İSTENİLEN PROJE BİLGİERİ

"5- Product, Customer, Cart ve Order tablolarının ve bu tablolarının miras aldığı bir Base Entity'nin bulunduğu Spring Boot ile geliştirilmiş bir proje oluşturun.

Bir müşterinin bir sepeti (cart) ve birden fazla siparişi (order) olabilecek şekilde ilişkilendirme işlemini gerçekleştirin.

Sepetin ve siparişin toplam fiyat bilgisi her işlemde (sepete ekleme, çıkarma, miktar arttırıp azaltma) hesaplansın ve sepete kaydedilsin.

Bir müşteri sipariş geçtikten sonra, sipariş içerisindeki ürünlerin fiyatı daha sonradan güncellendiğinde müşteri satın aldığı anki fiyatı geçmişe yönelik olarak görebilsin. Bunun için farklı bir tablo tutabilirsiniz. Bu tablo üzerinde ürün, fiyat, miktar gibi bilgiler tutulabilir.

Ürün üzerinde stok takibi yapılsın, bir ürünün stoğu bittikten sonra o üründen daha fazla sipariş verilemesin.

Yazılması beklenen servisler:

AddCustomer

GetProduct

CreateProduct

UpdateProduct

DeleteProduct

GetCart

UpdateCart

EmptyCart

PlaceOrder

GetOrderForCode

GetAllOrdersForCustomer

AddProductToCart

RemoveProductFromCart"

MySQL123456\*

PROJE OLUŞTURURKEN EDİNİLEN KAZANIMLAR ve ÖĞRENİLEN BİLGİLER

**@MappedSuperclass**

@MappedSuperclass, Java’nın **JPA (Java Persistence API)** framework’ünde kullanılan bir \*\*anotasyon (annotation)\*\*dur. Bu anotasyon, bir sınıfın **veritabanı tablosu olmadan** başka JPA varlık (entity) sınıfları tarafından **miras alınabilir** olmasını sağlar.

**Ne işe yarar?**

* Ortak alanları (id, createdAt, updatedAt, vs.) bir yerde toplayarak tekrar eden kodları azaltır.
* Bu sınıftaki alanlar, miras alan sınıfın veritabanı tablosuna **otomatik olarak eklenir**.
* JPA bu sınıfı **doğrudan bir entity gibi yönetmez**. Yani @Entity gibi davranmaz, sadece bir şablon görevi görür.

| **Özellik** | **Açıklama** |
| --- | --- |
| Tabloda yer almaz | @MappedSuperclass sınıfının kendisi için tablo oluşmaz. |
| Kalıtım için tasarlanmıştır | Diğer entity sınıflarına miras vererek ortak alanları paylaşmalarını sağlar. |
| Alanlar entity’ye aktarılır | İçindeki alanlar, miras alan sınıfın veritabanı tablosuna yansıtılır. |

**@Entity**

Entity, **JPA (Java Persistence API)** kapsamında kullanılan en temel anotasyonlardan biridir ve **bir sınıfın veritabanındaki bir tabloya karşılık geldiğini belirtir.** Başka bir deyişle, bu sınıf bir **kalıcı veri nesnesidir (persistent object)**.

 @Entity, sınıfı **JPA'nın yönetmesi gereken bir veri varlığı (entity)** olarak işaretler.

 Bu sınıfın nesneleri, **veritabanı kayıtları** olarak temsil edilir.

 JPA bu sınıfı otomatik olarak işler, sorgular, kaydeder, günceller ve siler.

| **Şart** | **Açıklama** |
| --- | --- |
| 1. Boş constructor olmalı | JPA, yansıtma (reflection) kullanarak sınıfı örnekler. |
| 2. @Id zorunlu | Her entity sınıfında **birincil anahtar (primary key)** tanımlanmalıdır. |
| 3. Public ya da protected olmalı | JPA erişebilmesi için. |
| 4. Final olmamalı | JPA proxy sınıflar oluşturur, final olursa miras alınamaz. |

**@GeneratedValue**

@GeneratedValue, JPA (Java Persistence API) içerisinde kullanılan bir anotasyondur ve bir entity sınıfındaki @Id (birincil anahtar) alanının **otomatik olarak nasıl üretileceğini** (yani değerinin nasıl atanacağını) belirtir.

@Entity

public class Product {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

private String name;

}

Bu örnekte:

* id alanı birincil anahtar (@Id)
* @GeneratedValue sayesinde otomatik olarak oluşturulacak

| **Strateji** | **Açıklama** | **Uyumlu Veritabanları** |
| --- | --- | --- |
| AUTO (varsayılan) | JPA, veritabanına göre en uygun stratejiyi otomatik seçer. | Tümü |
| IDENTITY | Veritabanı auto\_increment kullanarak id üretir. | MySQL, SQL Server |
| SEQUENCE | Bir **sequence (sayaç)** kullanılarak id oluşturulur. | PostgreSQL, Oracle |
| TABLE | ID üretmek için özel bir tablo oluşturulur. | Tümü (daha yavaş olabilir) |

### GenerationType.IDENTITY (MySQL için en yaygın)

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Long id;

→ MySQL gibi veritabanlarında AUTO\_INCREMENT olarak çalışır.

