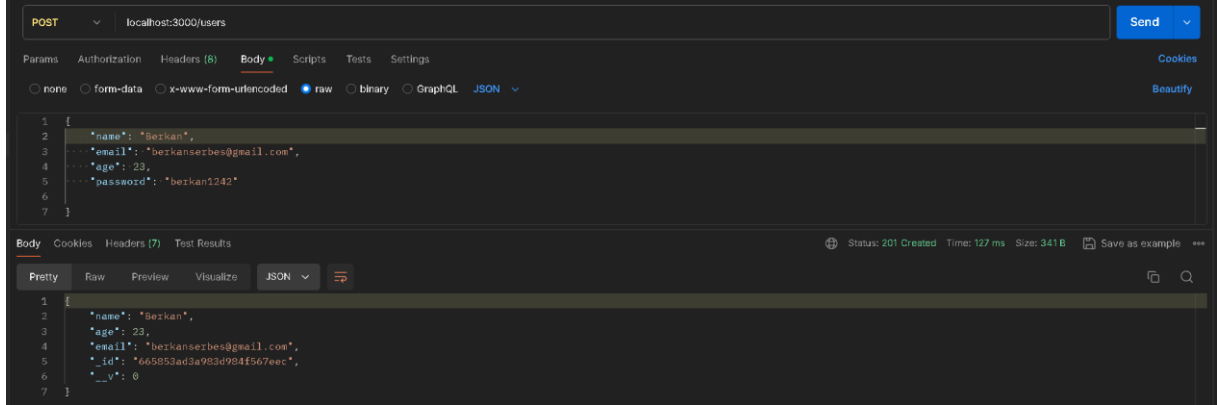
Daha sonra, MongoDB'ye bağlanmak için connectDB adlı bir asenkron fonksiyon tanımlanır ve bu fonksiyon uygulama başlatıldığında çağrılır. Bu, uygulamanın MongoDB veritabanına başarılı bir şekilde bağlandığından emin olmak için yapılır. Bağlantı sağlandığında veya bağlantı sırasında bir hata oluştuğunda bu durum konsola yazdırılır. Son olarak, uygulama belirtilen bir portta dinlemeye başlar. Eğer .env dosyasında bir port numarası belirtilmemişse, varsayılan olarak 3000 numaralı port kullanılır. Uygulama başarılı bir şekilde başlatıldığında, kullanılan port numarası konsola yazdırılır.

2.10 Uygulamanın Postman Üzerinden Test Edilmesi

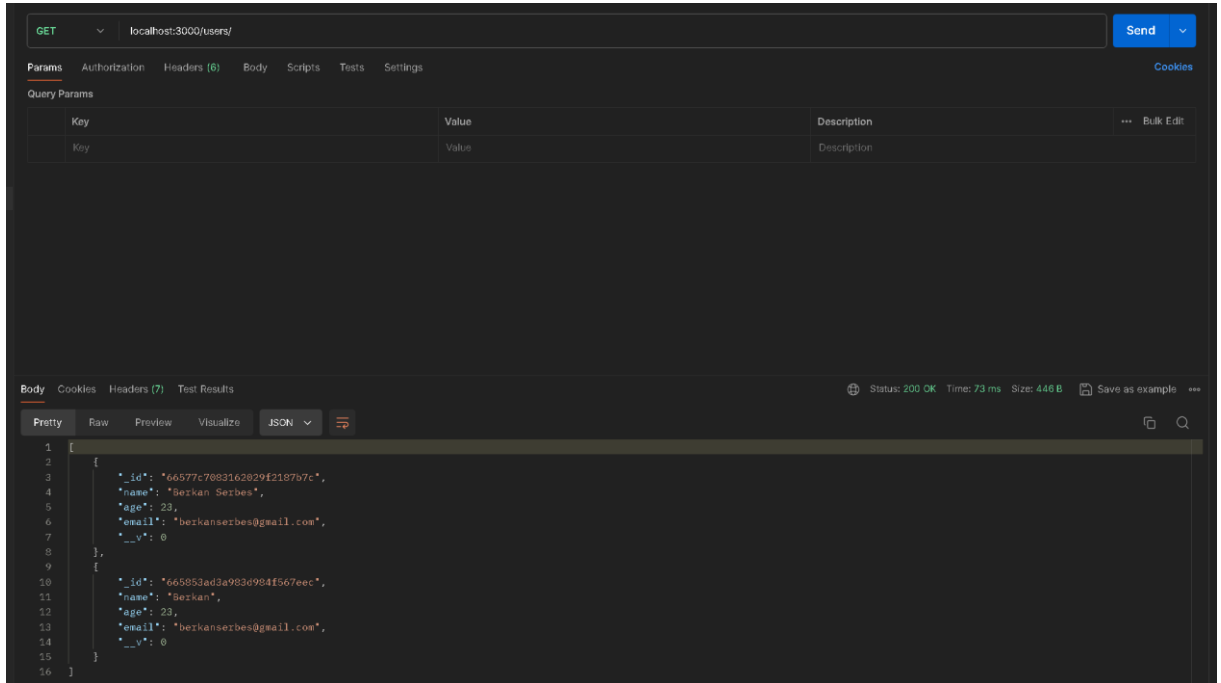
Uygulamamızı Postman üzerinden test ederek, kullanıcı ve görev rotalarının doğru bir şekilde çalışıp çalışmadığını, veritabanına başarılı bir şekilde veri eklenip eklenmediğini ve API isteklerine uygun yanıtlar alıp almadığımızı kontrol edebiliriz. Bu, uygulamanızın beklenen şekilde davrandığından emin olmanızı sağlar ve olası hataları tespit etmenize yardımcı olur. Şimdi, geliştirdiğimiz uygulamanın işlevselliğini Postman ile nasıl test edebileceğimize bir göz atalım.

