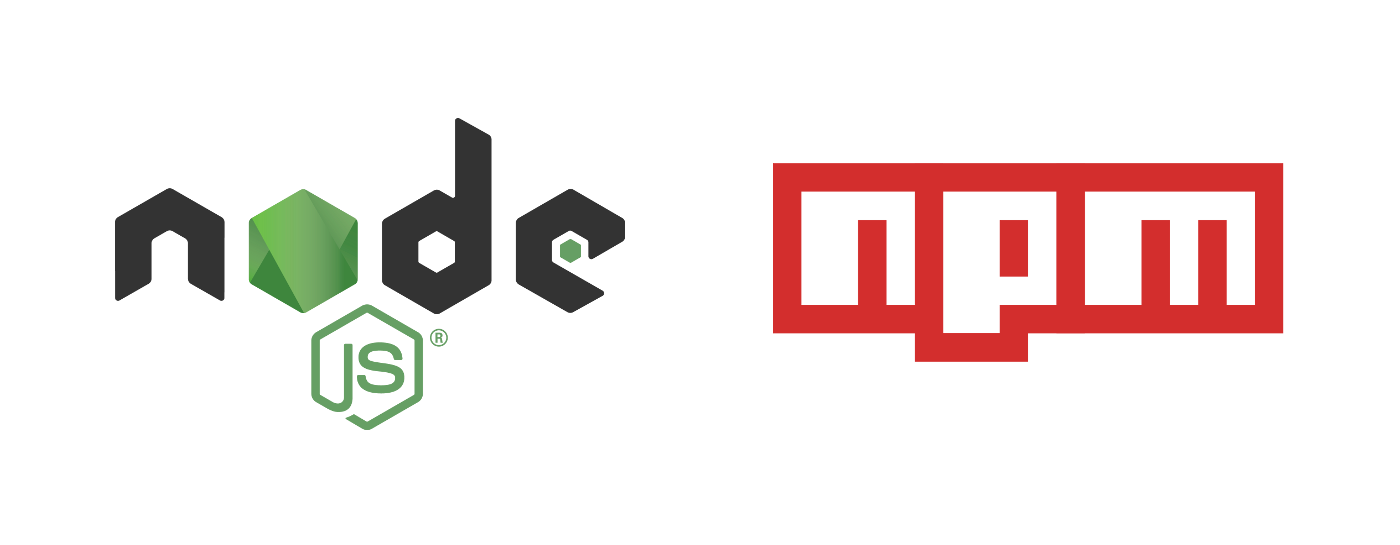
# Days\_66 Node.js nedir? gerekli kurulumlar \*\*work-space/node.js---> 30/08/2025

**KURULUM 🡪 Proje oluşturma:**

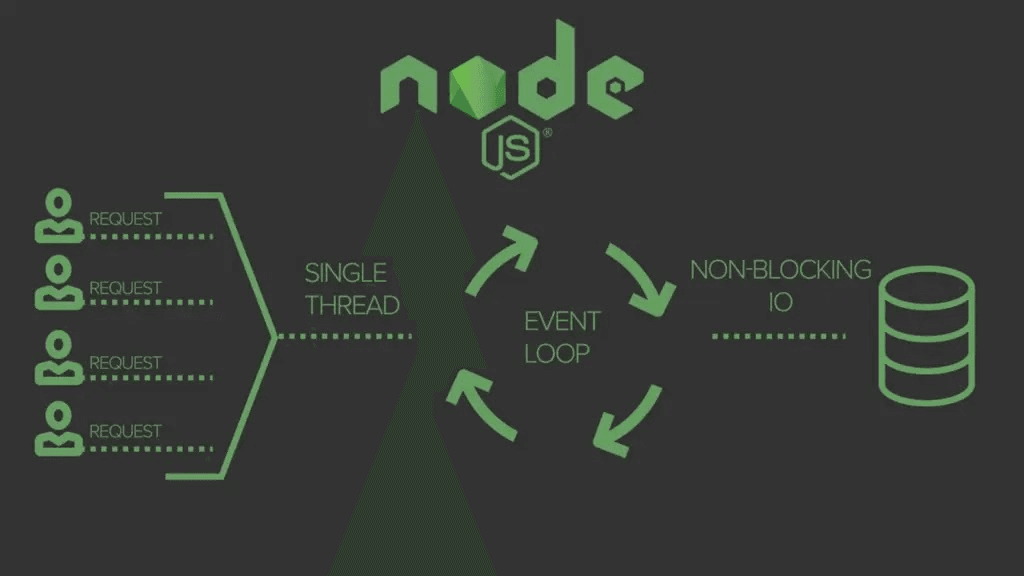
* npm init -y && npm install express && npm install -D typescript ts-node @types/node @types/express jest ts-jest @types/jest && npx tsc --init && npx ts-jest config:init
* npm i nodemon
* npm i ejs
* npm i bodey-parser



**Node.js'in Günümüzdeki Yeri ve Önemi**

**Node.js**, JavaScript'i sunucu tarafında (server-side) çalıştırmayı mümkün kılan, açık kaynaklı ve olay tabanlı (event-driven) bir çalışma ortamıdır. Google'ın Chrome V8 JavaScript motoru üzerine inşa edilmiştir. Geleneksel sunucu dillerinin (PHP, Java, Python gibi) aksine, Node.js eşzamansız (asynchronous) ve engellemeyen (non-blocking) I/O modeliyle öne çıkar. Bu model, özellikle gerçek zamanlı uygulamalar, mikro hizmetler ve yoğun veri akışının olduğu sistemler için idealdir.

Günümüzde Node.js, modern web geliştirmesinin en önemli bileşenlerinden biri haline gelmiştir. Artık sadece basit web sunucuları kurmakla kalmıyor, aynı zamanda karmaşık kurumsal uygulamalardan mobil uygulama backend'lerine, hatta masaüstü uygulamalarına kadar birçok alanda kullanılıyor.



**Node.js'i Bu Kadar Önemli Kılan Özellikler**

**1. Yüksek Performans ve Ölçeklenebilirlik**

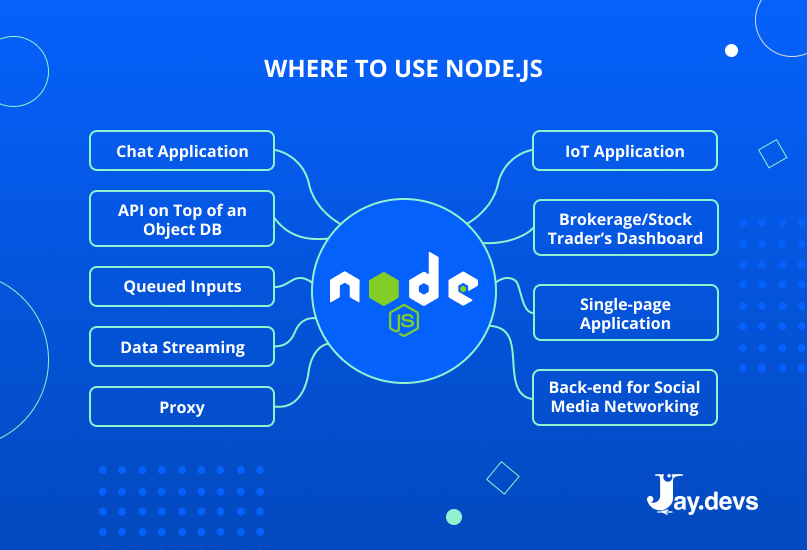
Node.js'in en büyük avantajı, eşzamansız mimarisidir. Bir istek geldiğinde, Node.js o isteği işlemini engellemeden arka plana atar ve bir sonraki isteği kabul etmeye devam eder. İşlem tamamlandığında, bir geri arama (callback) fonksiyonuyla sonuçları döner. Bu sayede, aynı anda binlerce bağlantıyı çok daha az sistem kaynağıyla yönetebilir. Bu özellik, özellikle anlık mesajlaşma, canlı yayın ve online oyunlar gibi gerçek zamanlı uygulamalar için hayati önem taşır.

**2. JavaScript Ekosistemi**

JavaScript, günümüzün en yaygın kullanılan programlama dillerinden biridir. Node.js sayesinde frontend (ön yüz) ve backend (arka yüz) geliştirme aynı dil üzerinde yapılabilir. Bu, geliştirici ekiplerin daha verimli çalışmasını sağlar ve kod paylaşımını kolaylaştırır. Ayrıca, **npm (Node Package Manager)** adı verilen devasa bir paket yöneticisi ekosistemine sahiptir. npm'de, hemen hemen her ihtiyaca yönelik milyonlarca açık kaynaklı kütüphane ve modül bulunur. Bu durum, geliştirme süreçlerini hızlandırır ve tekrarlayan işleri azaltır.

**3. Mikro Hizmet Mimarisi (Microservices)**

Günümüzdeki modern yazılım mimarilerinde büyük monolitik uygulamalar yerine, bağımsız ve küçük hizmetlerden oluşan mikro hizmetler tercih edilir. Node.js'in hafif yapısı ve modüler mimarisi, mikro hizmetler oluşturmak için mükemmel bir uyum sağlar. Her mikro hizmet, kendi Node.js sunucusunda çalışabilir ve farklı teknolojilerle bile iletişim kurabilir.



**Node.js'in Kullanım Alanları**

* **Gerçek Zamanlı Uygulamalar:** Anlık mesajlaşma, canlı sohbet uygulamaları, online oyunlar, gerçek zamanlı bildirim sistemleri. **Socket.IO** gibi kütüphaneler sayesinde bu tür uygulamaları geliştirmek oldukça kolaydır.
* **API Geliştirme:** RESTful API'ler ve GraphQL sunucuları oluşturmak için idealdir. **Express.js**, **Koa.js** ve **NestJS** gibi frameworkler, bu alanda sıkça kullanılır.
* **Mikro Hizmetler:** Büyük uygulamaları küçük, bağımsız ve yönetilebilir parçalara ayırmak için kullanılır.
* **Veri Yoğun Uygulamalar:** Veri akışının sürekli olduğu ve I/O işlemlerinin yoğun olduğu uygulamalar (örneğin, dosya yükleme servisleri, veri işleme boru hatları).
* **Tek Sayfa Uygulamaları (SPA) için Backend:** **React**, **Angular** veya **Vue.js** gibi frontend frameworkleri ile geliştirilen SPA'lar için güçlü bir backend sunar.
* **Komut Satırı Araçları:** Geliştiricilerin işlerini otomatikleştirmek için kullandığı araçlar (örneğin, **Grunt** ve **Gulp** gibi görev çalıştırıcılar).

**Node.js'in Avantajları ve Dezavantajları**

**Avantajları**

* **Yüksek Performans:** Eşzamansız yapısı sayesinde yüksek trafikli uygulamalar için idealdir.
* **Tek Dil:** JavaScript ile hem frontend hem de backend geliştirebilme imkanı sunar.
* **Geniş Ekosistem:** npm sayesinde geliştirme süreçleri oldukça hızlanır.
* **Ölçeklenebilirlik:** Küçük hizmetler halinde kolayca ölçeklenebilir.
* **Büyük Topluluk Desteği:** Aktif ve büyüyen bir geliştirici topluluğu vardır.

**Dezavantajları**

* **Yoğun İşlem Gerektiren Uygulamalar:** CPU-yoğun (örneğin, karmaşık matematiksel hesaplamalar) uygulamalarda eşzamanlı mimari avantaj sağlamaz ve performans sorunlarına yol açabilir.
* **Callback Cehennemi (Callback Hell):** Eşzamansız yapı, karmaşık iç içe geçmiş callback fonksiyonları nedeniyle kodun okunabilirliğini zorlaştırabilir. Bu sorun **async/await** gibi modern yaklaşımlarla büyük ölçüde çözülmüştür.
* **Olgun Olmayan Frameworkler:** Diğer dillerdeki gibi oturmuş, standart bir framework ekosistemi hala gelişmekte olan bir yapıdadır.

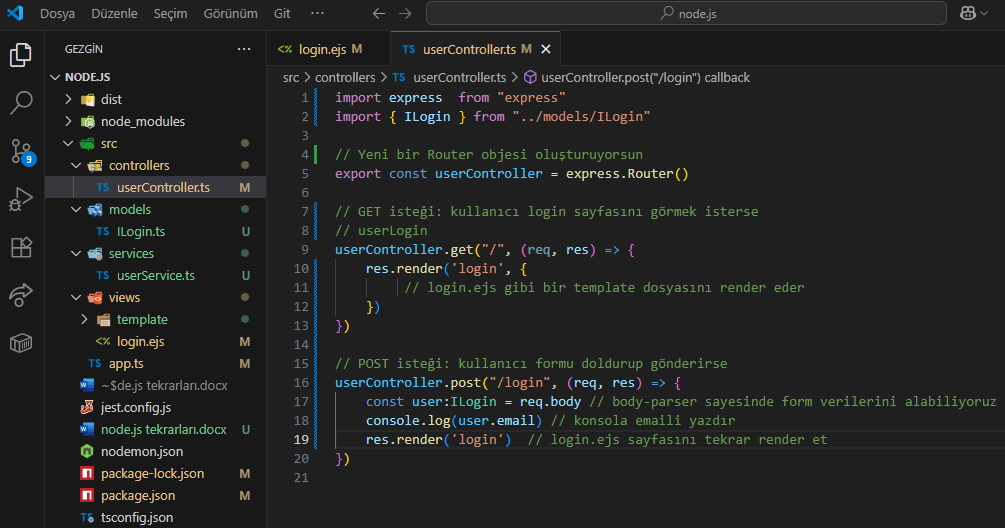
**Özetle Node.js Nedir?**

* **Tanım**: Node.js, tarayıcı dışında JavaScript çalıştırmamızı sağlayan bir **çalışma ortamıdır (runtime)**.
* **Çekirdeği**: Google’ın Chrome’da kullandığı **V8 JavaScript motoru** üzerine kuruludur.
* **Özellikleri**:
  + Tek iş parçacıklı (single-threaded) ama **event-driven** mimariye sahiptir.
  + **Non-blocking I/O** sayesinde çok sayıda istemciyle aynı anda iletişim kurabilir (özellikle web sunucuları için güçlüdür).
  + Sunucu tarafında JavaScript ile API’ler, web uygulamaları, gerçek zamanlı uygulamalar (chat, oyun, IoT vb.) geliştirmemizi sağlar.

**Express.js Nedir?**

* Express, **Node.js için geliştirilmiş bir web uygulama çatısı (framework)**’dır.
* Node.js’in ham yapısını daha kolay ve düzenli kullanabilmemizi sağlar.
* Sağladıkları:
  + **Router sistemi** → URL’leri ve HTTP isteklerini yönetme
  + **Middleware desteği** → istek/cevap sürecinde araya girip işlem yapabilme
  + **Template engine entegrasyonu** → res.render() ile HTML şablonlarını server-side render etme
  + Daha okunabilir ve modüler bir yapı

**Koddaki Örnek Üzerinden Açıklama**



**Burada olan şeyler:**

1. **express.Router()** → küçük bir **mini uygulama** gibi çalışır. Belirli path’ler için get, post, put, delete gibi route’lar tanımlayabilirsin.
   * Sonra bu userController’ı app.ts veya server.ts dosyanda app.use("/users", userController) diyerek ana uygulamaya bağlarsın.
2. **res.render('login')** → Express’in template engine ile (mesela **EJS**) HTML sayfasını oluşturup istemciye göndermesi.
3. **req.body →** Kullanıcının gönderdiği form verilerini almak için. Bunun çalışması için Express’te body-parser ya da **express.json() / express.urlencoded()** middleware’i gerekir.

**Genel Akış**

* Kullanıcı /users/ adresine gider → GET / çalışır → login sayfası açılır.
* Kullanıcı formu doldurup gönderir → POST /login çalışır → bilgiler backend’e gelir → log basılır → tekrar login sayfası render edilir.

**Body-parser nedir?**

* body-parser, Express.js uygulamalarında **HTTP isteği ile gelen verileri (request body)** kolayca okumak için kullanılan bir **middleware**’dir.
* Normalde Node.js, gelen ham HTTP isteğini **string/buffer** halinde bırakır. Bu ham veriyi anlamlı hale getirmek (JSON, form verisi vs.) için body-parser veya Express’in kendi gömülü fonksiyonları kullanılır.

**Ne İşe Yarar?**

1. **JSON verisi** (application/json) gönderildiğinde, req.body içinden doğrudan JS objesi olarak okunmasını sağlar.
2. **Form verisi** (application/x-www-form-urlencoded) gönderildiğinde, req.body içinden key-value olarak alınmasını sağlar.

**Kullanımı:**

Express 4.16+ sürümünden itibaren body-parser’ın çoğu özelliği Express’in içine gömüldü. Yani artık şunu yazmak yeterli:

**userController.post("/login", (req, res) => {**

**const user: ILogin = req.body** // işte burası body-parser/express.json sayesinde çalışıyor

**console.log(user.email)**

**res.render('login')**

**})**

👉 Yani özetle:  
Body-parser = gelen request body verilerini (JSON, form vs.) okunabilir hale getiren bir Express middleware’i.