**@Data**

@Data anotasyonu, Java'da **Lombok** kütüphanesine ait bir anotasyondur ve sınıfların (class) daha kısa, daha okunabilir ve daha az kodla yazılmasını sağlar. Özellikle **POJO (Plain Old Java Object)** sınıflarında sıkça kullanılır.

| **Anotasyon** | **Görevi** |
| --- | --- |
| @Getter | Tüm alanlar (field) için getter metodları oluşturur |
| @Setter | Tüm alanlar (field) için setter metodları oluşturur |
| @ToString | toString() metodunu otomatik olarak oluşturur |
| @EqualsAndHashCode | equals() ve hashCode() metodlarını oluşturur |
| @RequiredArgsConstructor | final veya @NonNull ile işaretlenmiş alanlar için constructor (kurucu) oluşturur |

**@Id**

Id anotasyonu, **Java Persistence API (JPA)**'da kullanılır ve bir sınıfın veritabanındaki **birincil anahtarını (primary key)** belirtmek için kullanılır.

## Neden int yerine Long Kullanılır?

* int veri tipi 2 milyar civarına kadar değer alabilir (32 bit).
* Long ise **çok daha büyük sayıları** destekler (64 bit).
* Gerçek uygulamalarda, özellikle büyük veri setlerinde Long tercih edilir.
* Ayrıca Long, null değer alabildiği için Hibernate ve Spring Data gibi framework'lerde daha uygundur.

| **Terim** | **Açıklama** |
| --- | --- |
| Long | Java'nın **nesne türü** 64 bit tamsayı veri tipi (java.lang.Long) |
| id | Genellikle nesneye özel **benzersiz tanımlayıcı alan** |
| Kullanım Alanı | Veritabanı entity'lerinde @Id ile birlikte |
| Neden Long? | Büyük sayılar ve null değer desteği için |

**@Column**

@Column anotasyonu, Java'da **JPA (Java Persistence API)** ile birlikte kullanılır ve bir **entity (varlık) sınıfındaki alanı (field)**, veritabanındaki bir **sütunla (column)** eşleştirir. Bu anotasyon sayesinde, sınıf içindeki alanın veritabanında nasıl temsil edileceği hakkında detaylı yapılandırmalar yapılabilir.

| **Parametre** | **Açıklama** |
| --- | --- |
| name | Veritabanındaki sütun adını belirtir. Belirtilmezse, alan adı kullanılır. |
| nullable | true ise sütun NULL değeri alabilir. false yaparsan NOT NULL olur. |
| unique | true yaparsan, bu sütun için **UNIQUE constraint** uygulanır. |
| length | (String türü için) Maksimum karakter uzunluğunu belirler. Varsayılan 255'tir. |
| updatable | false yaparsan, bu sütun güncellenemez. |
| insertable | false yaparsan, bu sütun veritabanına kayıt eklenirken dahil edilmez. |
| columnDefinition | SQL tarafında özel sütun tanımı yapılmasını sağlar (örneğin: "TEXT"). |
| precision ve scale | Ondalıklı sayılar için hassasiyet ve basamak sayısını belirtir. |

**ÖRNEK:**

@Column(name = "email", nullable = false, unique = true, length = 100)

private String email;

Bu örnek:

* Sütunun adı email olacak.
* NULL olamaz.
* Aynı değer birden fazla satırda olamaz (UNIQUE).
* Maksimum 100 karakterlik String değer saklar.

| **Anotasyon** | **Ne İçin Kullanılır?** |
| --- | --- |
| @Column | **Basit alanları** veritabanı sütunlarına eşlemek için kullanılır. (örneğin: String, int) |
| @JoinColumn | **İlişkili entity'leri** (başka sınıflarla ilişkileri) temsil eden yabancı anahtarları eşlemek için kullanılır. |

**Repository**

Spring Boot'ta **repository**, veritabanı işlemlerini (CRUD: Create, Read, Update, Delete) gerçekleştiren ve veri erişim katmanını temsil eden bir **arayüzdür**.

## Repository Nedir?

Repository, Java'da **veri erişim mantığını soyutlamak** için kullanılan bir tasarım desenidir.  
Spring Data JPA'da bu, genellikle JpaRepository, CrudRepository gibi arayüzleri genişleterek yapılır.

**ÖRNEK:**

public interface ProductRepository extends JpaRepository<Product, Long> {

}

Bu satırla Spring'e şu söyleniyor:

"Ben Product nesnesini veri tabanında Long türündeki id ile temsil ediyorum, ve JPA'nın sağladığı tüm veri erişim fonksiyonlarını kullanmak istiyorum."

| **İşlem** | **Açıklama** |
| --- | --- |
| save(entity) | Veri kaydetmek veya güncellemek |
| findById(id) | ID’ye göre veri bulmak |
| findAll() | Tüm verileri getirmek |
| deleteById(id) | ID’ye göre veriyi silmek |
| count() | Kayıt sayısını almak |
| existsById(id) | Belirli ID’ye sahip veri var mı diye kontrol etmek |

Bunların hiçbiri için SQL yazmaya gerek yok!