İlk olarak kullanıcı servisimizi test edelim.

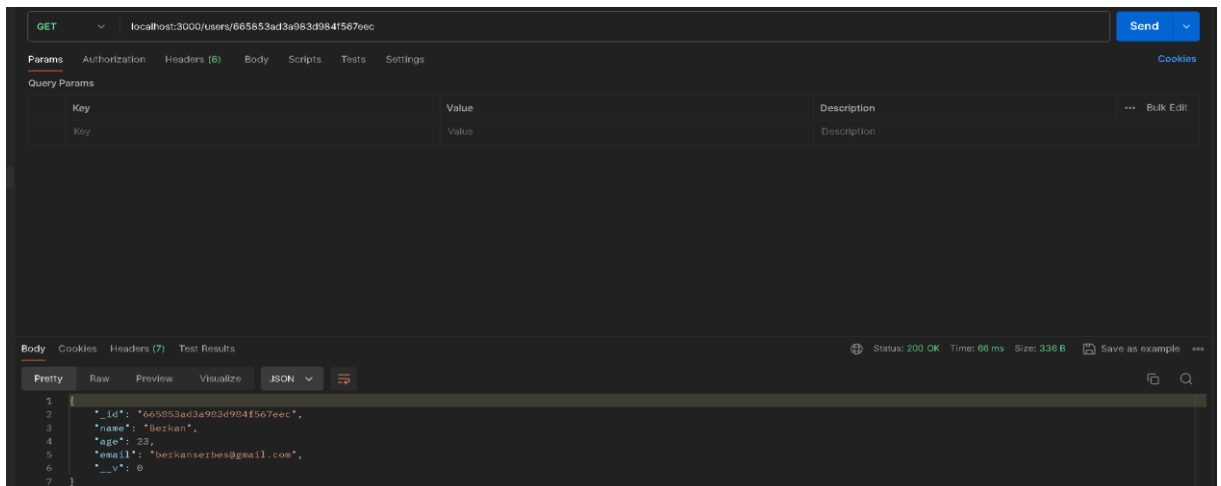
Kullanıcı oluşturma:



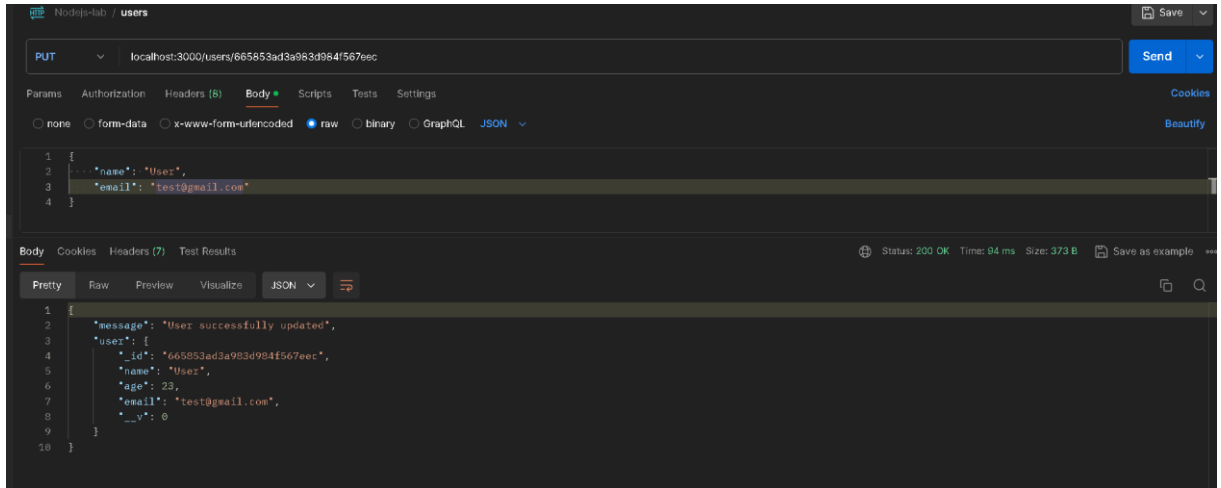
Tüm kullanıcıları getirme:



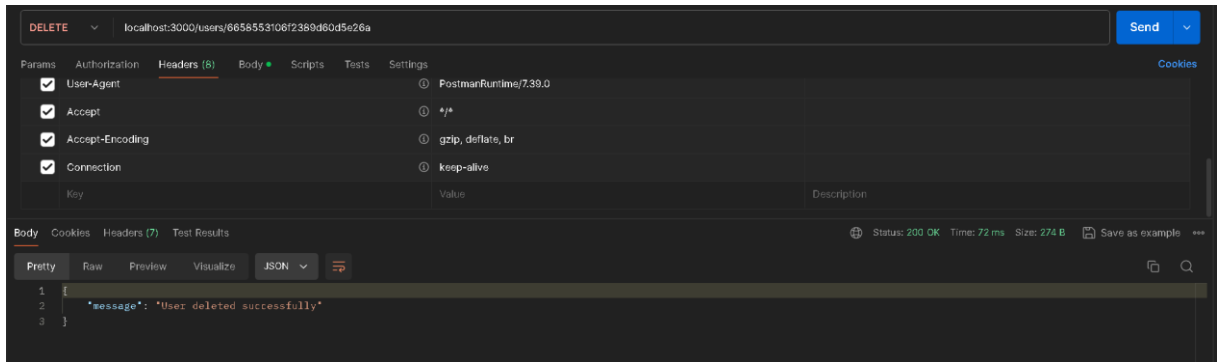
ID'ye göre kullanıcıyı getirme:



Kullanıcı güncelleme:

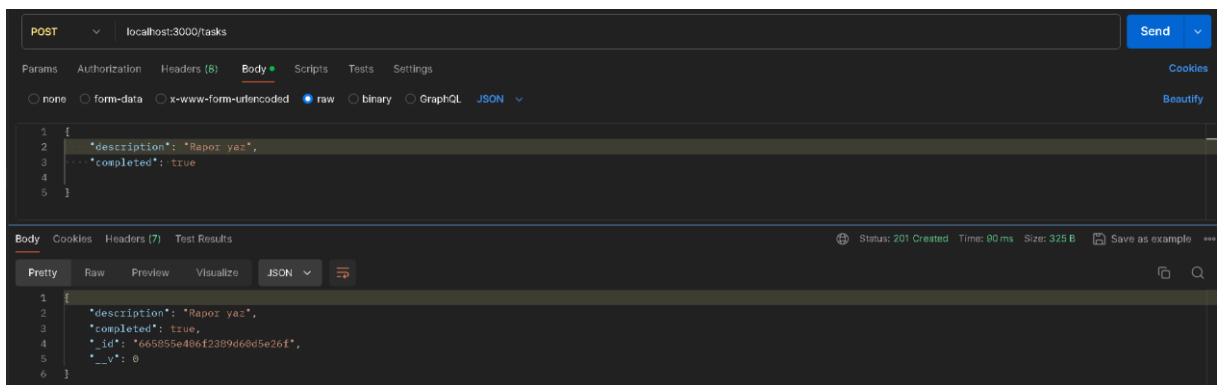


Kullanıcı silme:

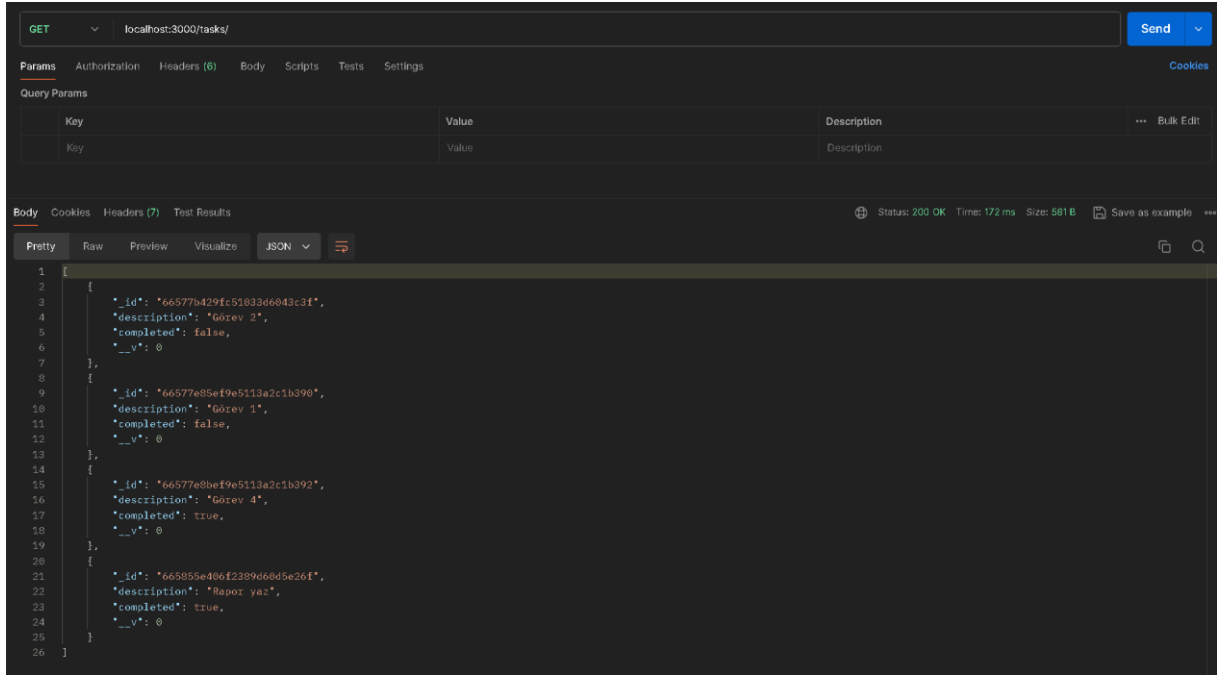


Kullanıcı servislerimiz başarılı bir şekilde çalışmaktadır. Şimdi de görev servisimizi test edelim.

Görev oluşturma:



Tüm görevleri getirme:



GET localhost:3000/tasks/ Send

Params Authorization Headers (6) Body Scripts Tests Settings Cookies

Query Params

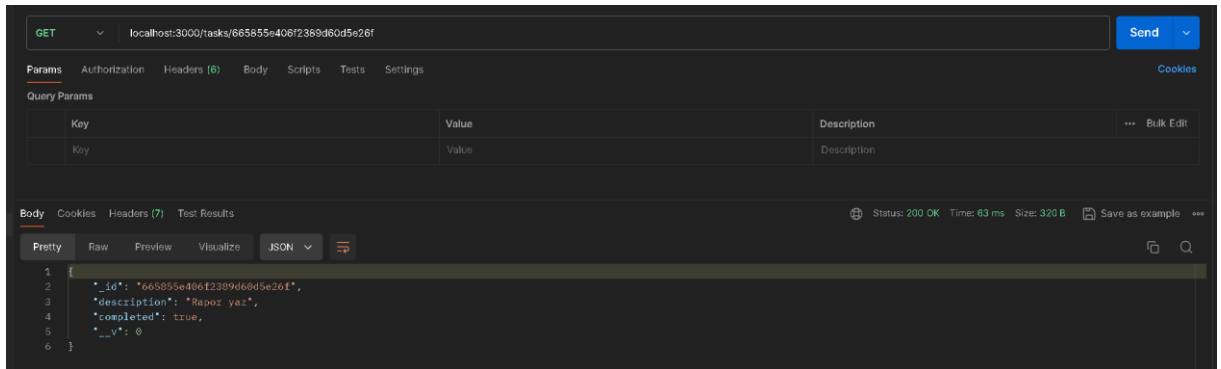
Key	Value	Description	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (7) Test Results Status: 200 OK Time: 172 ms Size: 581 B Save as example

Pretty Raw Preview Visualize JSON

```
1 {
2   [
3     {
4       "_id": "66577b429fc51833d6043c3f",
5       "description": "Görev 2",
6       "completed": false,
7       "__v": 0
8     },
9     {
10      "_id": "66577e85ef9e5113a2c1b390",
11      "description": "Görev 1",
12      "completed": false,
13      "__v": 0
14    },
15    {
16      "_id": "66577e8bef9e5113a2c1b392",
17      "description": "Görev 4",
18      "completed": true,
19      "__v": 0
20    },
21    {
22      "_id": "665855e406f2389d60d5e26f",
23      "description": "Rapor yaz",
24      "completed": true,
25      "__v": 0
26    }
27  ]
28 }
```

ID'ye göre görev getirme:



GET localhost:3000/tasks/665855e406f2389d60d5e26f Send

Params Authorization Headers (6) Body Scripts Tests Settings Cookies

Query Params

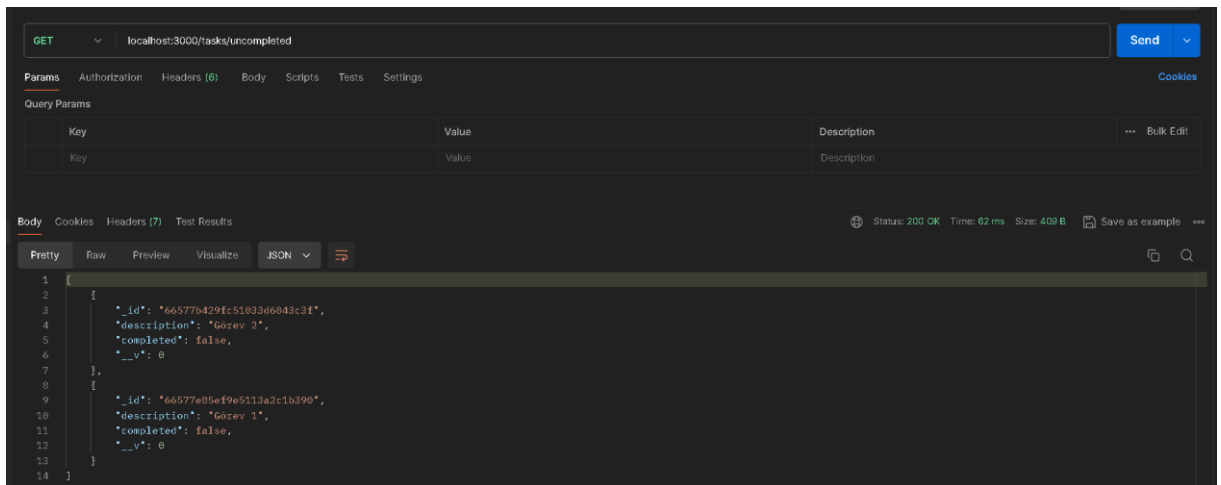
Key	Value	Description	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (7) Test Results Status: 200 OK Time: 63 ms Size: 320 B Save as example

Pretty Raw Preview Visualize JSON

```
1 {
2   "_id": "665855e406f2389d60d5e26f",
3   "description": "Rapor yaz",
4   "completed": true,
5   "__v": 0
6 }
```