**🔹 Günlük hayattan örnek 🚗**

* Diyelim ki **arabanı servise götürüyorsun**.
* Sen görevliye diyorsun: "Lastikler patladı, değiştirilsin."
* Senin söylediğin cümle aslında ham veri (request body).
* **body-parser** burada **çevirmen** gibi davranıyor → senin sözünü anlamlı hale getiriyor:

"Müşteri lastik değişimi istiyor."

* **Controller** bunu alıyor → "Tamam, bu isteği lastik servisine (service katmanı) ileteyim."
* **Service** → "Önce uygun lastik var mı bakayım, varsa değiştiriyorum."
* Sonuç sana geri dönüyor.

Yani **body-parser olmadan, servis lastik isteğini anlayamazdı.** 😅

EJS: “**Node.js’de neden <%= %> gibi EJS tagları kullanılıyor?**” sorunun cevabı aslında **sunucu taraflı HTML üretme ihtiyacından** geliyor.

**🚀 Neden Böyle Bir Yapı Var?**

**1. Dinamik HTML üretmek için**

* Normalde HTML statiktir → içine yazdığın şey hep aynıdır.
* Ama web uygulamalarında çoğu zaman değişken veriler göstermek gerekir (ör. kullanıcı adı, ürün listesi, yorumlar vs).
* İşte <%= %> sayesinde **JavaScript değişkenlerini HTML içine gömebiliyorsun**.

**2. JavaScript’i doğrudan HTML’de çalıştırmak için**

* EJS, HTML’in içine **JavaScript kodu yazmana izin verir**.
* Bu sayede if/else, for döngüleri, listeleme gibi şeyleri HTML tarafında kolayca yaparsın.

**3. MVC mantığına uyduğu için**

* Node.js + Express genelde **MVC (Model–View–Controller)** mantığında kullanılır:
  + **Model** → veri (ör. veritabanından gelen kullanıcı bilgisi)
  + **Controller** → veriyi alıp işleyip şablona yollar
  + **View (EJS)** → o veriyi HTML’e yerleştirir

Yani <%= %> aslında **“controller’dan gelen veriyi view’da göstermek”** işlevini yapıyor.

**4. Güvenlik (Escape mantığı)**

* <%= %> → XSS saldırılarına karşı veriyi **escape ederek** basar (HTML kodu çalıştırmaz).
* <%- %> → escape etmeden HTML’i direkt basar.

Böylece geliştirici isterse güvenli şekilde veri gösterebilir, isterse HTML render ettirebilir.

**✅ Özet:**

Node.js’de EJS (ve benzeri templating engine’lerdeki tag yapısı):

* **HTML’i dinamik hale getirmek** için,
* **JavaScript kodunu doğrudan şablon içinde çalıştırmak** için,
* **MVC düzenine uygun “veri → view” akışını sağlamak** için kullanılıyor.

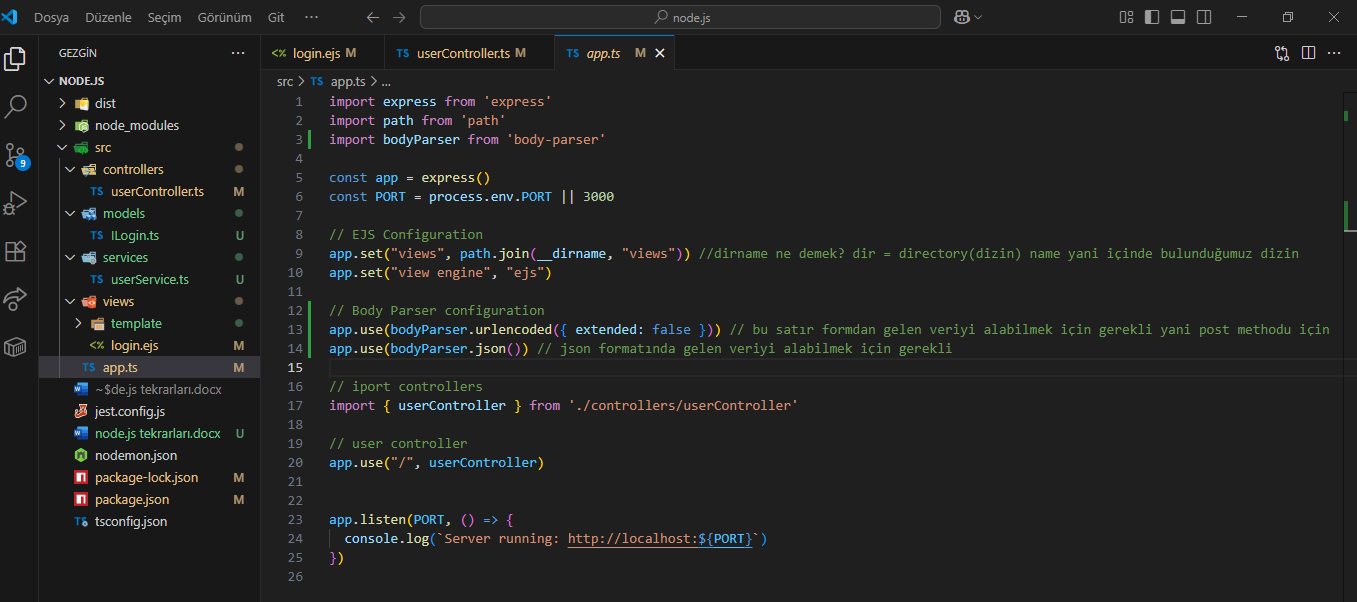
**📌 Escape Nedir?**

Escape etmek, Veriyi olduğu gibi değil, **özel karakterlerini dönüştürerek** yazdırmak demektir.

**Escape etmek** = özel HTML/JS karakterlerini güvenli eşdeğerlerine dönüştürmek

* + < → &lt;
  + > → &gt;
  + & → &amp;
  + " → &quot;
* <%= %> → escape eder (güvenli).
* <%- %> → escape etmez (HTML çalışır).

App.ts dosyası:



**1. Import Kısımları**

* **express** → Web sunucusu oluşturmak için ana framework.
* **path** → Dosya ve klasör yollarını yönetmek için Node.js'in yerleşik modülü.
* **body-parser** → POST veya PUT isteklerindeki verileri (req.body) yakalamak için kullanılan middleware.

**2. Sunucu Tanımı**

**const app = express()**

**const PORT = process.env.PORT || 3000**

* app → Express uygulaması.
* PORT → Eğer ortam değişkeninde (process.env.PORT) bir port varsa onu kullanır, yoksa **3000** portunu kullanır.

**3. EJS Yapılandırması**

**app.set("views", path.join(\_\_dirname, "views"))**

**app.set("view engine", "ejs")**

* views → EJS şablonlarının bulunduğu klasörü tanımlıyor.
  + \_\_dirname → app.ts dosyasının bulunduğu dizin.
  + path.join(\_\_dirname, "views") → src/views klasörünü işaret ediyor.
* view engine → EJS’yi template engine olarak ayarlıyor.

**4. Body Parser Yapılandırması**

**app.use(bodyParser.urlencoded({ extended: false }))**

**app.use(bodyParser.json())**

* **urlencoded** → application/x-www-form-urlencoded tipinde gelen verileri çözmek için (formlardan gelen veriler).
  + extended: false → sade key-value eşleşmeleri için.
* **json** → application/json tipinde gelen verileri çözmek için.

Böylece req.body içinden email, password gibi verilere erişebilirsin.

**5. Controller İçe Aktarma**

**import { userController } from './controllers/userController'**

* userController → Kullanıcı işlemleri (/login vs.) için yazdığın router.
* controllers/userController.ts dosyasındaki route tanımlarını burada uygulamana bağlıyorsun.

**6. Router Kullanımı**

**app.use("/", userController)**

* / ile başlayan tüm istekler userController tarafından işlenecek.
* Örn: /login isteği userController.post('/login', ...) metoduna düşecek.

**7. Sunucu Başlatma**

**app.listen(PORT, () => {**

**console.log(`Server running: http://localhost:${PORT}`)**

**})**

* Belirlenen portta Express sunucusu çalıştırılıyor.
* Terminale “Server running: [http://localhost:3000”](http://localhost:3000%E2%80%9D) yazdırıyor.

**📌 Özetle**

Bu app.ts şunları yapıyor:

1. Express app başlatıyor.
2. Görünüm motoru olarak EJS’yi ayarlıyor.
3. Gelen form ve json verilerini çözmek için body-parser ekliyor.
4. Kullanıcı controller’ını (userController) uygulamaya bağlıyor.
5. 3000 portunda sunucuyu ayağa kaldırıyor.

# Days\_67 Login-Register işlemleri \*\*work-space/node.js---> 31/08/2025

EJS’de Dinamiklik

Ejs ile statik olan html’yi dinamik hale getiriyoruz. Mesela html içinde if döngüsü yazıp toplama işlemi yaptıramazsın. Html sadece tasarım kısmıdır. Tema motoru tarafından ejs tagları ile anlamlandırıp ve sistemi daha dinamik hale getiriyor. Yani:

* **HTML** = tasarım iskeleti
* **EJS** = bu tasarımı **dinamik hale getiren motor**

**🔹 HTML**

* **Statik bir yapıdır.**
* Yani sadece yazdığın şekilde tarayıcıya gider.
* İçinde değişken, koşul (if), döngü (for) çalışmaz.
* Görevi tamamen **sayfanın iskeletini ve tasarımını** vermek.

**🔹 EJS (Embedded JavaScript)**

* HTML’nin içine **JavaScript gömmemizi** sağlar.
* Bir “template engine” yani **şablon motorudur**.
* <%= %>, <% %>, <%- %> gibi özel tag’lar kullanarak:
  + **Değişkenleri yazdırabilir**
  + **Koşullar** (if/else) çalıştırabilir
  + **Döngüler** (for/each) çalıştırabilir
  + **Parçalı view’leri include** edebilirsin

Node.js’te include mantığı:

**EJS** kullanırken sayfa parçalarını tekrar tekrar yazmamak için include mantığı kullanılır

**🔹 Normal HTML’de Durum**

Normalde HTML’de **aynı header, footer, menü** kodlarını her dosyada yazmak zorunda kalırsın.  
Bu hem zahmetli hem de değişiklik yapınca her dosyada tek tek güncellemen gerekir.

**🔹 EJS’de Include Mantığı**

EJS sana include özelliği veriyor:

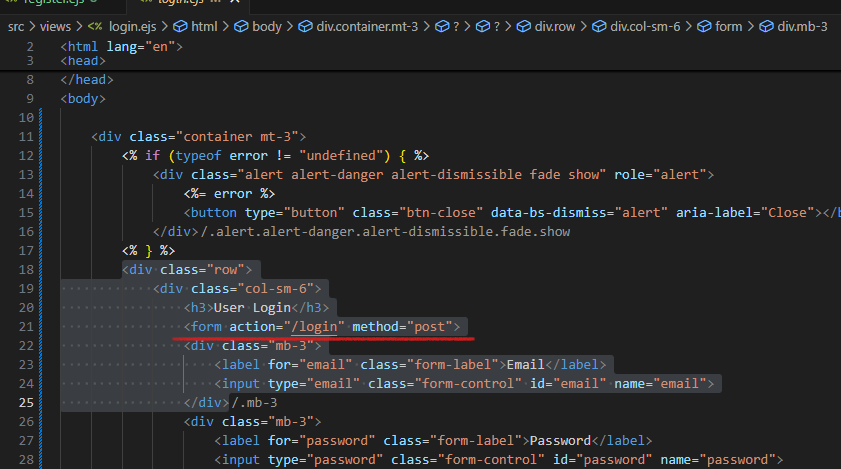
* Ortak parçaları (örneğin: **header.ejs, footer.ejs, navbar.ejs**) ayrı dosyaya ayırabilirsin.
* Sonra include ile istediğin yerde çağırırsın.

**🔹 Buradaki Farklar**

* <%- include(...) %> → include edilen dosyanın içeriğini **render edip ekler**.
* Boostrap.ejs dosyasını 20 sayfa da kullansan sadece **1 dosya** üzerinde değişiklik yaparsın.

👉 Özet:  
**include** = tekrar eden kodları **parçalara ayırıp yeniden kullanabilmeni sağlayan mantık**.

Login.ejs



Formlarda kullanılan **GET** ve **POST** metodlarının farkı hem **güvenlik** hem de **kullanım amacı** ile ilgilidir.

**🔹 1. GET Methodu**

* **Veriyi URL üzerinden gönderir** (?email=abc@mail.com&password=1234 gibi).
* URL'de görüneceği için herkes görebilir, loglarda saklanır, paylaşılabilir.
* **Güvenli değil**, o yüzden **şifre, kimlik bilgisi** gibi kritik şeyler için **ASLA kullanılmaz**.

✅ Kullanım alanları:

* **Arama/Filtreleme** = GET
* **Ürün linki paylaşımı** = GET

**🔹 2. POST Methodu**

* **Veriyi HTTP body** kısmında gönderir (URL'de görünmez).
* **Daha güvenlidir** çünkü URL’ye yazılmaz (ama tabii HTTPS kullanılmazsa yine ağda yakalanabilir).
* Büyük veriler gönderilebilir (form, dosya upload).

✅ Kullanım alanları:

* Kullanıcı girişleri (login)
* Kayıt olma (register)
* Form ile veri kaydetme (yorum ekleme, sipariş verme, dosya yükleme vb.)

**🔹 Bir örneğine uyarlayalım:**

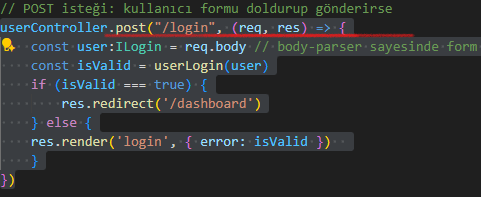
* **Login formu** → POST kullanılmalı. Çünkü email+şifreyi **gizlemek** gerekir.
* **Sahibinden ürün linki** → GET kullanılmalı. Çünkü linki arkadaşına gönderdiğinde sayfa açılmalı, içinde kritik bilgi yok, sadece ürünün id’si veya adı var (/ilan/12345).
* **Arama kutusu (Google, sahibinden arama çubuğu)** → GET. Çünkü aradığın kelime (?q=iphone) URL’ye yazılsa sorun olmaz, hatta iyi olur (bookmark, paylaşım yapılabilir).

**🔹 Özet Tablo**

| **Özellik** | **GET** | **POST** |
| --- | --- | --- |
| Veri nereye gider? | URL query string | HTTP body |
| Güvenlik | Düşük (URL’de görünüyor) | Daha güvenli (body’de saklı) |
| Veri boyutu | Limitli (URL uzunluğu) | Daha büyük veri/dosya gönderilir |
| Bookmark | Yapılabilir | Yapılamaz |
| Kullanım alanı | Arama, filtreleme, paylaşılabilir linkler | Login, register, yorum, ödeme, dosya yükleme |

(\*\* 🔖 Bookmark (Türkçe: Yer imi))

userController.ts



Node.js / Express’te controller metodlarının parametreleri genelde şu şekilde olur:

**userController.post("/login", (req, res) => { ... })**

**1. req → Request (İstek)**

Kullanıcının **sunucuya gönderdiği tüm bilgileri** taşır.

* Formdan gelen veriler (ör. email, password) → req.body
* URL parametreleri (ör. /users/5) → req.params
* Query string (/search?keyword=node) → req.query
* Header bilgileri (ör. Authorization, Content-Type) → req.headers
* Cookie bilgileri → req.cookies

📌 örnekte:

**const user: ILogin = req.body**

Formdaki email ve password değerleri burada yakalanıyor.

**2. res → Response (Yanıt)**

Sunucunun **kullanıcıya geri döndüğü cevabı** temsil eder.

* **HTML sayfası render etme** → res.render("sayfa")
* **JSON veri döndürme** → res.json({ message: "ok" })
* **Yönlendirme (redirect)** → res.redirect("/dashboard")
* **Basit text / status döndürme** → res.send("Merhaba"), res.status(404).send("Bulunamadı")

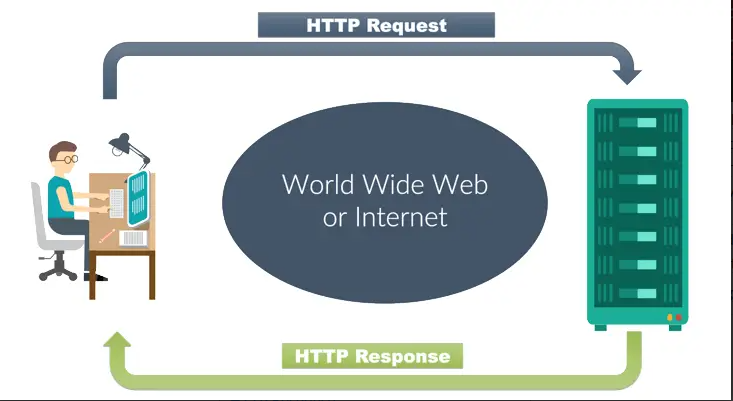
📌 örnekte:

**res.redirect('/dashboard')**

Başarılı girişte kullanıcıyı /dashboard sayfasına yönlendiriyor.