## Repository Ne Zaman Kullanılır?

* Servis katmanında veri tabanıyla iletişim kurmak için.
* Controller'dan gelen istekleri veriyle işlemek için.
* SQL yazmadan, veri erişim işlemlerini kolaylaştırmak için.

## Katmanlar Arasında Yeri

[Controller] → [Service] → [Repository] → [Database]

 **Controller**: HTTP isteklerini alır.

 **Service**: İş mantığı burada olur.

 **Repository**: Veritabanı işlemleri burada yapılır.

**DTO**

## ****DTO Nedir?****

**DTO (Data Transfer Object)**, adından da anlaşılacağı gibi, bir uygulamada veri transferi amacıyla kullanılan bir nesnedir. Genellikle **veri tabanı modelleri (Entity)** ile istemci (client), servis veya dış sistemler arasında veri taşımak için kullanılır.

## ****DTO’nun Amacı Nedir?****

### 1. **Katmanlar Arasında Veri Taşımak**

Örneğin:

* Controller → Service → Repository → Entity
* Entity → Service → DTO → Controller → Client (örneğin frontend)

DTO’lar sayesinde veri, katmanlar arasında **güvenli ve kontrollü bir şekilde** taşınır.

### 2. **Gereksiz Verinin Gizlenmesi**

* Entity içinde fazladan bilgiler olabilir (örneğin: password, createdAt, updatedAt).
* DTO kullanarak sadece ihtiyacımız olan veriyi taşıyabiliriz.

### 3. **Veri Doğrulama ve Formatlama**

DTO’larda @NotNull, @Size, @Email gibi anotasyonlarla doğrulama kuralları koyabiliriz.

### 4. **Performans Artışı**

* Ağ üzerinden taşınan veri boyutunu azaltır.
* Özellikle büyük entity'lerde gereksiz alanları taşımaz.

### 5. **Entity’nin Güvenliğini Sağlar**

* DTO ile yalnızca izin verilen alanları dış dünyaya açarsın.
* Entity doğrudan dış sistemlere açılırsa kötü niyetli kişiler hassas alanlara müdahale edebilir.

## ****DTO Kullanılmadığında Ne Olur?****

* Güvenlik açıkları (şifre gibi alanlar dışarı çıkar)
* Fazla veri gönderme nedeniyle performans düşer
* API kararsızlaşır (entity değiştikçe API’ler bozulur)
* Validation zorlaşır

| **Özellik** | **Entity** | **DTO** |
| --- | --- | --- |
| Veritabanı ile bağlantı | Evet (JPA ile) | Hayır |
| Anotasyonlar | @Entity, @Table vs. | @Data, @NotNull vs. |
| Amaç | Veritabanı ile çalışmak | Veri taşımak |
| İçerik | Tüm alanlar | Sadece gerekli alanlar |
| Dışa Açılma | Açılmamalı | Açılabilir |

**API**

## ****API Nedir? (Application Programming Interface)****

**API (Uygulama Programlama Arayüzü)**, iki farklı yazılımın **birbirleriyle iletişim kurmasını** sağlayan bir **arayüzdür**.

Kısa tanım: **Bir yazılımın başka bir yazılıma "şöyle çalış" demesini sağlayan arabirim.**

API, dış dünya ile sistemin içi arasında **bir aracı** görevindedir.

## Yazılımda API Ne İşe Yarar?

* **Veri alışverişi yapmanı sağlar.**
* Sistemler arasında iletişimi standartlaştırır.
* Bir uygulamanın başka bir uygulamanın **fonksiyonlarını kullanmasını sağlar.**

## API ve DTO Arasındaki İlişki

* DTO → veri formatıdır (ne gönderilip ne alındığını belirler).
* API → veri yoludur (hangi verinin nereye gidip geleceğini sağlar).

**Mapper**

**Mapper Ne İşe Yarar?**

| **Amaç** | **Açıklama** |
| --- | --- |
| Katmanları ayırmak | Entity ↔ DTO geçişi sağlar, servisleri sade tutar |
| Güvenlik | API’de hangi verilerin görüneceğini kontrol eder |
| Temizlik | Kod tekrarını ve karışıklığı azaltır |
| Esneklik | Veritabanı yapısını değiştirmeden API yapısını değiştirebilirsin |
| Test kolaylığı | Tek başına test edilebilir hale gelir |

## Mapper Gereksinimini Ne Belirler?

Bir sınıf için **mapper yazma ihtiyacı**, şu durumlarda ortaya çıkar:

### 1. **Veri dışarıdan alınacaksa (Request DTO)**

Örnek: CreateProductRequest → Product

Burada kullanıcıdan gelen veriyi Entity'ye çevirmeliyiz. Yani DTO ➝ Entity

### 2. **Veri dışarıya gösterilecekse (Response DTO)**

Örnek: Product → ProductResponse

Burada veritabanı nesnesini dış dünyaya uygun formata çevirmeliyiz. Yani Entity ➝ DTO

### 3. **Entity ile DTO birbirine benzemiyorsa / dönüştürme gerekiyorsa**

Eğer DTO ile Entity birebir aynı değilse (ki genellikle farklıdır), arada çeviri yapacak bir mapper gerekir.