Completed değeri false olan görevleri getirme:



GET localhost:3000/tasks/uncompleted Send

Params Authorization Headers (6) Body Scripts Tests Settings Cookies

Query Params

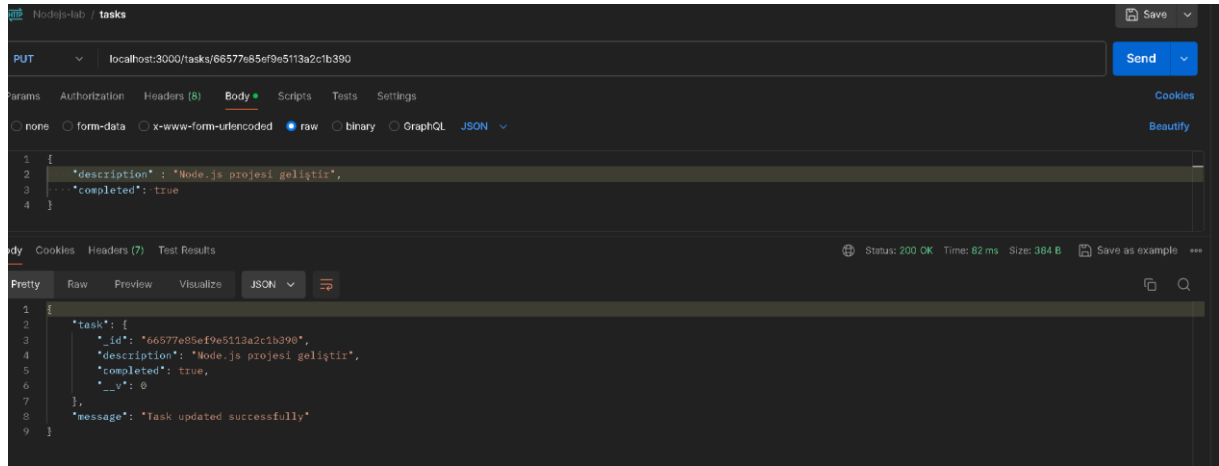
Key	Value	Description	...	Bulk Edit
Key	Value	Description		

Body Cookies Headers (7) Test Results Status: 200 OK Time: 62 ms Size: 408 B Save as example

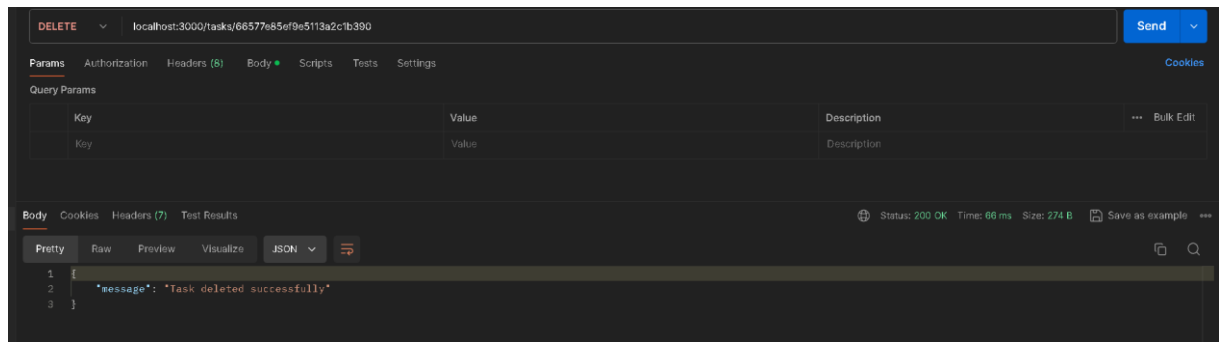
Pretty Raw Preview Visualize JSON

```
1 {
2   [
3     {
4       "_id": "66577b429fc51833d6043c3f",
5       "description": "Görev 2",
6       "completed": false,
7       "__v": 0
8     },
9     {
10      "_id": "66577e85ef9e5113a2c1b390",
11      "description": "Görev 1",
12      "completed": false,
13      "__v": 0
14    }
15  ]
16 }
```

Görevi güncelleme:



Görevi silme:



3. Kaynakça

<https://chatgpt.com/>