**res.render('login', { error: isValid })**

Eğer hata varsa login sayfasını tekrar gösteriyor, error değişkenini sayfaya gönderiyor.



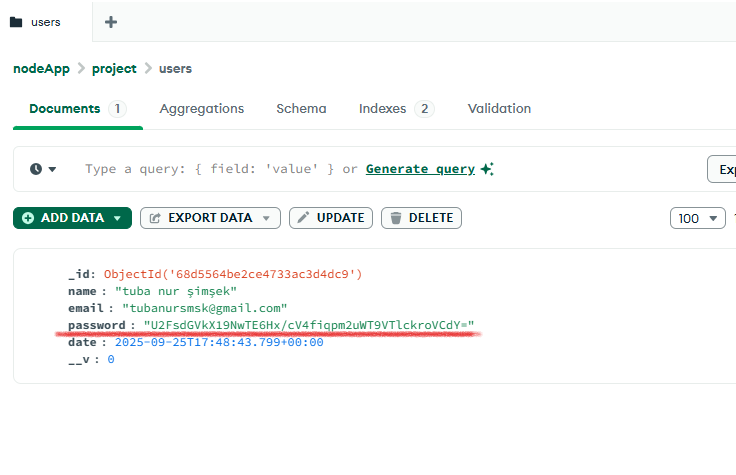
✅ Özetle:

* req = Kullanıcıdan gelen istek (data, headers, parametreler).
* res = Sunucudan giden cevap (html, json, redirect, status).

Days\_69 Node.js MVC projesinde şifrede kriptolojik yapılar (MD5hash) – http session

**Terminalden indirilenler:**

* npm i crypto-js
* npm i @types/crypto-js
* npm install express-session
* npm i --sav-dev @types/express-session



**🔑 Kriptolojik Yapılar ve MD5 Hash**

**1. Şifreleme (Encryption)**

* **AES (Advanced Encryption Standard)** gibi algoritmalar **simetrik şifreleme** yapar.
* Şifrelenmiş veriler (ciphertext) tekrar **aynı anahtar** ile çözülüp (decrypt) geri okunabilir.
* Kullanıcı girişlerinde **şifreleme genellikle tercih edilmez**, çünkü anahtar sızarsa tüm şifreler açığa çıkar.

**2. Hash Fonksiyonları**

* **MD5, SHA-1, SHA-256** gibi algoritmalar **tek yönlü (one-way)** fonksiyonlardır.
* Hash edilen veri tekrar geri çevrilemez.
* Amaç: aynı girdi her zaman aynı hash çıktısını üretir.
* Kullanıcı şifreleri genelde **hash edilerek** saklanır.
  + Kullanıcı giriş yaparken girilen şifre hash edilir ve veritabanındaki hash ile karşılaştırılır.

**3. MD5 Özellikleri**

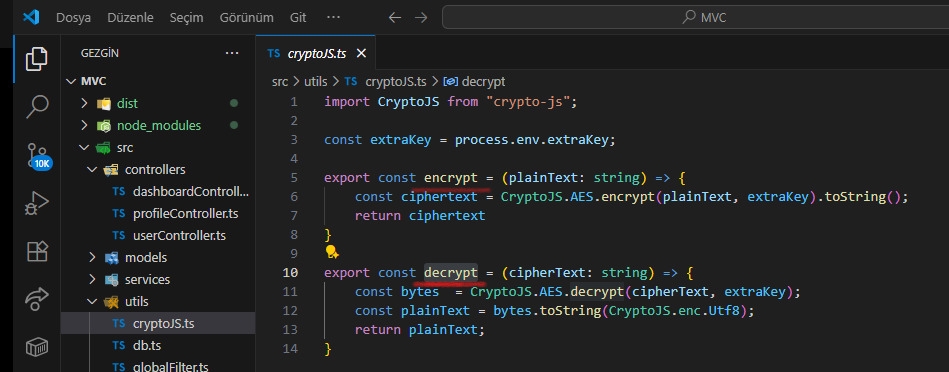
* **128-bit (32 karakterlik)** sabit uzunlukta çıktı üretir.
* Çok hızlıdır ama bu da **güvenlik açığıdır** → kolay brute-force ve rainbow table saldırılarına açıktır.
* Günümüzde **SHA-256 veya bcrypt, scrypt, Argon2** gibi algoritmalar tercih edilir.
* Ders tekrarı için örnek MD5:
* "123456" → e10adc3949ba59abbe56e057f20f883e

**4. Şifre Saklama Stratejisi**

* **Hash + Salt** kullanılmalıdır.
  + salt: her kullanıcıya özel eklenen rastgele veri.
  + Aynı şifreye sahip kullanıcıların veritabanında aynı hash değerine sahip olması önlenir.

**5. Projedeki Durum**

* Şu an cryptojs.ts dosyanda AES kullanıyorsun → şifre geri çözülebilir.
* Daha güvenli yöntem: **CryptoJS.MD5** veya Node.js bcrypt gibi hash fonksiyonlarını kullanmak.

****

**🔐 Encrypt / Decrypt Yapısı**

**1. Encrypt (Şifreleme)**

* **Amaç:** Düz metni (plainText → şifre, kredi kartı numarası vs.) okunamaz hale getirmek.
* Kullanılan yöntem: **AES (Advanced Encryption Standard)**
* AES **simetrik** algoritmadır → aynı anahtar hem şifreleme hem çözme için kullanılır.

Yazılan kodda:

**export const encrypt = (plainText: string) => {**

**const ciphertext = CryptoJS.AES.encrypt(plainText, extraKey).toString();**

**return ciphertext**

**}**

* + plainText: kullanıcıdan gelen düz veri (ör: şifre).
  + extraKey: gizli anahtar (server tarafında .env dosyasında saklanır).
  + ciphertext: şifrelenmiş, anlamı olmayan metin → veritabanına kaydedilir.

📌 Örnek:

**"123456" → U2FsdGVkX19NvTE6Hx/cV4fiqpm2uWT9VTlckroVCdY=**

**2. Decrypt (Şifre Çözme)**

* **Amaç:** Daha önce şifrelenmiş veriyi tekrar düz metne çevirmek.
* Yalnızca doğru **anahtar (extraKey)** ile çözülebilir.

📌 Örnek:

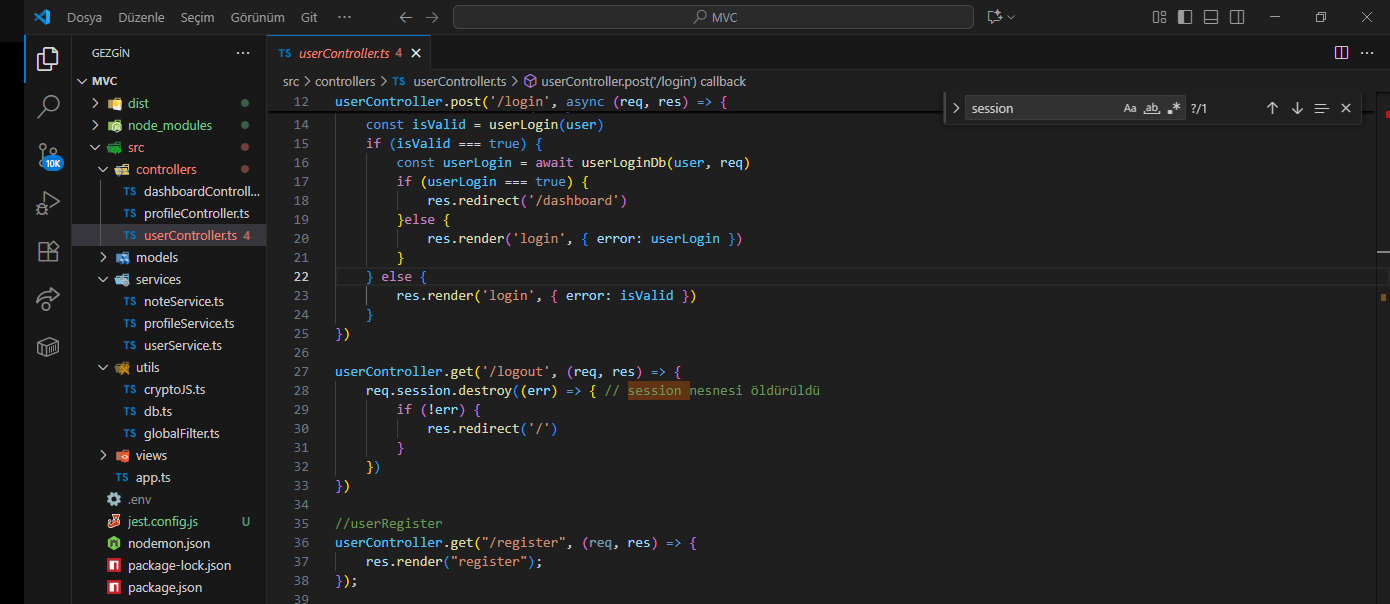
**"U2FsdGVkX19NvTE6Hx/cV4fiqpm2uWT9VTlckroVCdY=" → "123456"**

**3. Özet Mantık**

* **Encrypt:** PlainText + Anahtar → CipherText
* **Decrypt:** CipherText + Anahtar → PlainText

**4. Notlar**

* AES güçlü bir algoritmadır ama **şifreler için tek başına kullanılmaz** çünkü şifreyi tekrar çözmek mümkün olur.
* Şifre saklama için genelde **hash (MD5, SHA-256, bcrypt)** tercih edilir.
* Ama **encrypt/decrypt** yapısı finansal veriler, özel mesajlar gibi **mutlaka geri okunması gereken bilgilerde** kullanılır.



**🗂 HTTP Session Mantığı**

**1. HTTP’nin Özelliği**

* **HTTP protokolü stateless’tir.**  
  Yani her istek (request) birbirinden bağımsızdır, kullanıcı kimliğini hatırlamaz.
* Kullanıcı giriş yaptıktan sonra, sonraki sayfalarda **kimin giriş yaptığını hatırlamak için** ekstra bir mekanizma gerekir.
* İşte bu noktada **session** devreye girer.

**2. Session Nedir?**

* **Session**, kullanıcıya özel olarak sunucuda tutulan bir oturum bilgisidir.
* Kullanıcı giriş yaptığında:
  + Sunucuda bir **session objesi** oluşturulur.
  + Kullanıcıya **session ID** atanır (genelde cookie içinde tarayıcıya gönderilir).
  + Her yeni istekte bu ID sunucuya gönderilerek kullanıcı tanınır.
* Çıkış yapıldığında (req.session.destroy) oturum silinir, kullanıcı artık tanınmaz.

**3. Kod Mantığı**

****

**🔹 Logout İşlemi**

userController.get('/logout', (req, res) => {

req.session.destroy((err) => { // session öldürülür

if (!err) {

res.redirect('/')

}

})

})

* Kullanıcının session bilgisi **sunucudan silinir**.
* Tarayıcıdaki cookie geçersiz hale gelir.
* Kullanıcı tekrar giriş yapmak zorunda kalır.

**4. Session Akışı (Özet)**

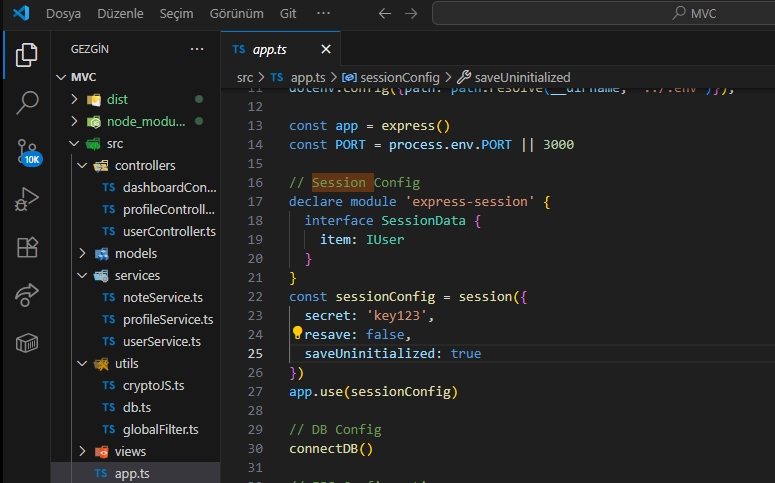
1. Kullanıcı → /login sayfasına form gönderir.
2. Sunucu → DB’den kontrol eder, doğruysa session oluşturur.
3. Tarayıcıya session ID gönderilir (cookie içine yazılır).
4. Kullanıcı /dashboard gibi sayfalara gittiğinde, session ID ile kimliği doğrulanır.
5. Çıkış (/logout) yapıldığında session yok edilir.

**5. Not**

* req.session nesnesi kullanıcıya özeldir.
* Session bilgisi **server tarafında** tutulur, client yalnızca ID’yi taşır.
* Session süresi (maxAge) ayarlanabilir.

📌 **Kısaca:**

* **Login:** req.session.user = user → oturum başladı.
* **Her istekte:** session ID → kullanıcı tanınır.
* **Logout:** req.session.destroy() → oturum bitti.



**⚙️ Express Session Config Mantığı**

**1. express-session Modülü**

* Express uygulamalarında session yönetmek için kullanılır.
* Kullanıcı giriş yaptığında, oturum bilgisini server tarafında tutar.
* Kullanıcıya özel bir **session ID** üretir ve bunu **cookie** olarak tarayıcıya gönderir.

**2. TypeScript için Session Tipi**

**declare module 'express-session' {**

**interface SessionData {**

**item: IUser**

**}**

**}**

* Normalde req.session içine her şey atılabilir ama **TypeScript hata verir** (tip belirsiz).
* Bu declare module ile SessionData arayüzünü genişletiyoruz.
* Artık req.session.item yazdığında **IUser tipinde** olduğunu bilir.

📌 Örnek:

**req.session.item = { id: 1, name: "Tuba", email: "x@gmail.com" }**

**3. Session Config Ayarları**

**const sessionConfig = session({**

**secret: 'key123',**

**resave: false,**

**saveUninitialized: true**

**})**

**app.use(sessionConfig)**

* **secret**:
  + Session ID’yi imzalamak için kullanılan gizli anahtar.
  + Güvenlik için .env dosyasına taşınmalı.
* **resave: false**
  + Eğer session değişmemişse her istekte yeniden kaydetme.
  + Performansı artırır.
* **saveUninitialized: true**
  + Yeni ama içi boş session’ları da kaydet.
  + Genelde kullanıcı giriş yapmadan önce de cookie oluşur.
* **app.use(sessionConfig)**
  + Middleware olarak uygulamaya eklenir.
  + Bundan sonra her req içinde req.session kullanılabilir.

**4. Akış Mantığı**

1. Kullanıcı giriş yaptı → req.session.item = user
2. Tarayıcıya session ID gönderildi.
3. Sonraki isteklerde session ID geri gelir → sunucu req.session.item üzerinden kullanıcıyı tanır.
4. Çıkış yapıldığında req.session.destroy() ile session silinir.

📌 **Kısaca Not:**

* express-session → oturumları yönetir.
* secret → güvenlik için zorunlu.
* resave, saveUninitialized → session’un DB’de veya memory’de nasıl saklanacağını kontrol eder.
* declare module → TypeScript ile req.session içine tip desteği kazandırır.