## Bu Projede Neden Sadece 4 Mapper Var?

### 1. **ProductMapper**

* CreateProductRequest, UpdateProductRequest, ProductResponse
* Product entity’sine çevirmek ve entity’den DTO üretmek için

### **2. CustomerMapper**

* CustomerRequest → Customer
* Customer → CustomerResponse

### **3. CartMapper**

* Cart → CartResponse, CartItemResponse
* Çünkü kullanıcı sepete ürün eklediğinde/çıkardığında geri dönüşte sepet içeriği gösterilecek

### 4. **OrderMapper**

* Order → OrderResponse, OrderItemResponse
* Sipariş geçildiğinde geçmiş sipariş verilerini göstermek için

## Peki Neden CartItem, OrderItem için Ayrı Mapper Yazmadık?

Çünkü bu sınıflar **Cart** ve **Order** mapper'larının içinde kullanılıyor. Onlar kendi başına DTO dönüşümü yapılması gereken bağımsız yapılar değil. Örneğin:

public CartItemResponse toCartItemResponse(CartItem item)

Bu yeterli çünkü bu dönüşüm sadece CartResponse içinde kullanılıyor. Bu yüzden CartItemMapper diye ayrı bir sınıf açmaya gerek yok.

**Service Katmanı:** Bu katman, iş mantığını içerir. Bir ürünün sepete eklenmesi, siparişin oluşturulması gibi işlemleri yönetir. DTO'ları entity'lere dönüştürmek ve repository'leri kullanarak veri tabanı işlemlerini yapmak bu katmanın görevidir.

**@Service**

## @Service Nedir? Ne İşe Yarar?

@Service, **Spring Framework**’e ait bir anotasyondur ve bir sınıfın **servis katmanına ait** olduğunu belirtir. Spring, bu anotasyon sayesinde sınıfı otomatik olarak bir **Spring Bean** olarak algılar ve **IoC Container** içerisine alır.

### Özellikleri:

* Bu sınıfa ait nesne Spring tarafından **otomatik olarak oluşturulur (instantiate edilir)**.
* **Dependency Injection (bağımlılık enjeksiyonu)** bu sınıfa uygulanabilir.
* Katmanlı mimaride Service katmanı genellikle **iş mantığını (business logic)** barındırır.

## @RequiredArgsConstructor Nedir? Ne İşe Yarar?

@RequiredArgsConstructor, **Lombok kütüphanesine** ait bir anotasyondur ve sınıfın içinde final olan tüm alanlar için otomatik olarak bir **constructor (yapıcı metot)** oluşturur.

### Özellikleri:

* Sınıfın içinde final olarak tanımlanan değişkenleri (özellikle private final service/repository gibi bağımlılıklar) constructor ile dışarıdan almanızı sağlar.
* Spring’in @Autowired yerine constructor-based injection ile bağımlılık enjekte etmeye olanak tanır.

ÖRNEK:

@Service

@RequiredArgsConstructor

public class ProductService {

private final ProductRepository productRepository;

public void listAllProducts() {

productRepository.findAll();

}

}

Bu kodla birlikte Lombok şu constructor’ı otomatik olarak yazar:

public ProductService(ProductRepository productRepository) {

this.productRepository = productRepository;

}

### Neden Kullanılır?

* @Autowired yerine constructor injection kullanmak, daha okunabilir ve test edilebilir kod sağlar.
* final ile tanımlandığı için bağımlılıkların **değiştirilemez** olduğu garanti altına alınır.
* Lombok sayesinde **boilerplate kod** (yani tekrar eden constructor yazma zorunluluğu) ortadan kalkar.

| **Anotasyon** | **Kütüphane** | **Görevi** |
| --- | --- | --- |
| @Service | Spring | Sınıfı servis katmanı olarak işaretler ve Spring Bean yapar. |
| @RequiredArgsConstructor | Lombok | final alanlar için otomatik constructor oluşturur. |

**@Valid**

@Valid Java'da özellikle **Spring Framework** ve **Jakarta Bean Validation (eski adıyla Javax Validation)** ile birlikte kullanılan bir **validasyon (doğrulama)** anotasyonudur.

Bu anotasyon, bir nesneye ait alanların belirli kurallara göre geçerli olup olmadığını kontrol etmek için kullanılır. Genellikle **controller katmanında** kullanılır ve bir DTO (Data Transfer Object) sınıfındaki alanlara eklenmiş validasyon kurallarının çalıştırılmasını sağlar.

1- Veri Doğrulama (Validation):

@Valid, verilerin uygunluğunu kontrol etmek için kullanılır. Özellikle HTTP isteklerinden gelen verilerin, DTO'lar veya modellerin uygunluğunu denetlemek için sıkça kullanılır. Bu, sunucu tarafında veri bütünlüğünü sağlamak ve güvenliği artırmak için önemlidir.

2- Java Bean Doğrulama:

@Valid, Java Bean doğrulama (Bean Validation) spesifikasyonu ile uyumlu çalışır. Bu spesifikasyon, veri doğrulama kurallarını tanımlamak için çeşitli anotasyonlar içerir (örneğin @NotNull, @Size, @Pattern, vb.). @Valid kullanarak bu anotasyonları etkinleştirir ve belirli kurallara uymayan verileri reddeder.

3-Nesne Doğrulama:

@Valid, iç içe geçmiş nesnelerin doğrulanmasını destekler. Yani, bir nesne başka nesneler içeriyorsa ve bunlar da doğrulanması gerekiyorsa, @Valid kullanarak bu iç içe geçmiş nesneleri de doğrulayabilirsiniz.

4-Hata İşleme:

Doğrulama başarısız olduğunda, hata mesajları genellikle istemciye döner ve istemci bu hataları işleyebilir. Bu, kullanıcı dostu hata mesajlarını göstermek veya hataları loglamak için kullanışlıdır.

**@Transactional**

### **@Transactional nedir?**

* @Transactional, **Spring Framework** içinde kullanılan bir anotasyondur.
* Bir metodun veya sınıfın **veritabanı işlemlerini (transaction)** yönetmek için kullanılır.
* Bu anotasyonla işaretlenen metod, bir işlem (transaction) kapsamında çalışır.
* Eğer metod içinde bir hata oluşursa, yapılan tüm veritabanı işlemleri geri alınır (**rollback** yapılır).
* Hata yoksa, işlem başarılı şekilde tamamlanır ve değişiklikler veritabanına kaydedilir (**commit** yapılır).

### **Neden kullanılır?**

* Veritabanı işlemlerinde tutarlılığı sağlamak için.
* Örneğin; birden fazla tabloya veri yazarken, tüm işlemlerin başarılı olması gerekir. Eğer birinde hata olursa, diğerleri de geri alınmalı.
* Bu sayede veriler tutarlı kalır, kısmi güncellemeler önlenir.

**@RestController**

* @RestController, Spring'in @Controller ve @ResponseBody anotasyonlarının birleşimidir.
* Bu anotasyon ile işaretlenen sınıf, HTTP isteklerini karşılar ve genellikle JSON veya XML formatında veri döner.
* Metodların dönüş değerleri otomatik olarak HTTP yanıt gövdesine (response body) yazılır; yani veriler direkt olarak istemciye gönderilir, bir HTML sayfası render edilmez.

### **Özetle:**

* **@Controller**: Web sayfası (View) döndürmek için kullanılır.
* **@RestController**: Veri (JSON, XML vb.) döndürmek için kullanılır.

**@RequestMapping**

@RequestMapping("/api/products") Spring Framework’de kullanılan bir anotasyondur ve HTTP isteklerini belirli bir URL yoluna (endpoint) yönlendirmek için kullanılır.

 @RequestMapping bir sınıf veya metoda konabilir.

 **Sınıfa konulursa:** O sınıftaki tüm metodların yolları, burada belirtilen yol ile başlar.

 **Metoda konulursa:** O metoda gelen isteğin tam yolu burada belirtilir (veya sınıf seviyesindeki yolun devamı olur).

 Yalnızca URL yolunu değil, HTTP metodunu (GET, POST, PUT, DELETE vb.) da belirtebilirsiniz (örneğin @RequestMapping(value="/api/products", method=RequestMethod.GET)).

**@PostMapping**

@PostMapping Spring Framework'te kullanılan bir anotasyondur ve HTTP **POST** isteklerini karşılamak için kullanılır.

 @PostMapping, @RequestMapping(method = RequestMethod.POST) anotasyonunun kısaltılmış, daha okunabilir halidir.

 Genellikle yeni bir veri oluşturmak (örneğin yeni ürün eklemek, kayıt oluşturmak) için kullanılır.

 @PostMapping ile işaretlenen metod, gelen POST isteğindeki verileri alır, işler ve genellikle oluşturulan kaydı ya da sonucu döner.

**@GetMapping**

@GetMapping da Spring Framework’te kullanılan ve HTTP **GET** isteklerini karşılamak için kullanılan bir anotasyondur.

* @GetMapping, @RequestMapping(method = RequestMethod.GET) anotasyonunun daha kısa ve okunabilir bir halidir.
* HTTP GET isteği, genellikle sunucudan veri **almak** (okumak) için kullanılır.
* @GetMapping ile işaretlenen metodlar, istemcinin URL üzerinden yaptığı GET isteklerine cevap verir.