Days\_70: Node.js MVC projesinde middleware kullanımı – noteadd kısmı aktifleşrilidi

**⚙️ Middleware Nedir?**

* **Middleware**, Express.js uygulamasında **istek (request) ve yanıt (response) arasına giren fonksiyonlardır.**
* Yani her gelen istekte çalışır ve şu işleri yapabilir:
  1. İsteği (req) değiştirmek
  2. Yanıtı (res) değiştirmek
  3. Kontroller yapmak (yetki, doğrulama, hata yönetimi)
  4. İsteğin devam edip etmeyeceğine karar vermek

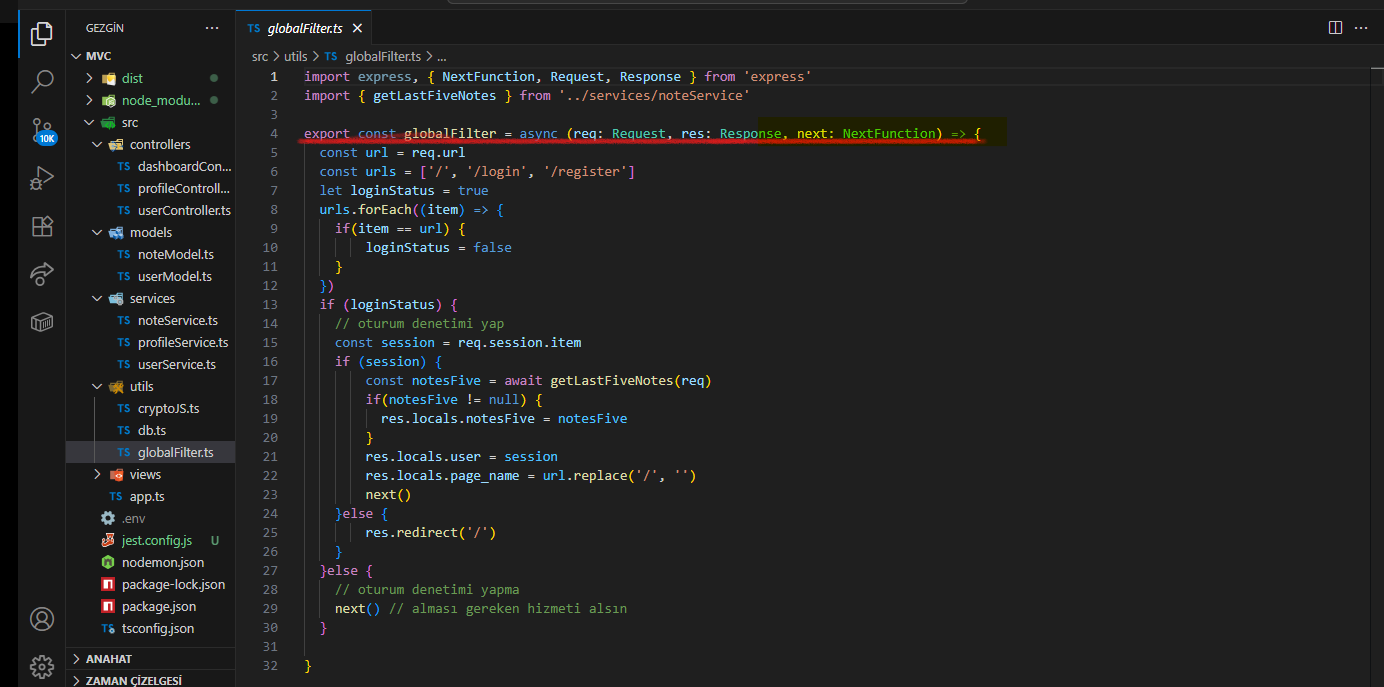
📌 Bir middleware fonksiyonu genellikle şu imzaya sahiptir:

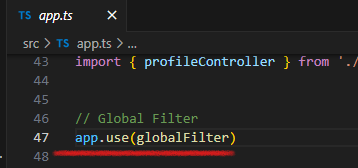
**(req, res, next) => { ... }**

* req → İstek nesnesi
* res → Yanıt nesnesi
* next() → Bir sonraki middleware veya route’a geçiş

**📝 Not Özet**

* Middleware = request–response arasına giren fonksiyonlar.
* next() → zincirde bir sonraki middleware’i çalıştırır.
* Kullanım alanları: loglama, doğrulama, yetkilendirme, hata yakalama.
* Session ile birlikte **“kimlik kontrolü” middleware’leri** çok yaygındır.



****

**1. Fonksiyon Tanımı**

**export const globalFilter = async (req: Request, res: Response, next: NextFunction) => {**

* req, res, next parametrelerini alıyor.
* async olduğu için içinde await kullanabiliyorsun.

**2. URL Kontrolü**

**const url = req.url**

**const urls = ['/', '/login', '/register']**

* İstek yapılan adresi alıyor (req.url).
* Eğer istek **login, register veya anasayfa** ise → **oturum kontrolü yapılmayacak**.

**3. Login Status Kontrolü**

**let loginStatus = true**

**urls.forEach((item) => {**

**if (item == url) loginStatus = false })**

* Eğer istek /login veya /register ise → loginStatus = false
* Yani giriş/ kayıt sayfalarında session zorunluluğu yok.

**4. Session Doğrulama**



**Açıklama:**

* Eğer loginStatus = true (yani kullanıcı login sayfasında değilse):
  + req.session.item kontrol ediliyor → kullanıcı giriş yapmış mı?
  + Eğer giriş yapmışsa:
    - getLastFiveNotes çağırılıyor → son 5 notu çekip res.locals içine atıyor.
    - res.locals.user = session → oturumdaki kullanıcı bilgisini view’e gönderiyor.
    - res.locals.page\_name → URL’den sayfa adını çıkarıp view’e iletiyor.
    - next() → middleware zincirinde devam et.
  + Eğer giriş yapmamışsa → / adresine yönlendiriliyor.
* Eğer loginStatus = false (yani login/register sayfasındaysa):
  + Hiç session kontrolü yapılmadan next() çağrılır.

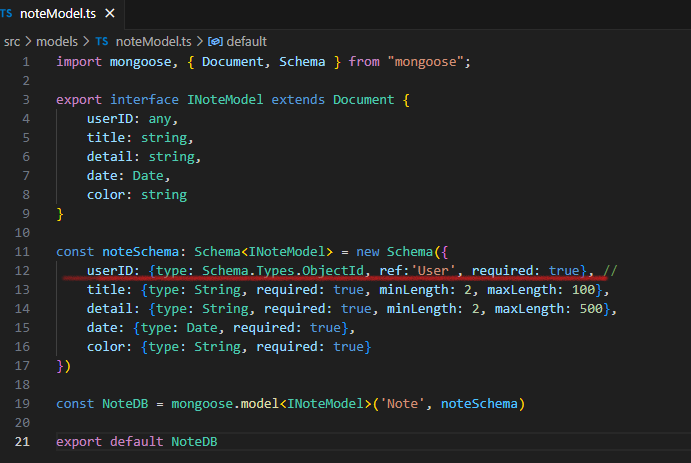
**📖 Middleware Çalışma Akışı**

1. Kullanıcı bir sayfaya istek atıyor.
2. globalFilter çalışıyor.
3. Eğer sayfa /login veya /register ise → serbest bırakılıyor.
4. Diğer sayfalarda session kontrolü yapılıyor.
   * Session varsa → kullanıcı bilgisi + not bilgisi view’e taşınıyor.
   * Session yoksa → / sayfasına redirect.

**📝 Özet Not**

* Bu middleware **global güvenlik filtresi** gibi çalışıyor. Amaç:
  + Kullanıcı giriş yapmadan /dashboard gibi sayfalara girmesin.
  + Giriş yaptıysa session + son notlar otomatik yüklensin.
* res.locals → view (EJS, Pug, Handlebars) tarafına veri göndermenin yolu.

MongoDB üzerinde Note (not) verilerini tutmak için bir Mongoose model dosyası



**1. Gerekli importlar**

**import mongoose, { Document, Schema } from "mongoose";**

* mongoose: MongoDB ile haberleşmek için kullanılan ODM (Object Data Modeling) kütüphanesi.
* Document: Mongoose modellerinin, MongoDB’deki **bir dökümanı temsil ettiğini** belirtmek için kullanılır.
* Schema: MongoDB koleksiyonundaki dokümanların **yapısını tanımlamak için şablon**.

**2. Arayüz (TypeScript interface)**

**export interface INoteModel extends Document {**

**userID: any,**

**title: string,**

**detail: string,**

**date: Date,**

**color: string**

**}**

* INoteModel: TypeScript tarafında **Note dokümanlarının tipini** tanımlayan interface.
* extends Document: Mongoose’un sunduğu MongoDB döküman özelliklerini de içeriyor.
* Alanlar:
  + userID: Notu kimin eklediğini belirtir, **User tablosuna referans** olacak.
  + title: Notun başlığı.
  + detail: Notun içeriği.
  + date: Notun tarihi.
  + color: Notun rengi (örneğin sarı, kırmızı gibi).

**3. Schema tanımı**

**const noteSchema: Schema<INoteModel> = new Schema({**

**userID: {type: Schema.Types.ObjectId, ref:'User', required: true},**

**title: {type: String, required: true, minLength: 2, maxLength: 100},**

**detail: {type: String, required: true, minLength: 2, maxLength: 500},**

**date: {type: Date, required: true},**

**color: {type: String, required: true}**

**})**

* userID:
  + Türü: ObjectId (MongoDB’deki başka bir dokümana referans vermek için).
  + ref: 'User': Bu alanın **User koleksiyonundaki bir kullanıcıya bağlı** olduğunu gösteriyor.
  + required: true: Boş bırakılamaz.
* title: String türünde, 2–100 karakter arasında, zorunlu.
* detail: String türünde, 2–500 karakter arasında, zorunlu.
* date: Tarih bilgisi. (Notun eklenme veya planlanma tarihi olabilir.)
* color: String, zorunlu. Kullanıcı notu renklendirmek için kullanabilir.

**4. Model tanımı**

**const NoteDB = mongoose.model<INoteModel>('Note', noteSchema)**

* Burada NoteDB isminde bir **model** oluşturuluyor.
* 'Note': MongoDB’de oluşturulacak koleksiyonun adı (otomatik olarak çoğul hale gelir → notes).
* noteSchema: Koleksiyonun hangi yapıda olacağını belirliyor.
* INoteModel: TypeScript tip güvenliği için kullanılıyor.

**5. Export**

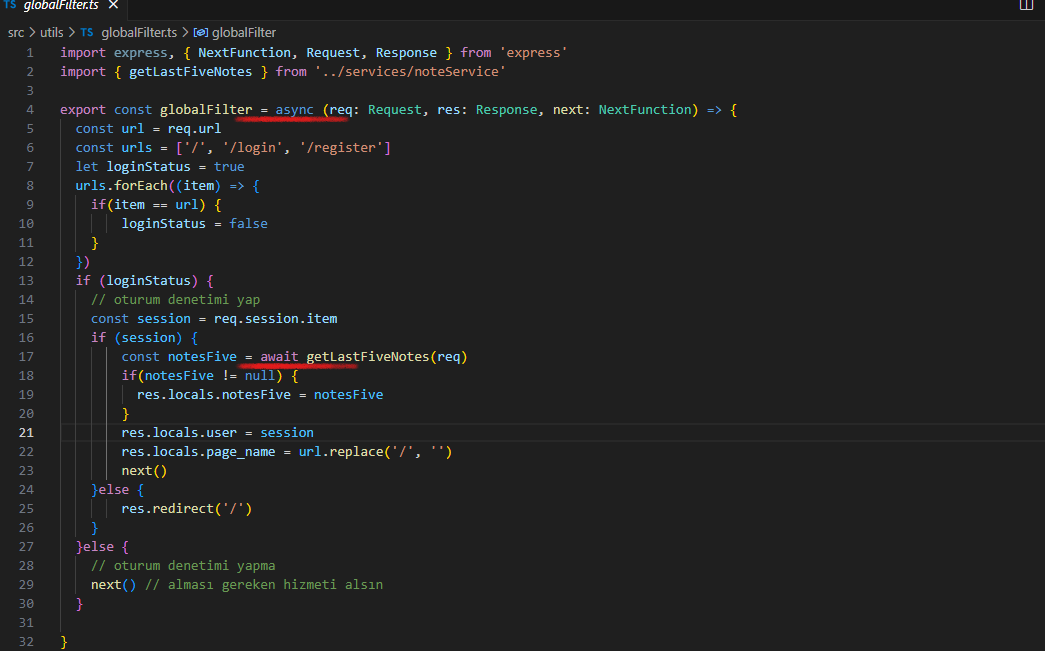
**export default NoteDB**

* Bu model NoteDB adıyla dışarıya aktarılıyor.
* Başka dosyalarda import NoteDB from './noteModel' şeklinde kullanılarak veritabanında not oluşturma, güncelleme, silme ve arama işlemleri yapılabilir.

✅ **Özetle:**  
Bu dosya, MongoDB’de **notları (title, detail, date, color, userID ile)** saklayacak olan bir şema (schema) ve model tanımlıyor.

* noteSchema: Notların yapısını belirler.
* NoteDB: O yapıya göre veritabanı işlemleri yapmamızı sağlar.

async / await mantığını kavradığında günlük hayattaki bir iş sırasına çok benziyor hem **teknik** hem de **günlük hayat örneğiyle** açıklayalım:



**🔹 1. Teknik olarak**

* async → Fonksiyonun **asenkron** çalışacağını söyler. Yani bu fonksiyonun içinde **zaman alan işler** olabilir (örneğin: veritabanından veri çekmek, dosya okumak, API’den cevap beklemek).
* await → “Dur bakalım, bu işlem bitene kadar bekle, ondan sonra devam et” demektir.

Eğer await koymazsan, işlem bitmeden sonraki satır çalışır → sonuç **boş**, **undefined** veya **hatalı** olabilir.

**🔹 2. Günlük hayattan örnek**

Diyelim ki **kafe** işletiyorsun.

**await OLMADAN:**

* Garsona "bir kahve getir" diyorsun.
* Garson mutfağa gidiyor ama sen beklemiyorsun → başka işlere geçiyorsun (örneğin faturayı ödemeye çalışıyorsun).
* Ama kahve ortada yokken faturayı ödemek garip olur 😅

**await İLE:**

* Garsona "bir kahve getir" diyorsun.
* **await** diyorsun → yani kahve gelene kadar bekliyorsun ☕
* Kahve geldiğinde, “tamam şimdi faturayı ödeyebilirim” diyorsun.

Yani:

* async → Bu kafe işinde beklemeli işler olabilir (kahve hazırlamak).
* await → Kahve hazır olana kadar **bekle**.

**🔹 3. Senin kodunda (getLastFiveNotes)**

**const notesFive = await getLastFiveNotes(req)**

* Burada getLastFiveNotes(req) → veritabanına gidiyor, "bu kullanıcının son 5 notunu getir" diyor.
* Bu işlem **milisaniyeler veya saniyeler sürebilir**.
* await koymazsan kod hemen sonraki satıra geçer → notesFive boş olur.

Ama await sayesinde → "dur bakalım, notlar gelene kadar bekle" diyorsun.  
Notlar geldikten sonra res.locals.notesFive = notesFive diyebiliyorsun.

✅ **Özet:**

* async → Bu fonksiyonda beklemeli işler olabilir.
* await → Beklemeden sonraki adıma geçme.
* Günlük hayatta → "kahven hazır olmadan masadan kalkma" gibi düşünebilirsin.

Days\_71: Node.js MVC projesinde - getAllNotes() tasarımı – noteDelete() – edit – Profile sayfaları

Days\_72: Node.js genel tekrar

* node.js projesi kurarken bu eklentiyi **npm install --sav-dev typescript** ekledik. Baktığımızda js, daha esenk ve geniş kullanımı olan bir dil. Ts’in ise ona göre tip güvenliği yüksek. Projelerde neden dosyalar ts uzantılı?

**🔹 1. Node.js ve JavaScript (JS) tarafı**

* **JavaScript** dinamik bir dil → yani bir değişkenin tipi çalışma zamanında belli olur.
* Örneğin:

**let age = 25**

**age = "yirmi beş"** // JS buna izin verir 😅

Bunun sonucu:

* Hataları **kod çalışırken** fark edersin.
* Büyük projelerde bu çok risklidir → çünkü yüzlerce dosya birbirine bağlıdır, tip uyumsuzluğu yüzünden uygulama patlayabilir.
* Gelişen kod yapısı ile bu sorunlarında önüne geçilebiliyor.

**🔹 2. TypeScript (TS) farkı**

* **TypeScript**, JavaScript’in **tip güvenliği eklenmiş versiyonu** gibi çalışır.
* Yani sen kod yazarken, “bu değişkenin tipi şu olacak” diyebilirsin.

let age: number = 25

age = "yirmi beş" // ❌ Hata: string, number'a atanamaz

Avantajları:

* Daha **erken hata yakalama** (derleme sırasında).
* **IDE (VSCode) desteği** → otomatik tamamlama, daha az hata.
* Büyük projelerde **bakımı kolaylaştırır**.

**🔹 3. Neden --save-dev typescript?**

* npm install --save-dev typescript komutu → projene **TypeScript derleyicisini** ekler.
* Çünkü Node.js aslında **TS’yi çalıştırmaz**.
* TS kodunu önce **JS’ye çevirmek** (transpile) gerekir.

Örneğin:

* Sen app.ts yazarsın.
* TS derleyicisi (tsc) → bunu app.js dosyasına çevirir.
* Node.js → sadece app.js dosyasını çalıştırır.

Yani typescript paketi bir **derleyici**dir (compiler), runtime’da gerekmez, sadece geliştirme sırasında kullanılır. O yüzden --save-dev ile eklenir.