**@PutMapping**

@PutMapping Spring Framework’te kullanılan bir anotasyondur ve HTTP **PUT** isteklerini karşılamak için kullanılır.

 @PutMapping, @RequestMapping(method = RequestMethod.PUT) anotasyonunun kısaltmasıdır.

 HTTP PUT isteği genellikle mevcut bir kaynağı **güncellemek** için kullanılır.

 @PutMapping ile işaretlenen metod, istemciden güncellenmiş veri alır ve ilgili kaynağı bu verilerle değiştirir.

**@DeleteMapping**

@DeleteMapping, Spring Framework’te HTTP **DELETE** isteklerini karşılamak için kullanılan bir anotasyondur.

 @DeleteMapping, @RequestMapping(method = RequestMethod.DELETE) anotasyonunun kısaltılmış halidir.

 HTTP DELETE isteği, genellikle belirli bir kaynağın **silinmesi** için kullanılır.

 @DeleteMapping ile işaretlenen metod, istemciden gelen silme isteğini alır ve belirtilen kaynağı sunucudan kaldırır.

**@PathVariable**

@PathVariable, Spring Framework’te URL içindeki değişken değerleri almak için kullanılan bir anotasyondur.

 RESTful API’lerde URL’ler genellikle belirli kaynakları belirtmek için kullanılır.

 URL içindeki dinamik kısımlar (örneğin /products/5 adresindeki 5), @PathVariable ile metoda parametre olarak alınabilir.

 Böylece, URL’den gelen değeri kolayca Java metodunda kullanabilirsin.

Eğer URL’deki değişken adı ile parametre adı aynı ise @PathVariable içinde parametre ismi yazmaya gerek yok.

**@RequestBody**

@RequestBody, Spring Framework’te bir HTTP isteğinin **gövdesindeki (body)** veriyi **otomatik olarak bir Java nesnesine (DTO veya Entity)** dönüştürmek için kullanılır.

## 1. Görevi Nedir?

* HTTP istekleri genellikle gövdelerinde (body) **JSON, XML, form-data** gibi yapılandırılmış veri taşır.
* @RequestBody, bu veriyi alır ve Java’daki bir sınıf örneğine (nesneye) otomatik olarak dönüştürür.
* Bu işleme **deserialization** (serileştirilmiş verinin çözülmesi) denir.

## 2. Nasıl Çalışır?

Spring MVC, @RequestBody ile işaretlenmiş bir parametre gördüğünde aşağıdaki adımları uygular:

### Aşama Aşama:

1. **HTTP gövdesi** (InputStream) olarak okunur.
2. Spring, uygun bir HttpMessageConverter seçer.
3. Bu converter, içerik tipine göre çalışır (örneğin JSON için MappingJackson2HttpMessageConverter).
4. Gövde verisi Java sınıfına dönüştürülür.
5. Oluşan Java nesnesi, metottaki parametreye enjekte edilir.

## 3. Kullanım Amacı

@RequestBody genellikle şuralarda kullanılır:

| **Kullanım** | **Açıklama** |
| --- | --- |
| API endpoint'lerinde | RESTful servislerde veri almak için |
| Form-submission | Frontend tarafından gönderilen form verilerini almak |
| JSON, XML, YAML | Gelişmiş veri yapılarını almak için |

@RequestBody genellikle @Valid veya @Validated ile birlikte kullanılır.

| **Özellik** | **Açıklama** |
| --- | --- |
| Anlamı | HTTP gövdesindeki veriyi Java nesnesine çevirir |
| Kullanım Alanı | REST API'de JSON veri almak |
| Mekanizma | Jackson ile deserialization |
| Şartlar | DTO'da alanlar eşleşmeli, Content-Type doğru olmalı |
| Sık hata | Uyuşmayan veri tipi, eksik header, eksik setter |
| İleri Kullanım | @Valid, @JsonProperty, @RequestBody(required = false) |

**ResponseEntity**

## ResponseEntity NEDİR?

ResponseEntity<T> sınıfı, Spring Framework’te HTTP yanıtlarını temsil eden bir sınıftır. Generic olarak tanımlanmıştır (<T>), yani döneceğiniz veri tipi ne olursa olsun, ona uygun kullanılabilir:

**ResponseEntity<ProductResponse>**

Yani:

* Bu bir **HTTP cevabıdır**,
* Cevap içinde **ProductResponse türünde veri taşıyacaktır.**

## NEDEN ResponseEntity KULLANILIR?

Spring'de @RestController altında metotlar genellikle doğrudan veri döner. Örneğin:

**@GetMapping**

**public ProductResponse getProduct() {**

**return productService.getProduct();**

**}**

Ama bu yöntem size sadece veriyi döner. **HTTP durum kodu (status), header, vs.** gibi detayları kontrol edemezsiniz.  
İşte burada ResponseEntity devreye girer:

**@GetMapping**

**public ResponseEntity<ProductResponse> getProduct() {**

**ProductResponse response = productService.getProduct();**

**return ResponseEntity.ok(response);**

**}**

Bu sayede:

* HTTP durum kodunu kontrol edersin (örneğin 200 OK, 201 CREATED, 400 BAD REQUEST...)
* Header ekleyebilirsin
* Body olarak istediğin veriyi dönebilirsin

## ResponseEntity NASIL ÇALIŞIR?

### 1. **Durum Kodu (HTTP Status)**

**ResponseEntity.ok() // 200 OK**

**ResponseEntity.status(HttpStatus.CREATED) // 201 CREATED**

**ResponseEntity.status(HttpStatus.NOT\_FOUND) // 404 NOT FOUND**

### 2. **Body (Gövde)**

**ResponseEntity.ok(productResponse); // Body'ye productResponse yerleştirir**

### 3. **Headers (Başlıklar)**

**HttpHeaders headers = new HttpHeaders();**

**headers.set("Location", "/products/123");**

**return new ResponseEntity<>(productResponse, headers, HttpStatus.CREATED);**

## ResponseEntity.ok(...) Ne Yapar?

ResponseEntity.ok(...) satırı aslında **bir HTTP yanıtı (response)** oluşturur.  
Bu yanıtın:

* **Durum kodu**: 200 OK
* **Body** (gövde): Parantez içine verdiğin nesne (...)
* **Header**: Varsayılan (sen belirtmediğin sürece)

ResponseEntity.ok(...) çağrıldığında:

1. ok(...) aslında bir **static factory method**'dur.

**public static <T> ResponseEntity<T> ok(T body) {**

**return new ResponseEntity<>(body, HttpStatus.OK);**

**}**

1. Yani şu yapılır:

* body nesnesi alınıp HTTP yanıt gövdesine koyulur.
* Status kodu 200 OK olarak ayarlanır.
* Yeni bir ResponseEntity nesnesi oluşturulur ve return edilir.

## HTTP 204 No Content Durum Kodu (Status Code)

* HTTP protokolünde 204 kodu şöyle tanımlanır:

"The server has successfully fulfilled the request and that there is no additional content to send in the response payload body."

* Yani:
  + İstek başarılı.
  + Ancak, istemciye dönülecek bir içerik (örneğin JSON, HTML vs.) yok.
* Özellikle **silme (DELETE)** gibi işlemlerde yaygın olarak kullanılır.

## Neden 204 No Content kullanılır?

1. **Silme işlemleri için standarttır:**
   * Bir kaynak (örneğin ürün) başarıyla silindiğinde,
   * İstemciye bilgi vermek yeterlidir,
   * Fazladan bir veri göndermek gerekmez.
2. **Veri döndürülmemesi gerektiğinde:**
   * Örneğin, güncelleme sonrası istemciyi yeniden veriyle doldurmak istemiyorsanız,
   * Ya da işlem başarılı ama cevap gövdesine gerek yoksa.
3. **Performans ve Ağ Trafiği açısından avantajlıdır:**
   * Gövde gönderilmediği için trafik ve kaynak kullanımı azalır.

| **Durum** | **Dönülen HTTP Durum Kodu** | **Gövde İçeriği** | **Ne Zaman Kullanılır?** |
| --- | --- | --- | --- |
| Başarılı veri döndürme | 200 OK | Güncellenmiş veya sorgulanan veri | Veri istemciye gönderilmek istendiğinde |
| Kaynak oluşturuldu | 201 Created | Oluşturulan kaynak bilgisi | Yeni kaynak yaratıldıysa |
| Başarı ama içerik yok | 204 No Content | Gövde yok | Silme işlemi veya boş yanıt gerektiğinde |
| Kaynak bulunamadı | 404 Not Found | Hata mesajı veya boş | İstenen kaynak yoksa |

## Build()

## build() Metodu Nedir? Ne İşe Yarar?

* build() metodu, ResponseEntity.BodyBuilder üzerinde çağrılır.
* Görevi:

Builder tasarım desenini kullanarak, tüm konfigürasyonlar tamamlandıktan sonra **ResponseEntity nesnesini oluşturup döndürmek**.

* Yani build() metodunu çağırmadan, HTTP cevabını temsil eden ResponseEntity nesnesi hazır olmaz.

build() **Metodunun İmzası ve Dönüş Tipi**

**public ResponseEntity<T> build()**

 Bu metodun generic tip parametresi vardır.

 Genellikle:

* ResponseEntity<Void> şeklinde kullanılır,
* Çünkü 204 No Content cevabında **body yoktur**.

## Neden build() Çağrılır?

* Çünkü builder deseni kullanılırken noContent() gibi metodlar **ResponseEntity.Builder** döner.
* Builder üzerinde yapılan tüm ayarlamalardan sonra **son adımda build çağrılır** ve gerçek nesne üretilir.
* build() çağrılmazsa, sadece yapılandırıcı nesne olur, HTTP cevabı oluşturulmaz.

| **Kavram** | **Açıklama** |
| --- | --- |
| ResponseEntity | HTTP cevabını temsil eden sınıf |
| noContent() | HTTP 204 No Content durum kodunu ayarlayan builder metodu |
| build() | Builder üzerinden nesne oluşturma metodudur. HTTP cevabı hazırlar |
| ResponseEntity.noContent().build() | 204 kodlu boş gövdeli HTTP cevabı oluşturur ve döner |

## ****Endpoint****

## ****Endpoint Nedir?****

**Endpoint**, bir uygulamanın dış dünya ile iletişim kurduğu **erişim noktasıdır**.

* Genellikle **URL + HTTP methodu (GET, POST, PUT, DELETE)** kombinasyonudur.
* **Bir işlemi tetikler**: veri alma, veri gönderme, veri silme vb.
* REST API'lerde, endpoint'ler bir **controller** sınıfı içinde tanımlanır.

## Ne İşe Yarar?

* **Dış dünyadan gelen HTTP isteklerini karşılar.**
* Belirli bir işlevi gerçekleştirir (örneğin, kullanıcı kaydetme, ürün listeleme).
* **Veri alışverişi sağlar**: JSON, XML, vb.

| **HTTP Method** | **Endpoint** | **Anlamı** |
| --- | --- | --- |
| GET | /products | Tüm ürünleri getir |
| POST | /products | Yeni ürün oluştur |
| PUT | /products/{id} | Ürünü güncelle |
| DELETE | /products/{id} | Ürünü sil |

## Nerelerde Kullanılır?

* **Spring MVC / Spring Boot**
* **Jakarta EE (eski adıyla Java EE)**
* REST API'ler, SOAP servisleri
* Mobil/web istemcilerle veri alışverişinde

**@RequestParam**

@RequestParam, genellikle HTTP **GET** (ve bazen **POST**) isteklerinde URL'de gönderilen **anahtar-değer (key-value)** çiftlerini almak için kullanılır.

## Ne Zaman Kullanılır?

* **GET veya POST isteklerinde** URL üzerinden veri almak isteniyorsa.

REST standartlarına göre:

* **Kaynak tanımlayıcıları** (ID’ler, entity'ler): → @PathVariable
* **Filtreleme, seçim, opsiyonlar**: → @RequestParam

 @PathVariable → **Kaynağın kimliğini** temsil eder (örneğin müşteri, sepet).

 @RequestParam → **İşlem için gerekli seçimi/filtreyi** temsil eder (örneğin hangi ürün?).

**@Builder**

## @Builder Nedir?

* @Builder, **Lombok** kütüphanesinin sağladığı bir anotasyon.
* Bir sınıf için **Builder Pattern**’i otomatik olarak üretir.
* Builder Pattern, nesne oluştururken “adım adım” alanları set etmemizi sağlar.

## Nerede Kullanmalı?

* DTO’larda (özellikle response’larda)
* Alan sayısı fazla olan entity’lerde
* Immutable nesneler tasarlarken (setter koymak istemiyorsan)

@Builder, nesne oluşturmayı **daha güvenli, okunabilir ve esnek** hale getiriyor.  
Setter/constructor yerine builder pattern’i tercih etmek, özellikle DTO katmanında **best practice** sayılır.

SQL

Product en çok ekleneni bul **+**  
inner query kullanabileceğim bir senaryo yaz **+**  
order bir müşterinin ne kadar siparişi var **+**  
price bütün ürünlerin fiyatlarını sırala düşükten yükseğe doğru **+**

Her service in bir interface i olacak **+**

Controller a yeni bir service ekle **+**

Customer Repository de A harfi ile başlayan müşterileri getirmesini isteyeceğim **+**

Cart controller -> remove düzenle. **+**

SELECT \*

FROM product

WHERE price > (

SELECT AVG(price)

FROM product

);

**Açıklama:**

* İç sorgu → ürünlerin ortalama fiyatını bulur.
* Dış sorgu → bu ortalama fiyatın üzerindeki ürünleri listeler.

SELECT \*

FROM product

WHERE price = (SELECT MIN(price) FROM product)

OR price = (SELECT MAX(price) FROM product);

**Açıklama:**

* İç sorgular → ürün tablosundaki en düşük ve en yüksek fiyatları bulur.
* Dış sorgu → bu fiyatlara sahip ürünleri getirir.

SELECT \*

FROM product

WHERE price > (SELECT AVG(price) FROM product)

AND stock = (

SELECT MAX(stock)

FROM product

WHERE price > (SELECT AVG(price) FROM product)

);

**Açıklama:**

* İlk iç sorgu → ürünlerin ortalama fiyatını bulur.
* İkinci iç sorgu → ortalamanın üstündeki ürünlerin en yüksek stok miktarını bulur.
* Dış sorgu → hem fiyatı ortalamanın üstünde hem de stok miktarı maksimum olan ürünleri getirir.