**🔹 4. Günlük hayattan örnek**

Bunu şöyle düşünebilirsin:

* **JS** = serbest şekilde yazı yazmak. Noktalama, dil bilgisi kuralı zorunlu değil. (Herkes farklı anlayabilir, hata çıkabilir)
* **TS** = yazıyı Word’de yazmak. Dil bilgisi, imla kontrolü, yanlış kelime altı kırmızı çizgiyle gösterilir.

Sonuç:

* TS ile yazmak biraz daha katı kurallar ister, ama büyük projelerde hata oranını **çok ciddi azaltır**.

✅ Özet:

* typescript paketi → TS dosyalarını JS’ye çevirmek için derleyicidir.
* --save-dev → sadece geliştirme aşamasında lazım olduğu için.
* JS esnek ama riskli → TS tip güvenliği sayesinde daha güvenli, hataları erken yakalayan bir ortam sunar.

🡪 **Node.js Controller neden gereklidir amacı nedir?**

**🔹 MVC Nedir?**

* **M** → **Model** (Veri & veritabanı katmanı)
* **V** → **View** (Kullanıcıya görünen kısım)
* **C** → **Controller** (Aracı / yönetici)

Bu üç katman birbiriyle konuşarak **uygulamanın düzenli çalışmasını sağlar**.

**🔸 1. Controller Katmanı**

**📌 Görevi:**

* Kullanıcıdan gelen **istekleri (request)** karşılar.
* Doğru **service / model** çağrılarını yapar.
* Gelen sonuçları **View**’a gönderir.
* Kısaca: Kullanıcı ile sistem arasındaki **köprü**dür.

**📌 İşleyişi:**

1. Kullanıcı bir butona tıklıyor / form gönderiyor → **request** gelir.
2. Bu request önce **Controller**’a düşer.
3. Controller bu isteği işler:
   * Gerekirse **Model** üzerinden veritabanına erişir.
   * **Business logic (iş kuralları)** varsa Service katmanını kullanır.
4. Sonuç olarak kullanıcıya **response** döner (ör. bir sayfa, JSON verisi, hata mesajı vs.).

**📌 Örnek:**

**// userController.ts**

**userController.post('/login', async (req, res) => {**

**const user = req.body**

**const isValid = userLogin(user) // Service çağrısı**

**if (isValid) {**

**res.redirect('/dashboard') // View’e yönlendirme**

**} else {**

**res.render('login', { error: "Hatalı giriş!" }) // View'e hata gönder**

**}**

**})**

**🔸 Günlük Hayattan Örnek**

Bir **restoran** hayal et:

* **Müşteri** → Kullanıcı (Request yapıyor).
* **Garson (Controller)** → Siparişi alır, mutfağa iletir, yemeği müşteriye getirir.
* **Aşçı (Model)** → Verileri işleyen kısım → mutfak = veritabanı.
* **Servis / Menü kuralları** → İş mantığı (Service).
* **Müşterinin önüne gelen tabak (View)** → Sonuç (response).

👉 Yani garson = **Controller**.  
Müşteri doğrudan mutfağa girip aşçıya sipariş vermez, garson aracıdır.

✅ **Controller’ın amacı:**

* Kullanıcı isteğini alır,
* Doğru katmanlara yönlendirir,
* Sonucu kullanıcıya iletir.

**🔸 2. Model Katmanı**

**📌 Görevi:**

* Uygulamanın **veri katmanı**dır.
* **Veritabanındaki tablolar / koleksiyonlar** için şemaları (schema) tanımlar.
* CRUD işlemlerini (Create – Read – Update – Delete) yönetir.
* Verilerin nasıl tutulacağını, hangi kurallara uyacağını belirler.

**📌 İşleyişi:**

1. Bir Schema oluşturulur → veritabanında hangi alanlar olacak, türleri ne olacak.
2. Bu şemaya göre bir Model çıkarılır.
3. Controller / Service katmanları bu Model’i kullanarak veritabanıyla konuşur.

…..

const NoteDB = mongoose.model<INoteModel>('Note', noteSchema)

export default NoteDB

👉 Burada noteSchema, veritabanına kaydedilecek notların kurallarını belirliyor.  
👉 NoteDB ise bu şemayı kullanarak MongoDB’de **koleksiyon** (tablo) oluşturan model.

**🔸 Günlük Hayattan Örnek**

Restoran örneğinden devam edelim:

* **Aşçı (Model)**: Yemeği yapan kişidir.
* Menüde “pizza” varsa → onun bir tarifi, gramajı, malzemeleri vardır (schema).
* Garson siparişi mutfağa ilettiğinde → aşçı tarife uygun şekilde yemeği yapar.

👉 Yani **Model = yemek tarifi + yemeği yapan aşçı**.  
Tarife uymadan malzeme koyamaz. (ör. hamursuz pizza olmaz)  
Aynı şekilde veritabanında “title boş olamaz” gibi kurallar da burada tanımlanır.

✅ **Model’in amacı:**

* Verilerin yapısını tanımlar,
* Veritabanıyla etkileşimi sağlar,
* Verilerin kurallara uygun olmasını garanti eder.

**🔸 3. View Katmanı**

**📌 Görevi:**

* Kullanıcıya gösterilen kısımdır (**arayüz**).
* Controller’dan gelen veriyi alır, görselleştirir.
* HTML, CSS, JS, EJS, Handlebars, Angular, React gibi teknolojilerle oluşturulabilir.
* Kullanıcı burada butona tıklar, form doldurur → bu istekler tekrar Controller’a gider.

**📌 İşleyişi:**

1. Kullanıcı tarayıcıda bir sayfa açar (ör. /dashboard).
2. Controller bu isteği işler, gerekli veriyi Model’den çeker.
3. Controller bu veriyi **View**’a yollar.
4. View bu veriyi kullanıcıya okunabilir/görülebilir bir şekilde sunar.

**📌 Örnek (EJS ile): \*\*\* ejs sayesinde, statik olan html yapısı dinamik hale gelir!**

**<!-- views/dashboard.ejs -->**

**<h1>Merhaba <%= user.username %>!</h1>**

**<ul>**

**<% notesFive.forEach(note => { %>**

**<li style="color: <%= note.color %>">**

**<strong><%= note.title %></strong> - <%= note.detail %>**

**</li>**

**<% }) %>**

**</ul>**

👉 Burada Controller res.render('dashboard', { user, notesFive }) dediğinde,

* user ve notesFive değişkenleri **View**’e aktarılır.
* View de bunları HTML içinde güzel bir şekilde kullanıcıya gösterir.

**🔸 Günlük Hayattan Örnek**

Restoran örneğini bitirelim:

* **Müşteriye gelen tabak (View)**: Hazırlanan yemeğin sunumudur.
* Garson (Controller) mutfaktan yemeği alır, tabağa konmuş şekilde müşteriye getirir.
* Müşteri tabağın içindeki yemeği görür, tadına bakar.

👉 Yani **View = yemeğin masaya gelen hali**.  
Müşteri mutfağı (Model) görmez, sadece önüne gelen tabağı (View) görür.

✅ **View’in amacı:**

* Kullanıcıya verileri okunabilir ve şık bir şekilde sunmak.
* Kullanıcıdan alınan etkileşimi tekrar Controller’a göndermek.

📌 Özet:

* **Model** → Yemek tarifi & aşçı (veri ve kurallar)
* **Controller** → Garson (istekleri alıp yöneten)
* **View** → Masadaki tabak (kullanıcıya görünen sonuç)

**🔹 Service Katmanı Nedir?**

MVC’de aslında **zorunlu bir katman değildir**, ama büyük projelerde **iş mantığını (business logic)** Controller’dan ayırmak için kullanılır.

* **Controller** → sadece istekleri alır, yönlendirir.
* **Model** → sadece veritabanıyla ilgilenir.
* **Service** → iş kuralları ve hesaplamalar burada yapılır.

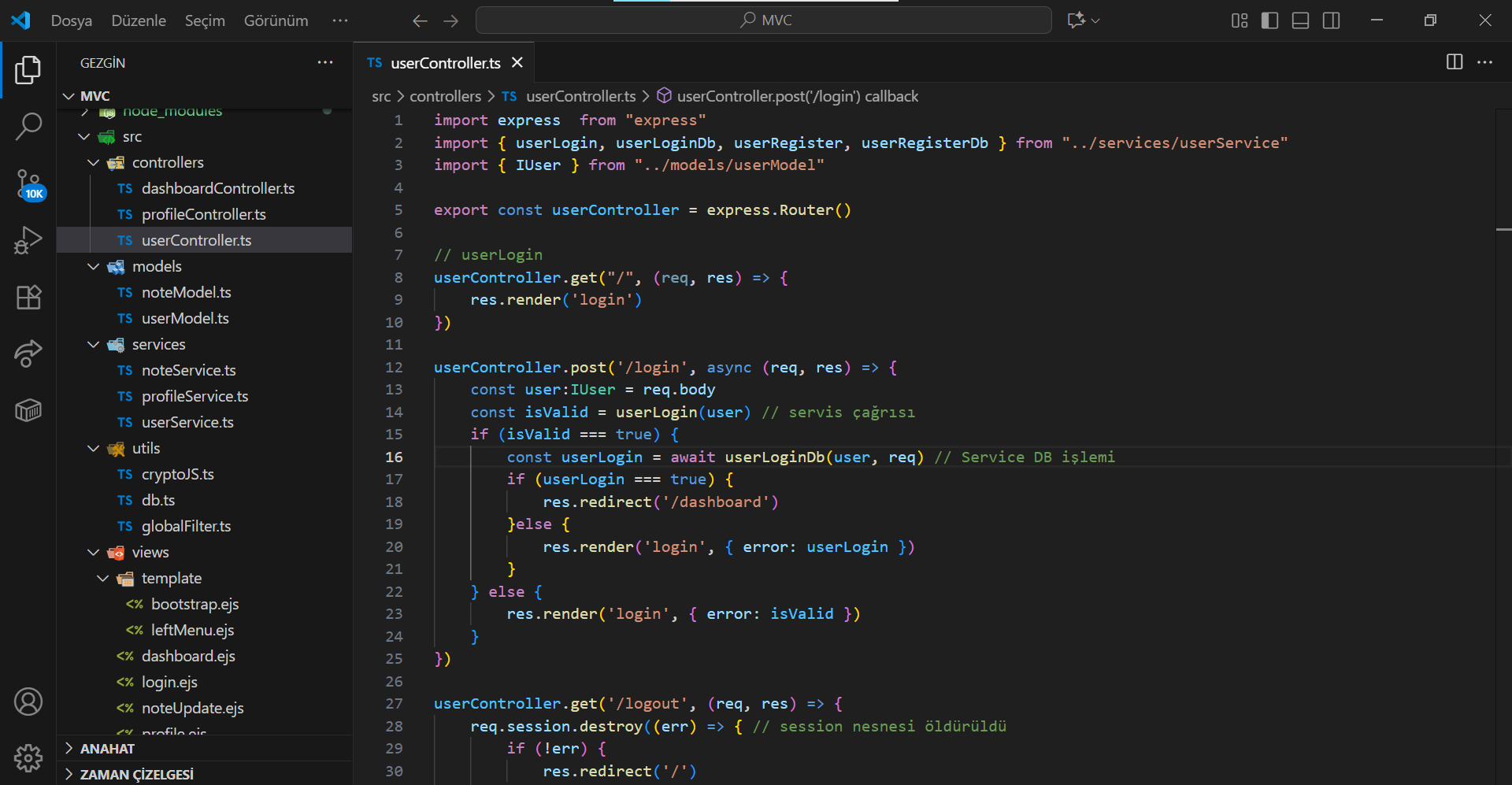
👉 Yani Service = \*\*“mantık katmanı”\*\*dır.

**🔹 Neden Service Kullanılır?**

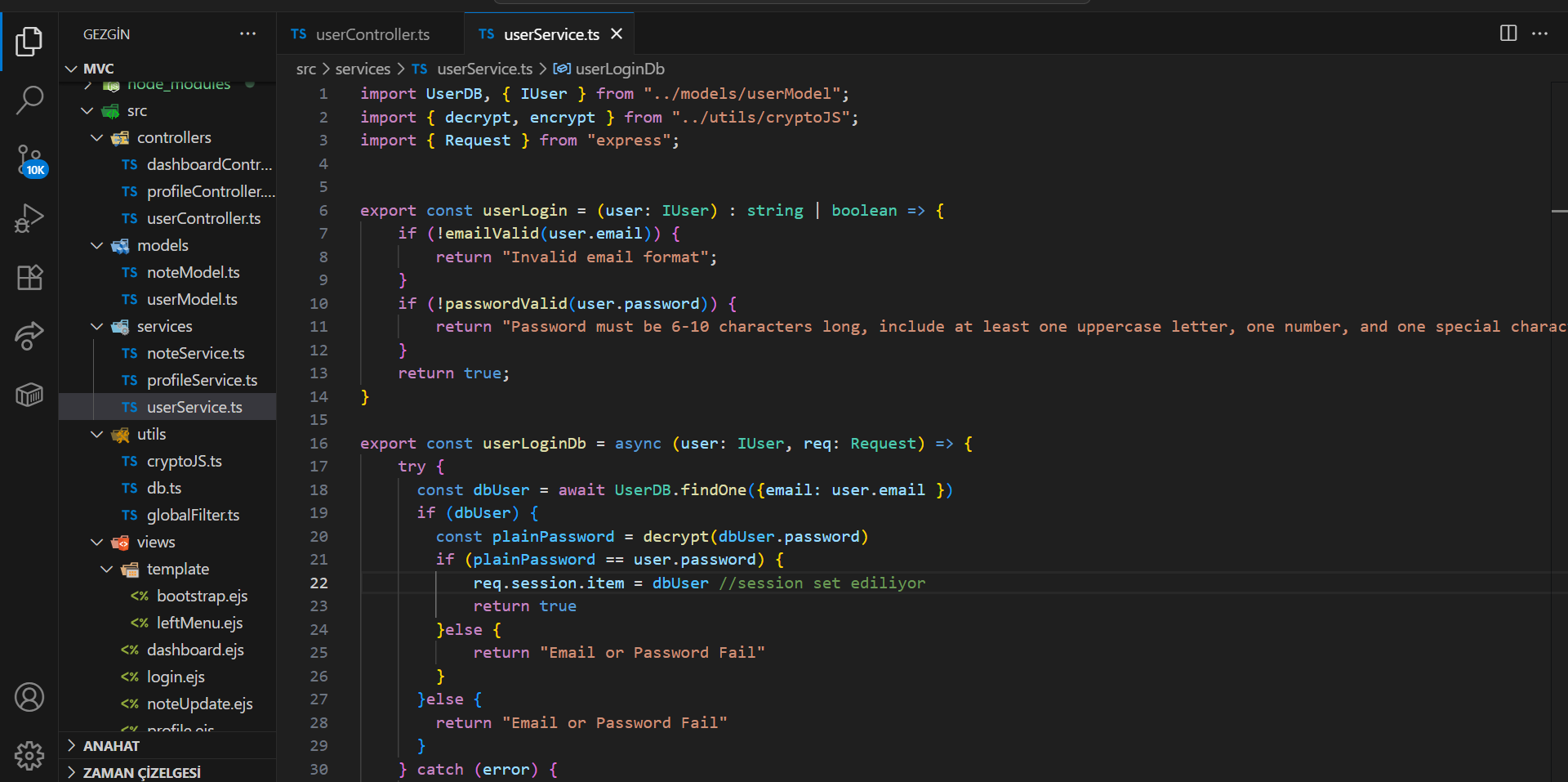
1. **Controller sade kalır** → Controller sadece yönlendirici olur, karmaşık kodlar orada birikmez.
2. **Tekrar kullanılabilirlik** → Aynı iş mantığını farklı Controller’larda tekrar tekrar kullanabilirsin.
3. **Test kolaylığı** → İş mantığını bağımsız olarak test etmek kolaylaşır.
4. **Katmanlı mimariyi düzenler** → MVC yapısı daha temiz olur.

**🔹 Örnek (senin projenle bağlantılı)**

**📌 Controller (sade görev)**



**📌 Service (iş mantığı)**

👉 Burada:

* userLogin → sadece validasyon yapıyor (boş mu, kurallara uygun mu?).
* userLoginDb → veritabanıyla karşılaştırma yapıyor.
* **Controller** → sadece bu servisleri çağırıyor, karmaşık iş mantığını kendisi yapmıyor.

**🔹 Günlük Hayattan Örnek**

Restoran benzetmemize devam edelim 🍽️

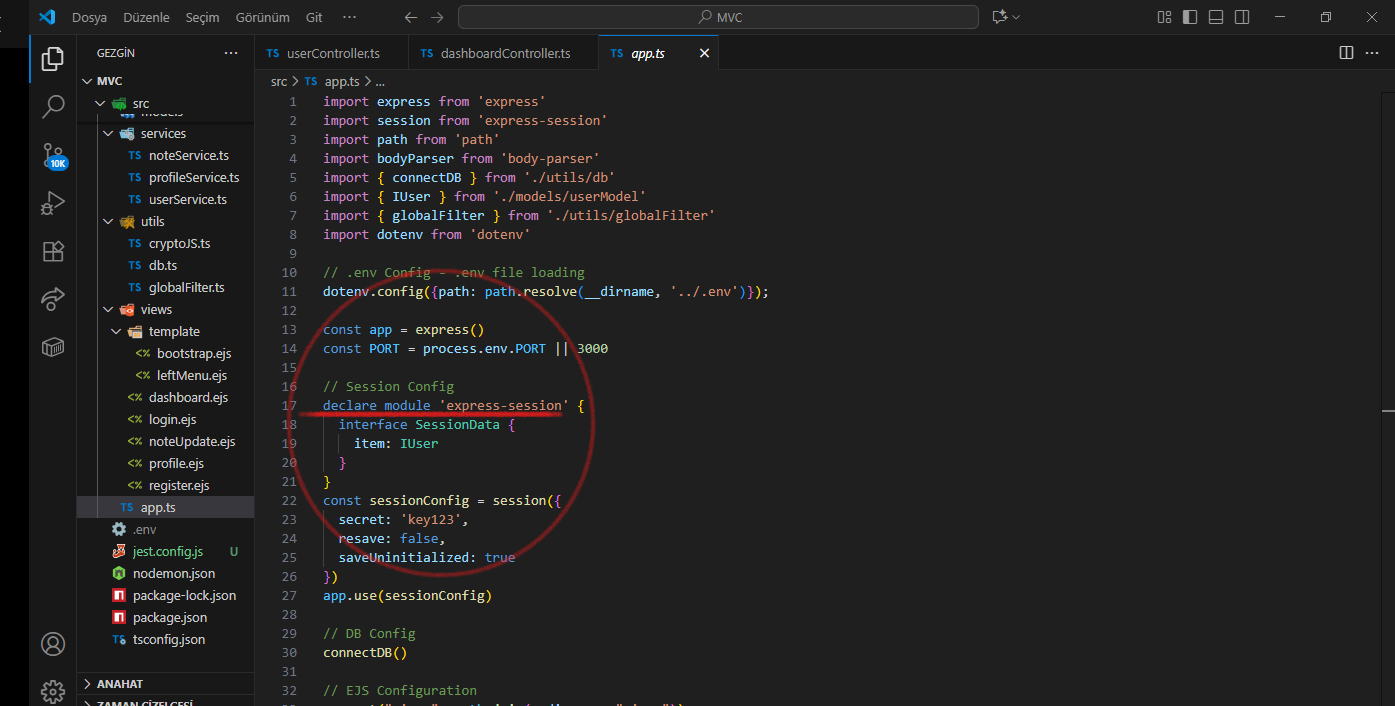
* **Garson (Controller)** → siparişi alır ve “pizza istiyor” der.
* **Aşçı (Model)** → mutfakta pizza yapma bilgisini (tarifi) bilir.
* **Service (Mantık Katmanı)** → “Eğer müşteri vegan ise şu malzemeleri çıkar, eğer çocuk menüsü istiyorsa küçük porsiyon hazırla, eğer indirim kodu varsa %10 düş” gibi **kuralları uygular.**

👉 Böylece garsonun kafası karışmaz, sadece siparişi iletir.

✅ **Service’in amacı:**

* İş mantığını Controller’dan ayırmak,
* Kodun okunabilirliğini ve test edilebilirliğini artırmak,
* Tekrar kullanılabilir fonksiyonlar oluşturmak.

Days\_73: Node.js mvc - .env kullanımı - RestApi projesine giriş

**

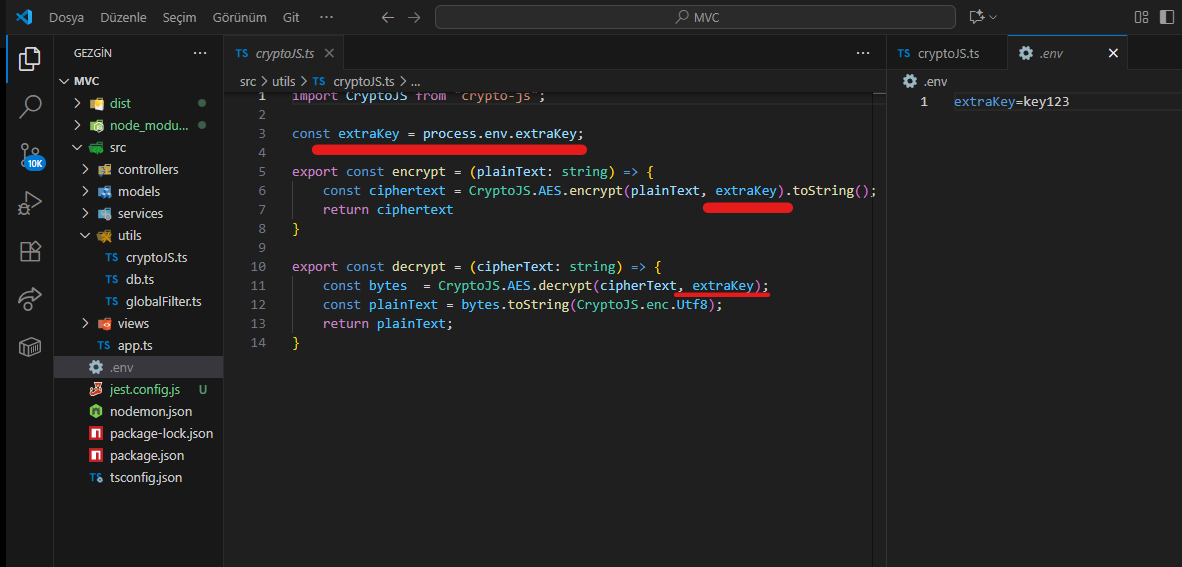
declare terimi neyi ifade eder? Exress session bir kütüphanedir.bu kütüphanenin içine tanımlı olmayan bir değer eklemek istediğimizde declare terimini kullanırız. Bu kodda interface ile SessionData objesi eklenmiş.

**📌. env Dosyası Mantığı ve Önemi**

* **npm i dotenv** proje terminalinden indir.

**1. .env dosyası nedir?**

* Projede **ortam değişkenlerini** (environment variables) tutmak için kullanılan dosyadır.
* Yani uygulamanın çalışması için gerekli ama kodun içinde açık yazmamak gereken gizli/konfigürasyon bilgilerini saklar.
* Örn: API key, DB bağlantı bilgisi, şifreler, token secret, özel anahtarlar.



**2. Çalışma mantığı**

* .env dosyasındaki değerler process.env objesine yüklenir.
* Kod içinde process.env.extraKey yazdığında, .env dosyasında tanımlanan değeri alırsın.
* Bu yüklemeyi yapmak için genelde dotenv paketi kullanılır:

**import dotenv from 'dotenv'**

**dotenv.config()**

**3. Kod örneğinde**

**.env dosyası:**

**extraKey=key123**

**crypto.js dosyası:**

**const extraKey = process.env.extraKey;**

👉 Burada extraKey değişkeni "key123" oluyor.  
👉 Yani **AES şifreleme/deşifreleme** bu key üzerinden yapılıyor.

**4. Neden önemli?**

* **Güvenlik:**  
  Şifreleme anahtarı (key123) veya DB şifresi gibi bilgiler kodun içine yazılmaz → GitHub’a push edersen açıkta kalmaz.
* **Ortam ayrımı:**  
  Her ortamda farklı ayar olabilir:
  + Geliştirme (development) → extraKey=dev123
  + Test → extraKey=test456
  + Canlı (production) → extraKey=prod789
* **Esneklik:**  
  Aynı kod her yerde çalışır, sadece .env dosyası değişir.

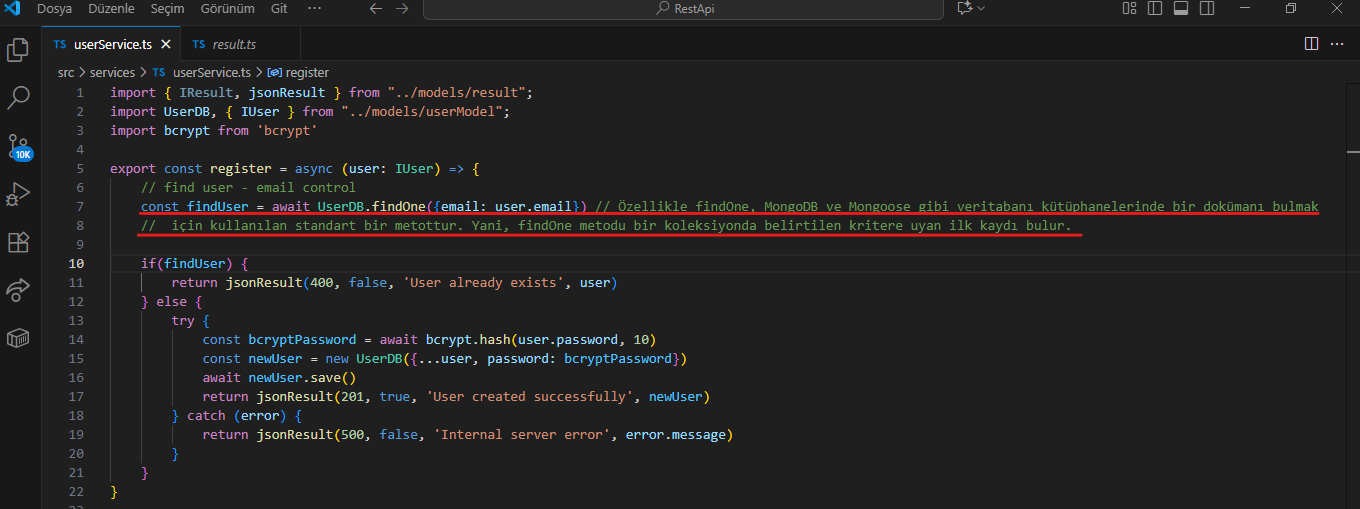
**5. Günlük hayattan örnek 🏦**

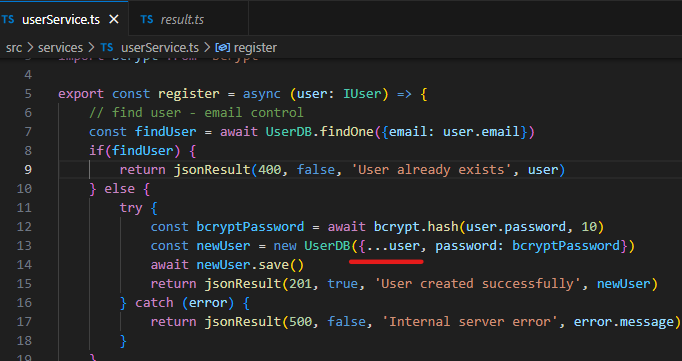
* ATM kartın var → kart numarası herkes görebilir.
* Ama şifre (PIN) gizlidir → bunu kağıda yazıp cüzdanda gezdirmezsin.
* Bunun yerine **kafanda tutarsın** → işte .env dosyası projenin kafasında tuttuğu gizli bilgi gibidir.
* Kodu (kart) kaybetsen bile, .env (şifre) yanında değilse bir işe yaramaz.

👉 Özet Not:

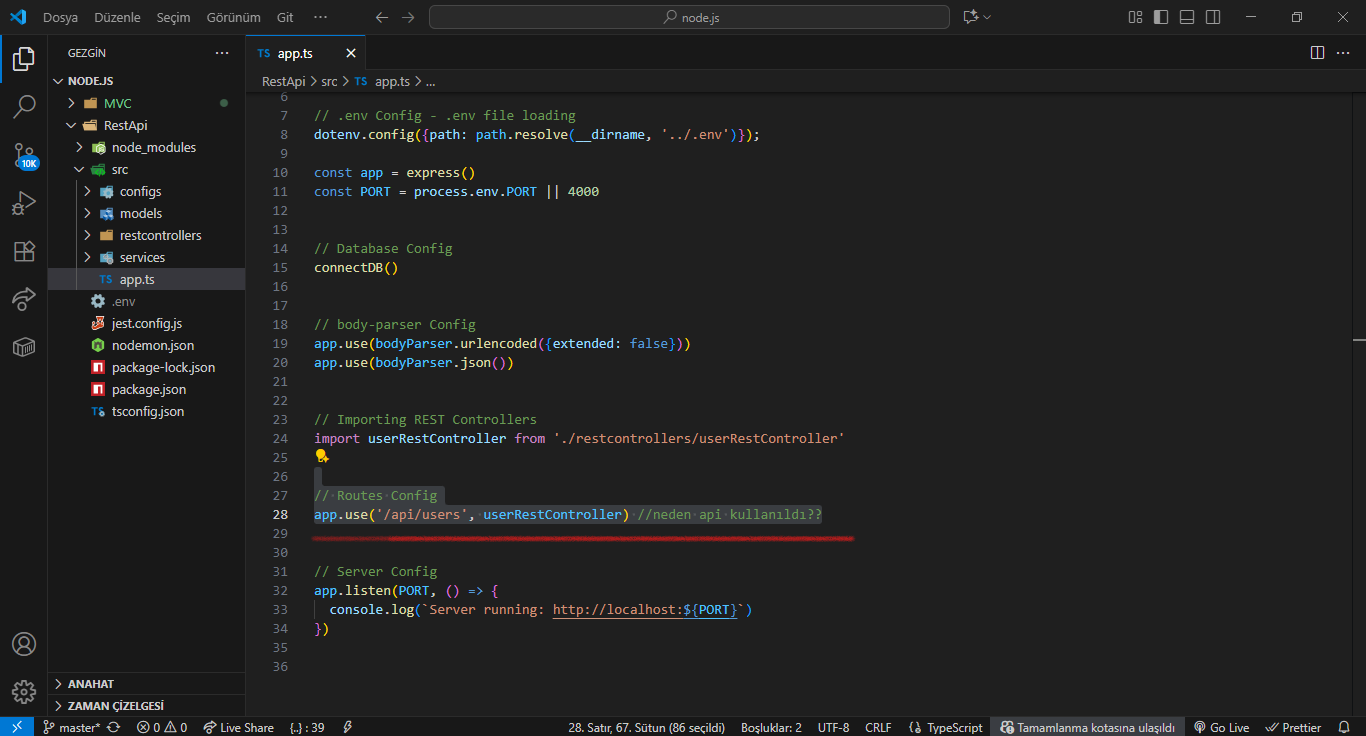
* .env → projede gizli/konfigürasyon bilgilerinin saklandığı dosya
* process.env → bu dosyadan okunan değerleri projede kullanma yolu
* Önemi → güvenlik, esneklik, ortam bağımsızlığı sağlar

Days\_74: RestApi üzerine JWT ekleme





...user ifadesi, JavaScript ve TypeScript'te "spread operator" olarak adlandırılır. Bu operatör, bir nesnenin tüm özelliklerini başka bir nesneye kopyalamak için kullanılır. Kodda {...user, password: bcryptPassword} şeklinde kullanıldığında, user nesnesinin tüm alanları yeni nesneye eklenir ve ardından password alanı hashlenmiş şifre ile güncellenir. Yani, user nesnesinin tüm bilgileri yeni kullanıcıya aktarılır, sadece şifre alanı değiştirilmiş olur. Bu yöntem, nesne oluştururken kodun daha kısa ve okunabilir olmasını sağlar.



Bu kodda /api kullanılmasının nedeni, bu yolun (route) bir web sayfası değil, bir **API (Uygulama Programlama Arayüzü)** uç noktası olduğunu belirtmek için kullanılan yaygın bir standarttır.

Kısacası, bu bir "kural" değil, projenin daha düzenli ve anlaşılır olmasını sağlayan bir "gelenektir".

**Temel Amaç: Ayrım ve Organizasyon**

Bir web uygulamasında genellikle iki tür yol (route) bulunur:

1. **Kullanıcı Arayüzü Yolları (Web Sayfaları):** Tarayıcıda doğrudan kullanıcılara gösterilen HTML, CSS ve JavaScript dosyalarını döndüren yollardır. Örneğin**, http://example.com/users** adresi, kullanıcıların listelendiği bir web sayfasını gösterebilir.
2. **API Yolları (Veri Uç Noktaları):** Ham veri döndüren yollardır. Bu veriler genellikle **JSON** formatındadır ve bir web sayfası tarafından değil, başka bir program tarafından (örneğin bir mobil uygulama, bir JavaScript framework'ü olan React/Vue veya başka bir sunucu) kullanılmak üzere tasarlanmıştır.

**app.use('/api/users', ...)** satırı şunu ifade eder: "/api/users" adresine bir istek geldiğinde, bu isteği userRestController yönetsin. Bu kontrolcü, bir web sayfası (HTML) döndürmek yerine, büyük olasılıkla kullanıcıların listesini JSON formatında bir veri olarak döndürecektir.

**Örnek:**

* http://example.com/users → Kullanıcıları gösteren bir **web sayfası** döner. 🖥️
* http://example.com/api/users → Kullanıcıların **verisini** (JSON formatında) döner. 📦

**/api Ön Ekini Kullanmanın Avantajları**

Bu standardı kullanmanın birçok pratik faydası vardır:

* **Anlaşılırlık:** Kodu okuyan herhangi bir geliştirici, /api ile başlayan bir yolun veri alışverişi için kullanıldığını hemen anlar. Bu, projenin bakımını kolaylaştırır.
* **İsim Çakışmasını Önleme:** Hem kullanıcıların listelendiği bir web sayfası (/users) hem de bu kullanıcıların verisini sağlayan bir API uç noktası (/api/users) oluşturabilirsiniz. /api ön eki olmadan bu iki yolu ayırt etmek zor olurdu.
* **Sürüm Yönetimi (Versioning):** Gelecekte API'nizde büyük değişiklikler yaptığınızda, eski sürümü bozmadan yeni bir sürüm yayınlayabilirsiniz. Örneğin:
  + /api/v1/users (Eski sürüm)
  + /api/v2/users (Yeni sürüm)
* **Sunucu ve Güvenlik Yapılandırması ⚙️:** Web sunucuları (Nginx, Apache gibi) veya API Gateway'ler, gelen istekleri yollarına göre kolayca filtreleyebilir. Örneğin, /api/ ile başlayan tüm isteklere özel kimlik doğrulama (authentication) kuralları veya istek limiti (rate limiting) uygulayabilirsiniz.

Özetle, kodunuzdaki app.use('/api/users', ...) satırı, bu yolun bir veri servisi olduğunu açıkça belirtir ve projenizi daha organize, ölçeklenebilir ve yönetilebilir hale getirir.

Restapi gibi servis uygulmalarında session yerine jwt neden avantajlıdır?

**🔑 Session vs JWT (JSON Web Token)**

**1. Session nasıl çalışır?**

* Kullanıcı giriş yapınca backend bir **session** oluşturur.
* Bu session **sunucuda saklanır** (RAM’de veya Redis gibi bir yerde).
* Kullanıcıya sadece bir sessionId verilir (cookie ile).
* Her istek geldiğinde bu sessionId ile sunucuya bakılır → kullanıcı bilgisi bulunur.

👉 Yani **sunucu taraflı state (durum) tutma** vardır.

**2. JWT nasıl çalışır?**

* Kullanıcı giriş yapınca backend bir **token (JWT)** üretir.
* Bu token kullanıcıya verilir (genelde Authorization: Bearer <token> şeklinde header’da taşınır).
* Token içinde **kullanıcı kimliği, rolü, süre bilgisi** gibi şeyler bulunur.
* Sonraki her istek bu token ile gelir.
* Backend token’ı doğrular (signature + expiry) → kullanıcıyı tanır.

👉 Yani **sunucu tarafında state tutulmaz**, her şey token’da vardır.

✅ Neden REST API’de JWT daha avantajlı?

**1. REST API nasıl çalışır?**

REST API’nin en büyük prensiplerinden biri **stateless (durumsuzluk)**.  
Yani:

* Sunucu, gelen her isteği **bağımsız** değerlendirmek ister.
* “Bu kullanıcı kimdi, nerede oturum açmıştı?” gibi bilgileri **hatırlamaz**.
* Her istekte gerekli bilgiler **isteğin kendisiyle** birlikte gelir.

👉 Bu yüzden REST API’de **session (oturum)** mantığı aslında REST felsefesiyle çelişir.

**2. Session mantığı nasıl çalışır?**

* Kullanıcı login olur.
* Sunucu, belleğinde (RAM, Redis, DB) bu kullanıcı için bir **session kaydı** oluşturur.
* Kullanıcıya bir sessionId verir (cookie’de saklanır).
* Her istek geldiğinde sunucuya şu mesaj gelir:

“Benim sessionId = 123, beni hatırla.”

**Örnek benzetme:**  
Bir kafeye gidiyorsun → garson sana bir masa numarası veriyor.  
Garson sürekli defterinden bakıyor: “Masa 5 = Tuba Nur’un siparişleri.”  
👉 Yani garson (sunucu) **her şeyi hafızasında tutmak zorunda.**

**3. JWT mantığı nasıl çalışır?**

* Kullanıcı login olur.
* Sunucu kullanıcı bilgilerini alır ve bir **JWT token** üretir.
* Token kullanıcıya verilir (tarayıcı localStorage’da, mobilde hafızada saklanır).
* Her istek geldiğinde kullanıcı token’ı gönderir.
* Sunucu bu token’ı **kripto ile doğrular**, bilgiyi token’dan okur.

**Örnek benzetme:**  
Bir kafeye gidiyorsun → garson sana bir **damgalı kart** veriyor.  
Kartın üzerinde hem adın hem siparişlerin yazılı.  
Sonraki her siparişte kartı gösteriyorsun. Garson deftere bakmıyor → kartın üzerindeki damgaya güveniyor.  
👉 Yani garson (sunucu) **hiç hafıza tutmuyor, sadece kartın (token’ın) geçerli olup olmadığına bakıyor.**

**4. Neden JWT daha avantajlı?**

Şimdi madde madde daha basit şekilde açıklayayım:

1. **Stateless uyumlu**
   * Session’da sunucu “hafıza” tutuyor → REST’e aykırı.
   * JWT’de hafıza yok → her şey token’ın içinde.
2. **Dağıtık sistemlerde kolaylık**
   * Session → tek sunucu çalışıyorsa sorun yok. Ama 3 sunucu varsa, oturumları paylaşmak için Redis gibi ek sistem kurman gerekir.
   * JWT → her sunucu token’ı doğrulayabilir, ek sisteme gerek yok.
3. **Mobil / web / başka servislerde kullanım kolaylığı**
   * Session cookie tabanlıdır → web tarayıcı için uygundur ama mobil uygulamada uğraştırır.
   * JWT → sadece HTTP header’da taşınır, mobil/web fark etmez.
4. **Performans**
   * Session → her istekte sunucu session bilgisini RAM/Redis’ten okur.
   * JWT → sadece token’ın imzası kontrol edilir, çok hızlıdır.
5. **Ekstra bilgi taşıma**
   * JWT’nin içinde kullanıcı rolü, izinler, ID gibi bilgiler olabilir.
   * Örneğin “admin mi?” bilgisi token’dan direkt okunur → DB’ye tekrar gitmeye gerek yok.

**5. Dezavantajları**

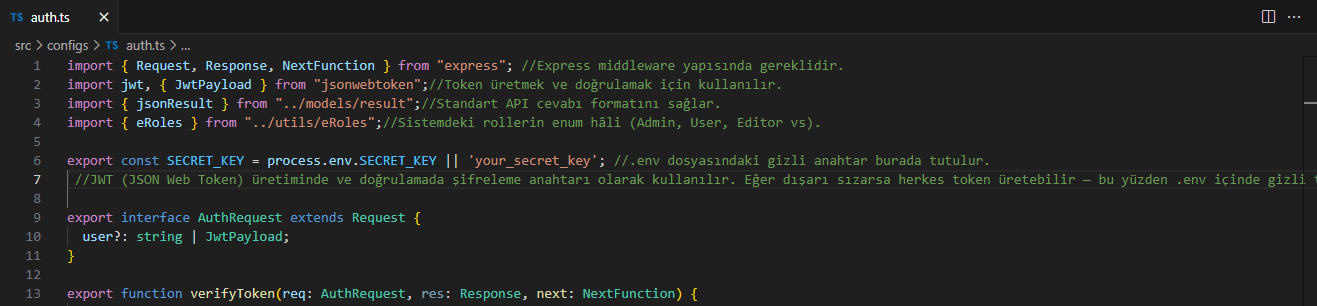
* JWT çalınırsa, expire süresi bitene kadar iptal edemezsin.
* Token boyutu biraz büyük olabilir, her istekte header’da taşınır.
* Refresh token mekanizması eklemek gerekir (uzun süreli oturum için).

**🔑 Özet**

* **Session:**
  + Küçük projelerde, sadece web tarayıcı için uygundur.
  + Sunucu hafızasında kullanıcıyı saklar → yük artar.
* **JWT:**
  + Büyük projelerde, mobil + web + API uyumluluğunda idealdir.
  + Sunucu hafızasız çalışır → ölçeklenebilir, hızlı, bağımsızdır.

👉 Kısaca:  
**Session = “sunucu beni hatırla”**  
**JWT = “benim kartım burada, kontrol et”**

Days\_75: RestApi projesinde user için eklemeler – postman denemeleri



**Node.js REST API projesinde auth.ts, kimlik doğrulama (authentication) ve yetkilendirme (authorization)** mekanizmasını kuruyor.  
Yani **kimin sisteme girebileceğini** ve **kimin ne yapabileceğini** belirleyen orta katman (middleware).

**🧠 AuthRequest (Özel Tip Tanımı)**

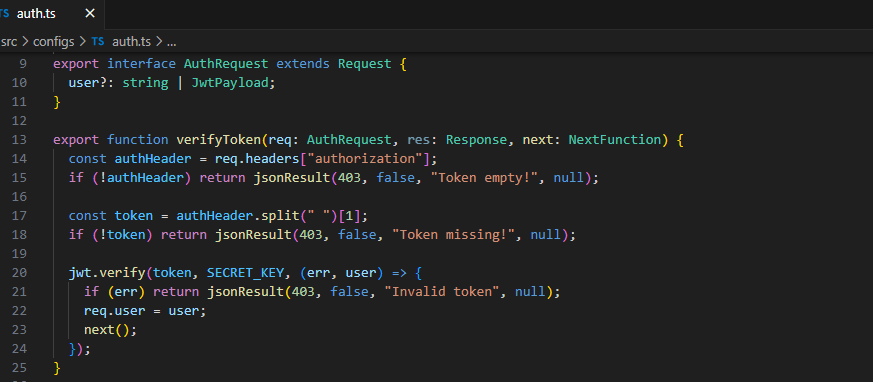
**export interface AuthRequest extends Request {**

**user?: string | JwtPayload; }**

* Express’in standart Request tipini genişletiyor.
* req.user alanını ekliyor.
* Böylece token doğrulandıktan sonra, token içindeki kullanıcı bilgileri bu alana yazılabilir.

📦 Örnek:  
Token doğrulanınca req.user = { id: "123", roles: ["Admin"] } gibi olur.

**🧱 verifyToken (Token Doğrulama Middleware’i)**

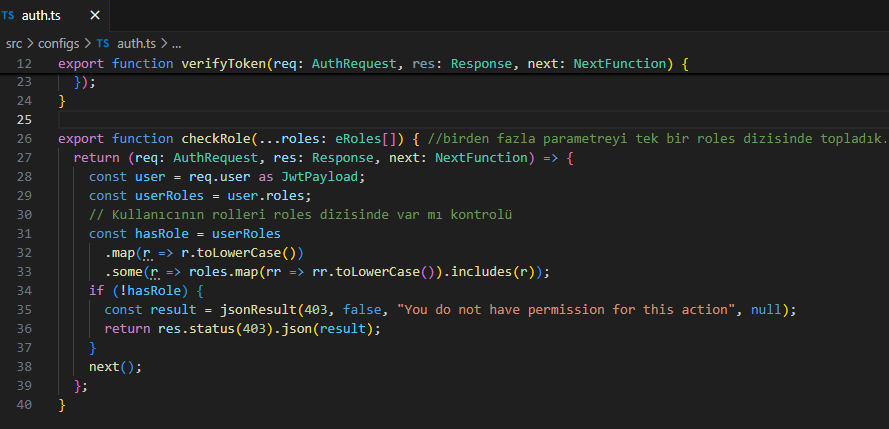


**🔍 İşleyiş:**

1. İstek (request) geldiğinde, Authorization başlığını (header) kontrol eder.
2. Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI... // burada postmanden login olup kullanıcının jwt’sini alıp yapıştır
3. Header yoksa → "Token empty!" hatası döner.
4. Token varsa → Bearer kısmını atıp token’ı alır.
5. jwt.verify(token, SECRET\_KEY) ile token geçerli mi kontrol edilir.
   * Token süresi dolmuşsa veya imzası bozuksa → “Invalid token”. // burada postmanden tekrar jwt ekle
6. Token geçerliyse → kullanıcı bilgisi (user) req.user içine yazılır.
7. next() çağrılır ve işlem devam eder (route’a geçer).

⚙️ **Etki:**  
Bu middleware, **her isteğin başında** kimlik doğrulaması yapar.  
Token yoksa → route’a gidemezsin.  
Token geçerliyse → istek onaylanır ve kullanıcı bilgisi taşınır.

🧱 checkRole (Yetkilendirme Middleware’i)



**🔍 İşleyiş:**

1. checkRole() bir fonksiyon döner — bu fonksiyon middleware olur.
2. Middleware, req.user içindeki rollerle (user.roles) istenen rolleri karşılaştırır.
3. Eğer kullanıcının rolü **izin verilen roller** arasında değilse:
   * 403 Forbidden hatası döner.
4. Rol uygunsa → next() ile devam eder.

**🧠 Günlük Hayattan Benzetme:**

🎟️ **verifyToken** = Kapıdaki güvenlik görevlisi.

“Bu binaya girme hakkın var mı? Kartını (token) göster.”

👮 **checkRole** = Binanın içindeki görevli.

“Tamam, binaya girdin ama bu odaya sadece müdürler girebilir. Sen stajyersin, olmaz.”

🧾 **.env** = Kasadaki gizli kod.

Sadece yönetici biliyor, başkası bilirse herkes sahte kart (token) basar.

**📘 Kısa Özet Notlar:**

| **Özellik** | **Görevi** | **Örnek Etki** |
| --- | --- | --- |
| verifyToken | Token doğrulama (kimlik kontrolü) | Token geçersizse kullanıcı reddedilir |
| checkRole | Yetki kontrolü (rol bazlı izin) | Sadece belirli roller erişebilir |
| .env SECRET\_KEY | Token imzası için gizli anahtar | Güvenliği sağlar |
| req.user | Doğrulanmış kullanıcı bilgisi | Diğer middleware’lerde veya route’larda erişilebilir |

Days\_75: RestApi Sawagger yapısı ve ince servis ayarları – Localde yapay zeka geliştirme

**Kurumlar:**

npm install express swagger-ui-express swagger-jsdoc

npm i --save-dev @types/swagger-ui-express

npm install swagger-jsdoc

npm i @types/swagger-jsdoc

npm install swagger-jsdoc –save

Terminalde;  **npm cache clean –force** komutu ne işe yarar?

**💡 Bu komutun amacı:**

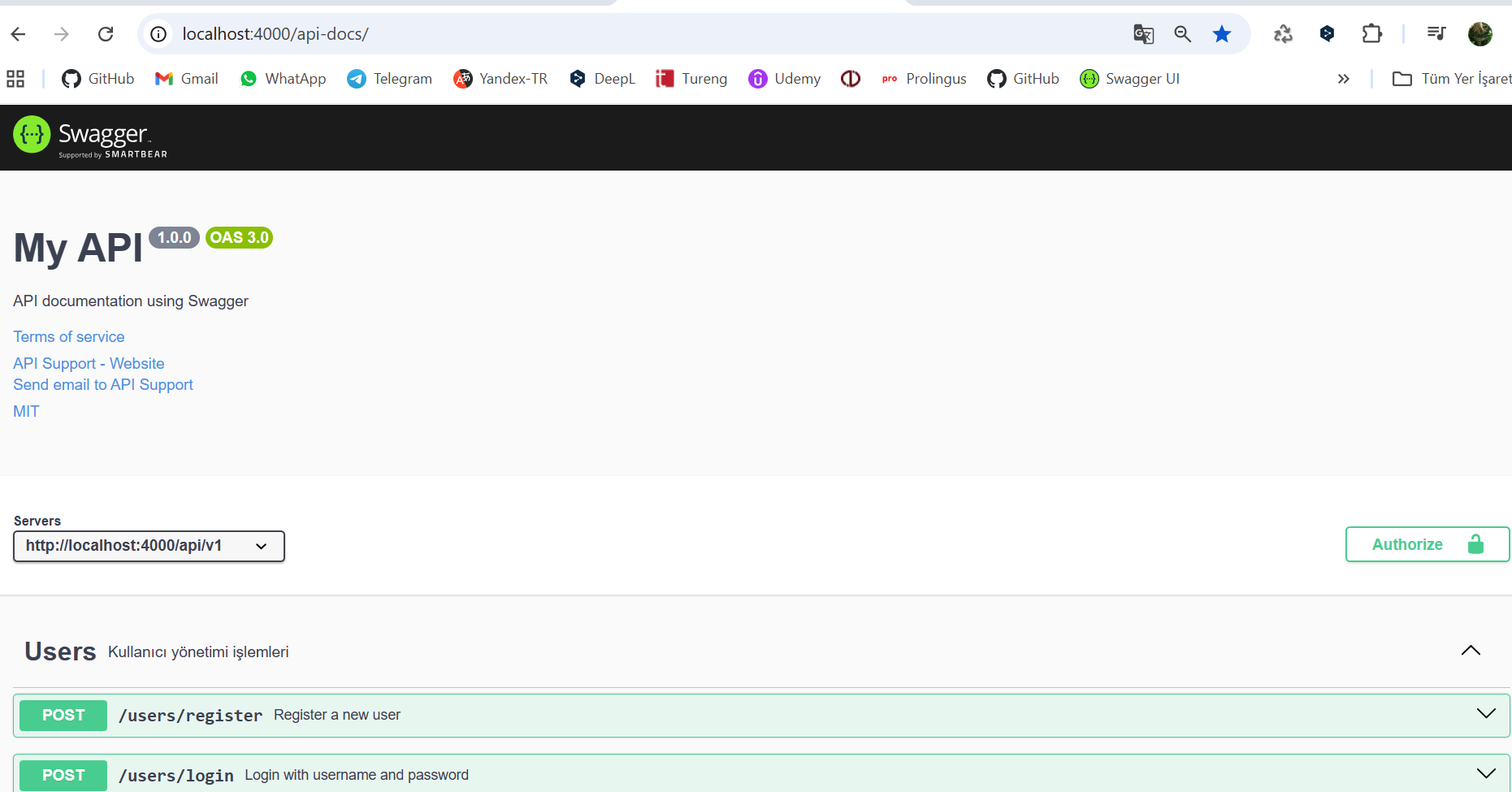
npm (Node Package Manager) tarafından tutulan **önbelleği (cache)** temizler. Normalde **npm cache clean** komutu **izin verilmeyen** bir işlemdir (npm seni korumak için engeller), ama --force ekleyince zorla temizler.

**🧱 Ne işe yarar?**

* Bozulmuş veya eski paket önbelleklerini siler.
* npm install sırasında oluşan hataları (örneğin “Cannot find module” veya “corrupted cache” gibi) çözebilir.
* Node modülleriyle ilgili garip, açıklanamayan hatalarda genellikle işe yarar.

**Swagger Nedir ve Ne İşe Yarar?**

**Swagger**, RESTful API’lerin dokümantasyonunu ve test edilmesini kolaylaştıran bir araçtır. Yani bir **API’nin nasıl çalıştığını**, **hangi endpoint’lerin (servislerin)** mevcut olduğunu, **hangi parametreleri** kabul ettiğini ve **hangi yanıtları döndürebileceğini** otomatik olarak gösterir. Swagger, API geliştiren kişiler ile bu API’yi kullanacak olan geliştiriciler arasındaki iletişimi kolaylaştırır.

****

**Swagger’ın Temel Amacı**

1. **API Dokümantasyonu:**  
   Kod içinde yazılmış olan endpoint’leri otomatik olarak algılar ve bunları kullanıcı dostu bir arayüzde gösterir.  
   Böylece “hangi URL ne işe yarıyor?” veya “bu servise ne göndermem gerekiyor?” gibi sorular ortadan kalkar.
2. **Test Ortamı Sağlama:**  
   Swagger arayüzü üzerinden (örneğin http://localhost:3000/api-docs)  
   herhangi bir endpoint’e istek atabilir, sonucu anında görebilirsin.  
   Bu, Postman gibi bir araç kullanmadan test yapmayı sağlar.
3. **Standart Sağlama (OpenAPI Specification):**  
   Swagger, **OpenAPI** standardını temel alır.  
   Bu sayede yazılan API’ler belirli bir kurala göre tanımlanır ve başka araçlar tarafından da anlaşılabilir hale gelir.

**Swagger Nasıl Çalışır?** Swagger, servis katmanında (örneğin Node.js, .NET, Java veya Python projelerinde)  
API’nin yapısını otomatik olarak tarar ve JSON/YAML formatında bir açıklama dosyası oluşturur.  
Bu dosya, API’nin “haritası” gibidir.

Ardından Swagger UI bu dosyayı okur ve kullanıcıya etkileşimli bir web arayüzü sunar. Yani Swagger, senin yazdığın kodu **okuyarak**, API’yi otomatik olarak “belgelenebilir” ve “test edilebilir” hale getirir.

**Swagger’ın Sağladığı Avantajlar**

✅ **Otomatik dokümantasyon:** Kod yazarken API açıklamalarını ayrıca belgelemene gerek kalmaz.  
✅ **Zaman kazandırır:** Geliştiriciler API yapısını hızlıca anlayabilir.  
✅ **İşbirliğini kolaylaştırır:** Frontend ve backend ekipleri arasında net bir anlaşma sağlar.  
✅ **Test kolaylığı:** API’yi doğrudan tarayıcı üzerinden test etme olanağı verir.  
✅ **Dil bağımsızdır:** Node.js, .NET, Java, Python gibi birçok dilde entegre edilebilir.

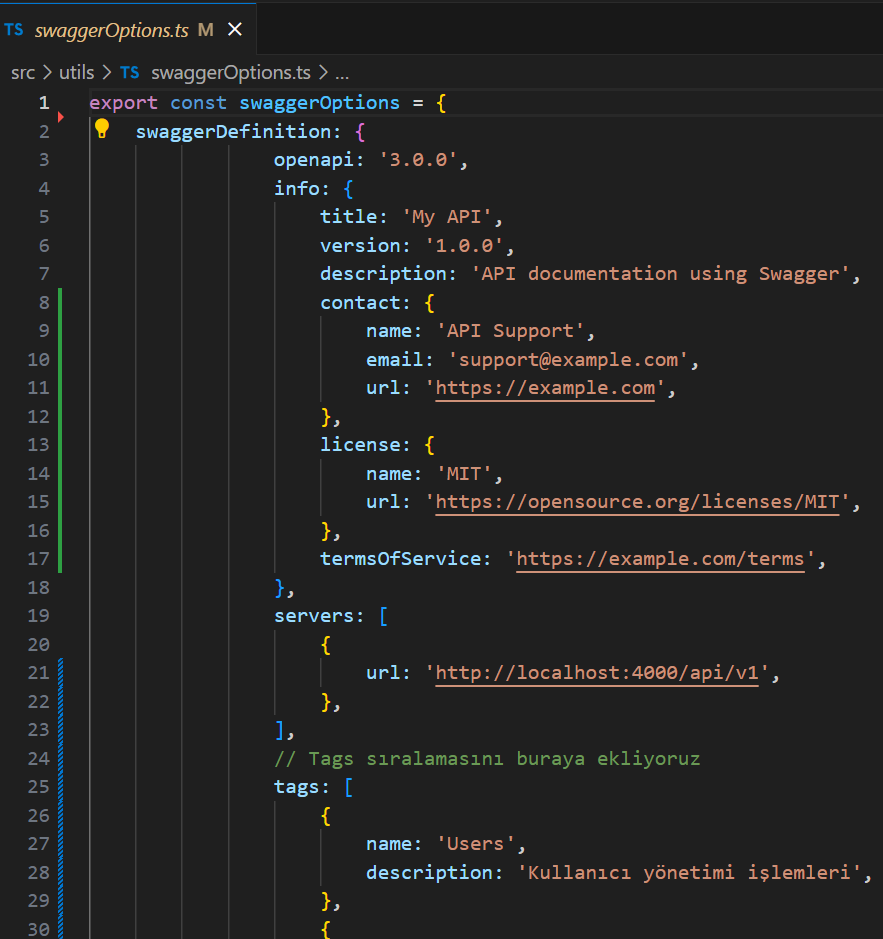
**Kısa Özet**

* **Swagger**, API’lerin yapısını otomatik olarak algılayıp dokümante eder.
* **OpenAPI standardı** sayesinde herkesin anlayabileceği ortak bir API tanımı oluşturur.
* **Swagger UI**, bu yapıyı görsel bir arayüze dönüştürür ve test etmeni sağlar.
* Servis katmanında kullanılır çünkü API endpoint’leri orada tanımlanır.

swaggerOptions.ts **👇**

**Bu yapı, Swagger’ın Node.js (Express) projesinde nasıl konfigüre edildiğini gösteriyor.**

**🔹 Genel Amaç**: swaggerOptions.ts, Swagger’ın tasarlanan projenin API’lerini nasıl okuyacağını, hangi URL’lerde göstereceğini, hangi bilgileri dokümantasyona dahil edeceğini belirleyen konfigürasyon dosyasıdır. Kısaca: Bu dosya, Swagger’a “API dokümanını nasıl oluşturacağını” anlatır.

**🔹 Kodun Bölüm Açıklaması**

**1. swaggerDefinition**

Bu kısım Swagger’ın temel yapı taşlarını tanımlar.  
Yani dokümantasyonun üst bilgileri (meta data), sunucu adresleri, etiketler (tags) ve güvenlik ayarları (security) burada belirlenir.

**🧩 openapi: '3.0.0'**

Swagger, OpenAPI standardını kullanır. 3.0.0 sürümü, modern API yapılarının çoğunu destekleyen en güncel OpenAPI standardıdır. Bu satır, Swagger’a hangi API sürümü formatını kullanacağını söyler.

**2. info bölümü**

Burada API dokümanının genel bilgileri tanımlanır — dokümantasyon sayfasının üst kısmında görünür.

**info: {**

**title: 'My API', // Doküman başlığı**

**version: '1.0.0', // API versiyonu**

**description: 'API documentation using Swagger', // Açıklama**

**contact: { // İletişim bilgileri**

**name: 'API Support',**

**email: 'support@example.com',**

**url: 'https://example.com',**

**},**

**license: { // Lisans bilgisi**

**name: 'MIT',**

**url: 'https://opensource.org/licenses/MIT',**

**},**

**termsOfService: 'https://example.com/terms', // Kullanım koşulları**

**},**

**📘** Bu bilgiler, Swagger UI ekranının üst kısmında otomatik olarak görüntülenir.

**3. servers bölümü**

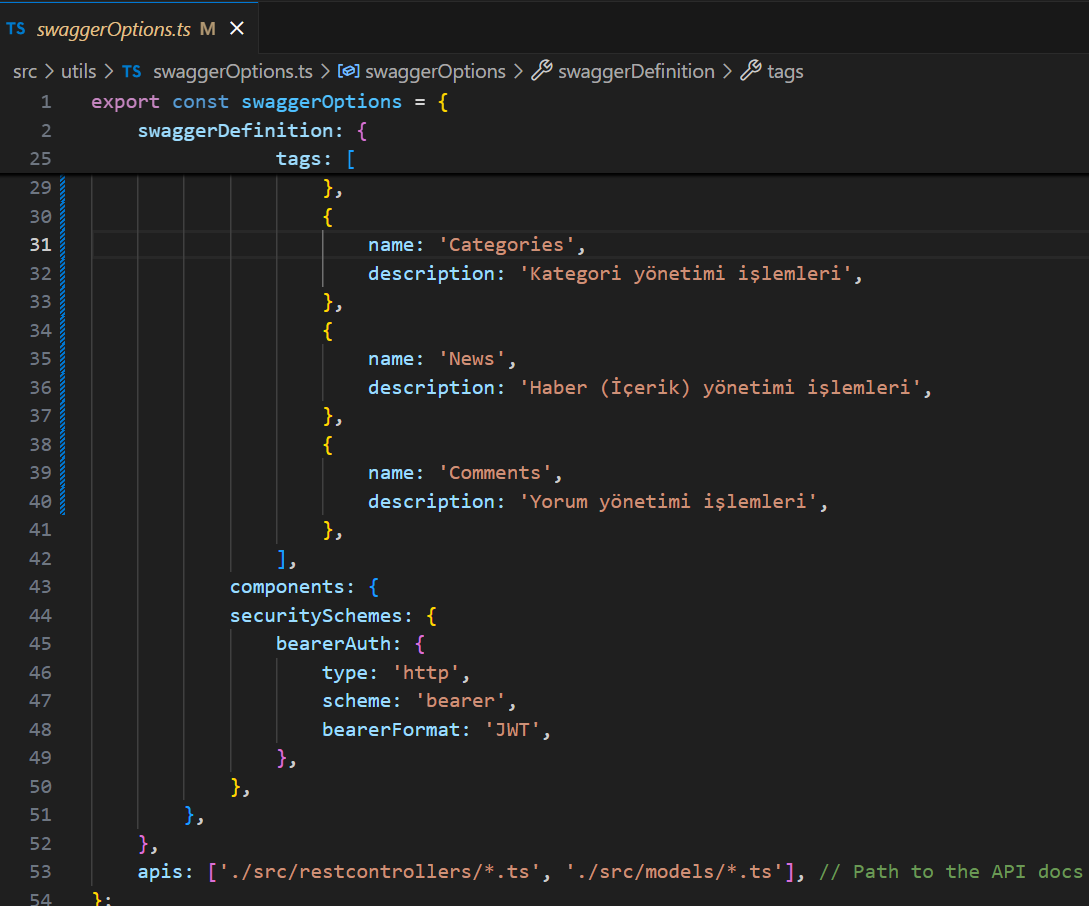
Burada API’nin hangi sunucu adreslerinde çalıştığı belirtilir.

**servers: [{**

**url: 'http://localhost:4000/api/v1',},**

**],**

Bu kısım, Swagger arayüzünde istek atarken temel URL olarak kullanılır. Örneğin, users endpoint’in /users olarak tanımlıysa,  
Swagger bunu http://localhost:4000/api/v1/users şeklinde tamamlar.

****

**4. tags bölümü**

Bu bölüm, API endpoint’lerini kategori (grup) olarak ayırmak için kullanılır.

**tags: [**

**{ name: 'Users', description: 'Kullanıcı yönetimi işlemleri' },**

**{ name: 'Categories', description: 'Kategori yönetimi işlemleri' },**

**{ name: 'News', description: 'Haber (İçerik) yönetimi işlemleri' },**

**{ name: 'Comments', description: 'Yorum yönetimi işlemleri' },**

**],**

👉 Bu sayede Swagger arayüzünde her endpoint kendi başlığı altında listelenir. Örneğin:

* **“Users” başlığı altında GET /users, POST /users olur.**
* **“Comments” altında POST /comments, DELETE /comments/{id} gibi işlemler görünür.**

**5. components → securitySchemes**

Bu bölüm, API’nin güvenlik yapısını (örneğin JWT Token, Basic Auth, API Key) tanımlar.

c**omponents: {**

**securitySchemes: {**

**bearerAuth: {**

**type: 'http',**

**scheme: 'bearer',**

**bearerFormat: 'JWT', },},},**

👉 Bu tanım sayesinde Swagger arayüzünde sağ üstte “Authorize” butonu çıkar. Buraya JWT token girerek korumalı endpoint’leri test edebilirsin. Yani Swagger bile yetkilendirme (authorization) işlemlerini senin için simüle eder.

**6. apis bölümü**

Swagger’ın dokümantasyonu hangi dosyalarda arayacağını belirler:

**apis: ['./src/restcontrollers/\*.ts', './src/models/\*.ts']**

**Yani bu satır diyor ki:** “Swagger, restcontrollers klasöründeki tüm .ts dosyalarını ve models klasöründeki modelleri tarasın.” Bu dosyalar içinde şu tarz açıklamalar varsa, Swagger bunları okuyup UI’a ekler:

**/\*\***

**\* @swagger**

**\* /users:**

**\* get:**

**\* summary: Get all users**

**\* tags: [Users]**

**\* responses:**

**\* 200:**

**\* description: Success**

**\*/**

Bu açıklama satırları, Swagger’ın otomatik olarak endpoint’leri tanımasını sağlar.

**🔹 Kısaca Özetle**

| **Bölüm** | **Görevi** |
| --- | --- |
| Openapi | API format standardını belirler |
| İnfo | API hakkında genel bilgileri tanımlar |
| Servers | API’nin hangi adreste çalıştığını belirtir |
| Tags | Endpoint’leri gruplandırır |
| components.securitySchemes | Yetkilendirme (ör. JWT) yapısını tanımlar |
| Apis | Swagger’ın endpoint açıklamalarını hangi dosyalarda arayacağını belirtir |

**🔹 Sonuç**

swaggerOptions.ts, Swagger’a senin projenin:

* Nerede çalıştığını,
* Hangi endpoint’leri olduğunu,
* Güvenlik yapısını ve bunların nasıl gruplanacağını anlatan bir rehberdir. Bu yapı sayesinde app.ts içinde:

**const swaggerDocs = swaggerJSDoc(swaggerOptions);**

**app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(swaggerDocs));** satırlarıyla Swagger arayüzü aktif hale gelir ve  
[http://localhost:4000/api-docs](http://localhost:4000/api-docs%20) adresinden API’yi görsel olarak test edebilirsin ✅